7 Number Theory

In which, after becoming separated, our heroes arrange a place to meet, by sending messages that stay secret even as snooping spies listen in.

7.1 Why You Might Care

> I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind.
>
> Sir William Thomson, Lord Kelvin (1824–1907)
> lecture on "Electrical Units of Measurement" (1883)

A chapter about numbers (particularly when it's so far along in this book!) probably seems a little bizarre—after all, what is there to say about numbers that you didn't figure out by elementary school?!? But, more so than any other chapter of the book, the technical material in this chapter leads directly to a single absolutely crucial (and ubiquitous) modern application of computer science: *cryptography,* which deals with protocols to allow multiple parties to communicate securely, even in the presence of eavesdropping adversaries. (The word *cryptography* comes from the Greek *kryptos* "concealed/secret" and *graph* "writing.") Cryptographic systems are used throughout our daily lives—both in the security layers that connect us as users to servers (for example, in banking online or in registering for courses at a college), and in the backend systems that, we hope, protect our data even when we aren't interacting with it.

Our goal in this chapter will be to build up the technical machinery necessary to define and understand the *RSA cryptosystem,* one of the most commonly used cryptographic systems today. (RSA is named after the initials of its three discoverers, Rivest, Shamir, and Adleman.) By the end of the chapter, in Section 7.5, we'll be able to give a full treatment of RSA, along with sketched outlines of a few other important ideas from cryptography. (Later in the book, in Chapter 9, we'll also encounter the historical code*breaking* work of Alan Turing and colleagues, which deciphered the German encryption in World War II—a major part of the allied victory. See p. 9-75.)

To get there, we'll need to develop some concepts and tools from *number theory.* ("Number theory" is just a slightly fancy name for "arithmetic on integers.") Our focus will be on *modular arithmetic*: that is, the numbers on which we'll be doing arithmetic will be a set of integers $\{0, 1, 2, \ldots, n-1\}$, where—like on a clock—the numbers "wrap around" from $n-1$ back to 0. In other words, we'll interpret numerical expressions *modulo n,* always considering each expression via its remainder when we divide by n. Modular arithmetic shares a lot of properties with elementary-school arithmetic, and we'll start by noting and exploring those commonalities. We begin in Section 7.2 with formal definitions of modular arithmetic, and the adaptation of some basic ideas from elementary-school arithmetic (addition, multiplication, greatest common divisors, least common multiples—and efficient algorithms for all of these operations) to this new setting. We'll then turn in Section 7.3 to *primality* (when a number has no divisors other than 1 and itself) and *relative primality* (when two numbers have no common divisors other than 1).

Modular arithmetic and elementary-school arithmetic begin to diverge more substantially when we start to think about division. There's no integer that's one eighth of 4 (because $\frac{1}{2}$ isn't an integer). But, on a clock where we treat 12:00 as 0, there *is* an integer that's a eighth of 4—namely 2, because $2+2+2+2+$

$2 + 2 + 2 + 2$ is 4 (because 4:00pm is 16 hours after midnight—so $8 \cdot 2$ *is* 4, modulo 12). In Section 7.4, we'll explore exactly what division means in modular arithmetic—and some special features of division that arise when n is a prime number.

As we go, we'll see a few other applications of number theory: to error-correcting codes, secret sharing, and the apparently unrelated task of generating all 4-letter sequences (`AAAA` to `ZZZZ`). And, finally, we'll put the pieces together to explore RSA.

7.2 Modular Arithmetic

> It's always a mistake to try for universal approbation, universal approval, because if you fear making anyone mad, then you ultimately probe for the lowest common denominator of human achievement.
>
> Jimmy Carter (b. 1924)
> Remarks at the National Convention of the Future Farmers of America (1978)

We will start with a few reminders of some basic arithmetic definitions from Chapter 2—multiplication, division, modular arithmetic—as these concepts are the foundations for all the work that we'll do in this chapter. We'll also introduce a few algorithms for *computing* these basic arithmetic quantities, including one of the oldest known algorithms: the *Euclidean algorithm*, from about 2300 years ago, which computes the greatest common divisor of two integers n and m (the largest number that evenly divides both).

7.2.1 Remainders: A Reminder

Let's start with a few basic facts about integers. Every integer is 0 or 1 more than some even number. Every integer is 0, 1, or 2 more than a multiple of three. Every integer is at most 3 more than a multiple of four. And, in general, for any integer $k \geq 1$, every integer is r more than a multiple of k, for some $r \in \{0, 1, \ldots, k-1\}$. We'll begin with a precise statement and proof of the general version of this property:

Theorem 7.1: Floors and Remainders ("The Division Theorem").
Let $k \geq 1$ and n be integers. Then there exist integers d and r such that (i) $0 \leq r < k$, and (ii) $kd + r = n$. Furthermore, the values of d and r satisfying (i) and (ii) are unique.

Before we prove the theorem, let's look at a few examples of what it claims:

Example 7.1: Some examples of the Division Theorem.
For $k = 202$ and $n = 379$, the theorem states that there exist integers $r \in \{0, 1, \ldots, 201\}$ and d with $202d + r = 379$. Specifically, those values are $r = 177$ and $d = 1$, because $202 \cdot 1 + 177 = 379$.

Here are a few more examples, still with $k = 202$:

$n = 55057$	$n = 507$	$n = 177$	$n = 404$	$n = -507$	$n = -404$
$d = 272$	$d = 2$	$d = 0$	$d = 2$	$d = -3$	$d = -2$
$r = 113$	$r = 103$	$r = 177$	$r = 0$	$r = 99$	$r = 0$

You can verify that, in each of these six columns, indeed we have $202d + r = n$.

Now let's give a proof of the general result:

Proof of Theorem 7.1. Consider a fixed integer $k \geq 1$. Let $P(n)$ denote the claim

$$P(n) = \text{there exist integers } d \text{ and } r \text{ such that } 0 \leq r < k \text{ and } kd + r = n.$$

We must prove that $P(n)$ holds for all integers n. We'll first prove the result for nonnegative n (by strong induction on n), and then show it for $n < 0$ (making use of the result for nonnegative n).

> *Problem-solving tip:* To prove that a property is true for all inputs, it often turns out to be easier to *first* prove a special case and then use that special case to show that the property holds in general. (Another example: it's probably easier to analyze the performance of Merge Sort on inputs whose size is an exact power of 2, and to then generalize to arbitrary input sizes.)

Case I: $n \geq 0$. We'll prove that $P(n)$ holds for all $n \geq 0$ by strong induction on n.

For the base cases ($0 \leq n < k$), we simply select $d = 0$ and $r = n$. Indeed, these values guarantee that $0 \leq r < k$ and $kd + r = k \cdot 0 + n = 0 + n = n$.

For the inductive case ($n \geq k$), we assume the inductive hypotheses—namely, we assume $P(n')$ for any $0 \leq n' < n$—and we must prove $P(n)$. Let $n' = n - k$. Because $n \geq k$ and $k > 0$, it is immediate that n' satisfies $0 \leq n' < n$. Thus we can apply the inductive hypothesis $P(n')$ to conclude that there exist integers d' and r' such that $0 \leq r' < k$ and $kd' + r' = n'$. Select $d = d' + 1$ and $r = r'$. Thus, indeed, $0 \leq r < k$ and

$$\begin{aligned} kd + r &= k(d' + 1) + r' && \textit{definition of } d \textit{ and } r \\ &= kd' + k + r' && \textit{distributive property} \\ &= n' + k && n' = kd' + r', \textit{ by definition} \\ &= n. && \textit{definition of } n' = n - k \end{aligned}$$

Case II: $n < 0$. To show that $P(n)$ holds for an arbitrary $n < 0$, we will make use of Case I. Let r' and d' be the integers guaranteed by $P(-n)$, so that $kd' + r' = -n$. We consider two cases based on whether $r' = 0$:

Case IIA: $r' \neq 0$. Then let $d = -d' - 1$ and let $r = k - r'$. (Because $k > r' > 0$, we have $0 < k - r' < k$.) Thus

$$\begin{aligned} kd + r &= k(-d' - 1) + k - r' && \textit{definition of } d \textit{ and } r \\ &= -(kd' + r) && \textit{algebraic simplification} \\ &= -(-n) = n. && \textit{definition of } d' \textit{ and } r' \end{aligned}$$

Case IIB: $r' = 0$. Then let $d = -d'$ and $r = r' = 0$. Therefore

$$\begin{aligned} kd + r &= -d'k + r' && \textit{definition of } d \textit{ and } r \\ &= -(-n) = n. && \textit{definition of } d' \textit{ and } r' \end{aligned}$$

We have thus proven that $P(n)$ holds for all integers n: Case I handled $n \geq 0$, and Case II handled $n < 0$. (We have not yet proven the uniqueness of the integers r and d; see Exercise 7.4.) □

This theorem now allows us to give a more careful definition of modular arithmetic. (In Definition 2.11, we gave the slightly less formal definition of n mod k as the remainder when we divide n by k.)

Definition 7.2: Modulus [reprise].
For integers $k > 0$ and n, the quantity n mod k is the unique integer r such that $0 \leq r < k$ and $kd + r = n$ for some integer d (whose existence is guaranteed by Theorem 7.1).

Incidentally, the integer d whose existence is guaranteed by Theorem 7.1 is $\lfloor \frac{n}{k} \rfloor$ (just as Definition 7.2 says that the integer r is n mod d). For any $k \geq 1$ and any integer n, we can write n as $n = \left\lfloor \frac{n}{k} \right\rfloor \cdot k + (n \bmod k)$.

Taking it further: One of the tasks that we can accomplish conveniently using modular arithmetic is *base conversion* of integers. We're used to writing numbers in decimal ("base 10"), where each digit is "worth" a factor of 10 more than the digit to its right. (For example, the number we write "31" means $1 \cdot 10^0 + 3 \cdot 10^1 = 1 + 30$.) Computers store numbers in binary ("base 2") representation, and we can convert between bases using modular arithmetic. For more, see p. 7-16.

7.2.2 Computing n mod k and $\lfloor \frac{n}{k} \rfloor$

So far, we've taken arithmetic operations for granted—ignoring *how* we'd figure out the numerical value of an arithmetic expression like $2^{1024} - 3^{256} \cdot 5^{202}$, which is simple to write, but whose value is definitely not so instantaneous to calculate. (Quick! Is $2^{1024} - 3^{256} \cdot 5^{202}$ evenly divisible by 7?) Indeed, many of us spent a lot of time in elementary-school math classes learning algorithms for basic arithmetic operations like addition, multiplication, long division, and exponentiation (even if back then nobody told us that they were called algorithms).

Thinking about algorithms for some basic arithmetic operations will be useful, for multiple reasons: because they're surprisingly relevant for proving some useful facts about modular arithmetic, and because computing them efficiently turns out to be crucial in the cryptographic systems that we'll explore in Section 7.5. We'll start with the algorithm shown in Figure 7.1, which computes n mod k (and simultaneously computes $\lfloor \frac{n}{k} \rfloor$ too). The core idea for this algorithm was implicit in the proof of Theorem 7.1: we repeatedly subtract k from n until we reach a number in the range $\{0, 1, \ldots, k-1\}$.

mod-and-div(n, k):
Input: integers $n \geq 0$ and $k \geq 1$
Output: n mod k and $\lfloor \frac{n}{k} \rfloor$
1 $r := n$; $d := 0$
2 **while** $r \geq k$:
3 $\quad r := r - k$; $d := d + 1$
4 **return** r, d

Each iteration of the execution is shown: first $r = 64$ and $d = 0$, then $r = 59$ and $d = 1$, etc.

mod-and-div$(64, 5)$:

$r = 64$	59	54	49	44	39	34	29	24	19	14	9	4
$d = 0$	1	2	3	4	5	6	7	8	9	10	11	12

return values: $r = 4$ and $d = 12$

(Indeed, we can write $64 = 12 \cdot 5 + 4$, where $4 = 64 \bmod 5$ and $12 = \lfloor \frac{64}{5} \rfloor$.)

Figure 7.1 An algorithm to compute n mod k and $\lfloor \frac{n}{k} \rfloor$, and an example of its execution.

Example 7.2: An example of mod-and-div.
Figure 7.1 shows the computation of **mod-and-div**$(64, 5)$. We start by setting $r := 64$ and $d := 0$, and repeatedly decrease r by 5 and increase d by 1 until $r < 5$. Thus **mod-and-div**$(64, 5)$ returns 4 and 12.

Similarly, **mod-and-div**$(20, 17)$ starts with $d = 0$ and $r = 20$, and executes one (and only one) iteration of the loop, returning $d = 1$ and $r = 3$.

(The algorithm in Figure 7.1 is called **mod-and-div** because some programming languages—Pascal, for one [admittedly dated] example—use `div` to denote integer division, so that `15 div 7` is 2. In C++, slightly disconcertingly, a function called `div` computes the same pair of integers that our **mod-and-div** computes.)

The **mod-and-div** algorithm is fairly intuitive, if fairly slow: it simply keeps removing multiples of k from n until there are no multiples of k left to remove. (For simplicity, the algorithm as written in Figure 7.1 only handles the case where $n \geq 0$; you'll extend the algorithm to handle negative n in Exercise 7.10. See Example 7.1 for some examples of the Division Theorem with $n < 0$.)

Lemma 7.3: Correctness and efficiency of mod-and-div.
For any integers $n \geq 0$ and $k \geq 1$, calling **mod-and-div**(n, k) returns n mod k and $\lfloor \frac{n}{k} \rfloor$, using a total of $\Theta(\lfloor \frac{n}{k} \rfloor)$ arithmetic operations.

Proof. We claim that, throughout the execution of the algorithm, we have $dk + r = n$ (and also $r \geq 0$). This fact follows by induction on the number of iterations of the while loop in **mod-and-div**: it's true before the loop starts (when $r = n$ and $d = 0$), and if it's true before an iteration of the while loop then it's true after that iteration (when dk has increased by k, and r has decreased by k). Furthermore, when the while loop terminates, we also have that $r < k$. Thus the returned values satisfy $dk + r = n$ and $0 \leq r < k$—precisely as required by Definition 7.2.

The total number of iterations of the while loop is exactly the returned value of $d = \lfloor \frac{n}{k} \rfloor$, and we do three arithmetic operations per iteration: a comparison (is $r \geq k$?), a subtraction (what's $r - k$?), and an addition (what's $d + 1$?). Thus the total number of arithmetic operations is $3 \lfloor \frac{n}{k} \rfloor + \Theta(1) = \Theta(\lfloor \frac{n}{k} \rfloor)$. □

Should we consider **mod-and-div** fast? Not really! Imagine using this algorithm to determine whether, say, $n = 123{,}456{,}789$ is divisible by 7. (It isn't: $n = 17{,}636{,}684 \cdot 7 + 1$.) Our algorithm would take over 10 million iterations ($\lfloor \frac{n}{7} \rfloor > \frac{70{,}000{,}000}{7} = 10{,}000{,}000$) iterations to compute the answer—and that seems (and is!) very slow. One way to think about **mod-and-div**(n, k) is that it performs a *linear search* for the integer d such that $kd \leq n < (k + 1)d$: we keep increasing d by one until this property holds. We could instead give a much faster algorithm based on *binary search* to find that value of d. (See Exercises 7.11–7.16.) The improved algorithm requires only logarithmically many arithmetic operations—an exponential improvement over **mod-and-div**.

Taking it further: What should count as an efficient algorithm when the inputs are numbers? In previous chapters, we've talked about the generally accepted definition of *efficient* as meaning "requiring a number of steps that is polynomial in the size of the input." That is, on an input of size n, our algorithm should run in at most $O(n^c)$ steps, for some fixed c. (See p. 6-35.) So why did we say that an algorithm like **mod-and-div** isn't efficient? After all, on input n and k, Lemma 7.3 says that the algorithm took only about $\frac{n}{k}$ steps. But the key point is that an algorithm that takes a numerical input n does *not* receive an input of size n. The number 123,456,789 takes only 9 characters (the nine digits in the number!) to write down—not 123,456,789 characters. (Unless you wrote down the numbers in *unary,* using tally marks instead of digits: 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 ⋯.)

Generally, an algorithm that takes a number n as input receives that number *n written in binary.* The binary representation of n uses $\lceil \log_2 n \rceil$ bits. As usual, we consider an algorithm to be efficient if it takes time that's polynomial in the number of bits of the input—so we consider an algorithm that takes a number n as input to be efficient *if it requires a number of operations that is at most* $(\log_2 n)^c$ *for some fixed* c. (That is, the algorithm should run in time that is *polylogarithmic* in n.) Every grade-school algorithm that you learned for arithmetic—addition, subtraction, multiplication, long division, etc.—was efficient, requiring you to do a number of operations related to the number of digits in the numbers, and *not* to the value of the numbers themselves.

7.2.3 Congruences, Divisors, and Common Divisors

Lemma 7.3 says that **mod-and-div**(n, k), which repeatedly subtracts k from n in a loop, correctly computes the value of n mod k. We used induction to prove Lemma 7.3, but we could have instead argued for the correctness of the algorithm, perhaps more intuitively, via the following fact:

For any integers $a \geq 0$ and $k \geq 1$, we have $(a + k) \bmod k = a \bmod k$.

That is, the remainder when we divide an integer a by k isn't changed by adding an exact multiple of k to a. This property follows from the definition of mod, but it's also a special case of a useful general property of modular arithmetic, which we'll state (along with some other similar facts) in Theorem 7.4. Here are a few examples of this more general property:

Example 7.3: The mod of a sum, and the sum of the mods.
Consider the following expressions of the form $(a + b) \bmod k$.

- $(17 + 43) \bmod 7 = 60 \bmod 7 = 4$. (Note $17 \bmod 7 = 3$, $43 \bmod 7 = 1$, and $3 + 1 = 4$.)
- $(18 + 42) \bmod 9 = 60 \bmod 9 = 6$. (Note $18 \bmod 9 = 0$, $42 \bmod 9 = 6$, and $0 + 6 = 6$.)
- $(25 + 25) \bmod 6 = 50 \bmod 6 = 2$. (Note $25 \bmod 6 = 1$, $25 \bmod 6 = 1$, and $1 + 1 = 2$.)

Based on these three examples, it might be tempting to conjecture that $(a + b) \bmod k$ is always equal to $(a \bmod k) + (b \bmod k)$, but be careful—this claim has a bug, as this example shows:

- $(18 + 49) \bmod 5 = 67 \bmod 5 = 2$. (Note $18 \bmod 5 = 3$, $49 \bmod 5 = 4$, but $3 + 4 \neq 2$.)

Instead, it turns out that $(a + b) \bmod k = [(a \bmod k) + (b \bmod k)] \bmod k$—we had to add an "extra" mod k at the end.

Here are some of the useful general properties of modular arithmetic:

Theorem 7.4: Properties of modular arithmetic.
For integers a and b and $k > 0$:

$$k \bmod k = 0 \tag{7.4.1}$$

$$a + b \bmod k = [(a \bmod k) + (b \bmod k)] \bmod k \tag{7.4.2}$$

$$ab \bmod k = [(a \bmod k) \cdot (b \bmod k)] \bmod k \tag{7.4.3}$$

$$a^b \bmod k = [(a \bmod k)^b] \bmod k. \tag{7.4.4}$$

Again notice the "extra" mod k at the end of the last three of these equations—remember that it is not the case that $ab \bmod k = (a \bmod k) \cdot (b \bmod k)$ in general. For example, $14 \bmod 6 = 2$ and $5 \bmod 6 = 5$, but $(14 \cdot 5) \bmod 6 = 4 \neq 2 \cdot 5$. (We could give formal proofs of these properties based on the definitions of mod, but we'll omit them here. Exercise 7.17 asks you to give a formal proof for one of these properties.)

In the cryptographic applications that we will explore later in this chapter, it will turn out to be important to perform "modular exponentiation" efficiently—that is, we'll need to compute $b^e \bmod n$ very quickly, even when e is fairly large. Fortunately, Theorem 7.4.4 will help us do this computation efficiently; see Exercises 7.23–7.25.

Congruences

We've now talked a little bit (in Theorem 7.4, for example) about two numbers a and b that have the same remainder when we divide them by k—that is, with $a \bmod k = b \bmod k$. There's useful terminology, and notation, for this kind of equivalence:

Definition 7.5: Congruence.
Two integers a and b are *congruent mod* k, written $a \equiv_k b$, if $a \bmod k = b \bmod k$.

(Typically $a \equiv_k b$ is read as "a is equivalent to b mod k" or "a is congruent to b mod k." If you read the statement $a \equiv_k b$ out loud, it's polite to pause slightly, as if there were a comma, before the "mod k" part.)

> **Taking it further:** Some people write $a \equiv_k b$ using the notation $a \equiv b \pmod{k}$. This notation is used to mean the same thing as our notation $a \equiv_k b$, but note the somewhat unusual precedence in this alternate notation: it says that $[a \equiv b] \pmod{k}$ (and it does not, as it might appear, say that the quantity a and the quantity $[b \bmod k]$ are equivalent).

Divisors, factors, and multiples

We now return to the *divisibility* of one number by another, when the first is an exact multiple of the second. As with the previous topics in this section, we gave some preliminary definitions in Chapter 2 of divisibility (and related terminology), but we'll again repeat the definitions here, and also go into a little bit more detail.

Definition 7.6: Divisibility, factors, and multiples [reprise].
For two integers $k > 0$ and n, we write $k \mid n$ to denote the proposition that $n \bmod k = 0$. If $k \mid n$, we say that k *divides* n (or that k *evenly divides* n), that n is a *multiple* of k, and that k is a *factor* of n.

For example, we can say that $42 \mid 714$, that 102 divides 714, that 119 evenly divides 714, that 6 and 17 are factors of 714, and that 714 is a multiple of 7.

Here are a few useful properties of division:

Theorem 7.7: Properties of divisibility.
For integers a and b and c:

$$a \mid 0 \tag{7.7.1}$$

$$1 \mid a \tag{7.7.2}$$

$$a \mid a \tag{7.7.3}$$

$$a \mid b \text{ and } b \mid c \quad \Rightarrow \quad a \mid c \tag{7.7.4}$$

$$a \mid b \text{ and } b \mid a \quad \Rightarrow \quad a = b \text{ or } a = -b \tag{7.7.5}$$

$$a \mid b \text{ and } a \mid c \quad \Rightarrow \quad a \mid (b + c) \tag{7.7.6}$$

$$a \mid b \quad \Rightarrow \quad a \mid bc \tag{7.7.7}$$

$$ab \mid c \quad \Rightarrow \quad a \mid c \text{ and } b \mid c \tag{7.7.8}$$

These properties generally follow fairly directly from the definition of divisibility. A few are left to you in the exercises, and we'll address a few others in Chapter 8, which introduces relations. (Some of these facts are some standard properties of some relations that the "divides" relation happens to have: *reflexivity* (7.7.3), *transitivity* (7.7.4), and so-called *antisymmetry* [a version of (7.7.5)]. See Chapter 8.) To give the flavor of these arguments, here's one of the proofs, that $ab \mid c$ implies that $a \mid c$ and $b \mid c$:

Proof of (7.7.8). Assume $ab \mid c$. Then, by definition of mod (and by Theorem 7.1), there exists an integer k such that $c = (ab) \cdot k$. Taking both sides mod a, we have

$$\begin{aligned}
c \bmod a &= abk \bmod a && k \text{ is the integer such that } c = (ab) \cdot k \\
&= [(a \bmod a) \cdot (bk \bmod a)] \bmod a && \text{Theorem 7.4.3} \\
&= [0 \cdot (bk \bmod a)] \bmod a && \text{Theorem 7.4.1} \\
&= 0 \bmod a && 0 \cdot x = 0 \text{ for any } x \\
&= 0. && 0 \bmod a = 0 \text{ for any } a
\end{aligned}$$

Thus $c \bmod a = 0$, so $a \mid c$. Analogously, because $b \cdot (ak) = c$, we have that $b \mid c$ too. □

Greatest common divisors and least common multiples

We now turn to our last pair of definitions involving division: for two integers, we'll be interested in two related quantities—the largest number that divides both of them, and the smallest number that they both divide.

Definition 7.8: Greatest Common Divisor [GCD].
The *greatest common divisor* of two positive integers n and m, denoted $\text{GCD}(n, m)$, is the largest $d \in \mathbb{Z}^{\geq 1}$ such that $d \mid n$ and $d \mid m$.

Definition 7.9: Least Common Multiple [LCM].
The *least common multiple* of two positive integers n and m, denoted $\text{LCM}(n, m)$, is the smallest $d \in \mathbb{Z}^{\geq 1}$ such that $n \mid d$ and $m \mid d$.

Here are some examples of both GCDs and LCMs, for a few pairs of small numbers:

Example 7.4: Examples of GCDs and LCMs.
The greatest common divisor of 6 and 27 is 3, because 3 divides both 6 and 27 (and no integer $k \geq 4$ divides both). The least common multiple of 6 and 27 is 54, because 6 and 27 both divide 54 (and no integer $k \leq 53$ is divided by both). Similarly,

$$\begin{array}{lll} \text{GCD}(1,9) = 1 & \text{GCD}(12,18) = 6 & \text{GCD}(202,505) = 101 \\ \text{LCM}(1,9) = 9 & \text{LCM}(12,18) = 36 & \text{LCM}(202,505) = 1010 \end{array} \quad \text{and} \quad \begin{array}{l} \text{GCD}(11,202) = 1 \\ \text{LCM}(11,202) = 2222. \end{array}$$

Both of these concepts might be (at least vaguely!) familiar from elementary school, specifically from when you learned about how to manipulate fractions:

Example 7.5: Fractions in lowest terms.
We can rewrite the fraction $\frac{38}{133}$ as $\frac{2}{7}$, by dividing both numerator and denominator by the common factor 19—and we can't reduce it further because 19 is the *greatest* common divisor of 38 and 133. (We have "reduced the fraction to lowest terms.")

We can rewrite the sum $\frac{5}{12} + \frac{7}{18}$ as $\frac{15}{36} + \frac{14}{36}$ (which equals $\frac{29}{36}$) by rewriting both fractions with a denominator that's a common multiple of the denominators of the two addends—and we couldn't have chosen a smaller denominator, because 36 is the *least* common multiple of 12 and 18. (We have "put the fractions over the lowest common denominator.")

7.2.4 Computing Greatest Common Divisors

In the remainder of this section, we'll turn to the task of *efficiently computing* the greatest common divisor of two integers. (Using this algorithm, we can also find least common multiples quickly, because GCDs and LCMs are closely related: for any two positive integers n and m, we have $\text{LCM}(n, m) \cdot \text{GCD}(n, m) = n \cdot m$.)

```
Euclid(n, m):
Input: positive integers n and m ≥ n
Output: GCD(n, m)
1 if m mod n = 0 then
2     return n
3 else
4     return Euclid(m mod n, n)
```

Euclid$(17, 42)$
$= \textbf{Euclid}(\underbrace{42 \bmod 17}_{=8}, 17)$ $42 \bmod 17 = 8 \neq 0$, so we're in the else case.
$= \textbf{Euclid}(\underbrace{17 \bmod 8}_{=1}, 8)$ $17 \bmod 8 = 1 \neq 0$, so we're in the else case again
$= 1$ $8 \bmod 1 = 0$, so we're done, and we return 1.

(Indeed, the only positive integer that divides both 17 and 42 is 1, so GCD$(17, 42) = 1$.)

Figure 7.2 The Euclidean algorithm for GCDs, and an example of its execution.

The "obvious" way to compute the greatest common divisor of n and m is to try all candidate divisors $d \in \{1, 2, \ldots, \min(n, m)\}$ and to return the largest value of d that indeed evenly divides both n and m. This algorithm is slow—very slow!—but there is a more efficient approach. Amazingly, a faster way to compute GCDs has been known for approximately 2300 years: the *Euclidean algorithm*, named after the Greek geometer Euclid, who lived in the 3rd century BCE. (Euclid is also the namesake of the *Euclidean distance* between points in the plane—see Exercise 2.175—among a number of other things in mathematics.) The algorithm is shown in Figure 7.2. Here are three small examples of the Euclidean algorithm in action:

Example 7.6: GCDs using the Euclidean algorithm.
Figure 7.2 shows the calculation of GCD$(17, 42) = 1$ using the Euclidean algorithm. Here are two more examples. First, for 48 and 1024,

$$\begin{aligned} \textbf{Euclid}(48, 1024) &= \textbf{Euclid}(\underbrace{1024 \bmod 48}_{=16}, 48) && 1024 \bmod 48 = 16 \neq 0, \textit{ so we're in the else case.} \\ &= 16, && 48 \bmod 16 = 0, \textit{ so we return } 16. \end{aligned}$$

and, second (written more compactly), for 91 and 287,

$$\textbf{Euclid}(91, 287) = \textbf{Euclid}(\underbrace{287 \bmod 91}_{=14}, 91) = \textbf{Euclid}(\underbrace{91 \bmod 14}_{=7}, 14) = 7.$$

Taking it further: Euclid described his algorithm in his book *Elements*, from c. 300 BCE, a multivolume opus covering the fundamentals of mathematics, particularly geometry, logic, and proofs. Most people view the Euclidean algorithm as the oldest nontrivial algorithm that's still in use today; there are some older not-quite-fully-specified procedures for basic arithmetic operations like multiplication that date back close to 2000 BCE, but they're not quite laid out as algorithms.

Donald Knuth—the 1974 Turing Award winner, the inventor of TEX (the underlying system that is used to typeset virtually all scholarly materials in computer science—including this book), and a staunch advocate and practitioner of expository writing about computer science in general and algorithms in particular—describes the history of the Euclidean algorithm (among many other things!) in *The Art of Computer Programming*, his own modern-day version of a multivolume opus covering the fundamentals of computer science, particularly algorithms, programming, and proofs [72]. Among the fascinating things that Knuth points out about the Euclidean algorithm is that Euclid's "proof" of correctness only handles the case of up to three iterations of the algorithm—because, Knuth argues, Euclid predated the idea of mathematical induction by hundreds of years. (And Euclid's version of the algorithm is quite hard to read, in part because Euclid didn't have a notion of zero, or the idea that 1 is a divisor of any positive integer n.)

The intuition of the algorithm, and making the intuition formal

Before we try to prove the correctness of the Euclidean algorithm, let's spend a few moments on the intuition behind it. The basic idea is that any common divisor of two numbers must also evenly divide their difference. For example, does 7 divide both 63 and 133? If so, then it would have to be the case that $7 \mid 63$ *and* that 7 also divides the "gap" between 133 and 63. (That's because $63 = 7 \cdot 9$, and if $7k = 133$, then $7(k-9) = 133 - 63$.) More generally, suppose that d is a common divisor of n and $m \geq n$. Then it must be the case that d divides $m - cn$, *for any integer c where $cn < m$.* In particular, d divides $m - \lfloor \frac{m}{n} \rfloor \cdot n$; that is, d divides $m \bmod n$. (We've only argued that if d is a common divisor of n and m then d must also divide $m \bmod n$, but actually the converse holds too; we'll formalize this fact in the proof.) See Figure 7.3.

We will now make this intuition formal, and give a full proof of the correctness of the Euclidean algorithm: that is, we will establish that $\textbf{Euclid}(n,m) = \text{GCD}(n,m)$ for any positive integers n and $m \geq n$, with a proof by induction. There's a crucial lemma that we'll need to prove first, based on the intuition we just described: we need to show that for any n and $m \geq n$ where $m \bmod n \neq 0$, we have $\text{GCD}(n,m) = \text{GCD}(n, m \bmod n)$. We will prove this fact by proving that *the common divisors of* $\{n,m\}$ are identical to *the common divisors of* $\{n, m \bmod n\}$. (Thus the *greatest* common divisor of these two pairs of integers will be identical.)

Lemma 7.10: When $n \nmid m$, the same divisors of n divide m and m mod n.
Let n and m be positive integers such that $n \leq m$ and $n \nmid m$. Let $d \mid n$ be an arbitrary divisor of n. Then $d \mid m$ if and only if $d \mid (m \bmod n)$.

Here's a concrete example before we prove the lemma:

Example 7.7: An example of Lemma 7.10.
Consider $n = 42$ and $m = 98$. Then $n \leq m$ and $n \nmid m$, as Lemma 7.10 requires. The divisors of 42 are $\{1,2,3,6,7,14,21,42\}$. Of these divisors, the ones that also divide 98 are $\{1,2,7,14\}$.

The lemma claims that the common divisors of 42 and $98 \bmod 42 = 14$ are also precisely $\{1,2,7,14\}$. And they are: because $14 \mid 42$, all divisors of 14—namely, 1, 2, 7, and 14—are common divisors of 14 and 42.

Proof of Lemma 7.10. By the assumption that $d \mid n$, we know that there's an integer a such that $n = ad$. Let $r = m \bmod n$, so that $m = cn + r$ for an integer c (as guaranteed by Theorem 7.1). We must prove that $d \mid m$ if and only if $d \mid r$.

For the forward direction, suppose that $d \mid m$. (We must prove that $d \mid r$.) By definition, there exists an integer b such that $m = bd$. But $n = ad$ and $m = bd$, so

$$m = cn + r \quad \Leftrightarrow \quad bd = c(ad) + r \quad \Leftrightarrow \quad r = (b - ac)d$$

for integers a, b, and c. Thus r is a multiple of d, and therefore $d \mid r$.

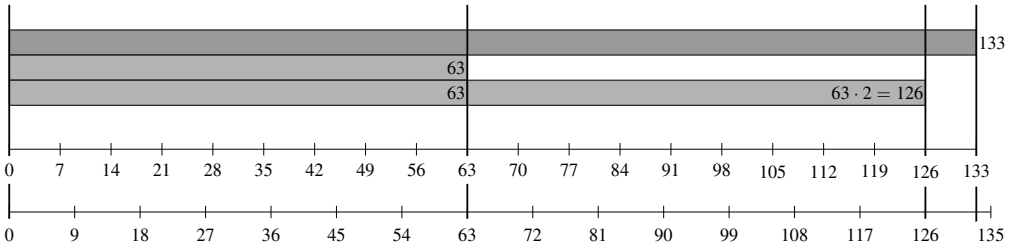


Figure 7.3 The intuition of the Euclidean algorithm: d is a common divisor of 63 and 133 if and only if d also divides $133 - 63$ and $133 - 63 \cdot 2 = 133 - 126$. Indeed $d = 7$ is a common divisor, but 9 is not (because 9 does not divide $133 - 126 = 7$).

For the converse, suppose that $d \mid r$. (We must prove that $d \mid m$.) By definition, we have that $r = bd$ for some integer b. But then $n = ad$ and $r = bd$, so

$$m = cn + r \quad = \quad c(ad) + bd \quad = \quad (ac + b)d$$

for integers a, b, and c. Thus $d \mid m$. □

Corollary 7.11. Let n and m be any two positive integers such that $n \leq m$ and $n \nmid m$. Then $\text{GCD}(n, m) = \text{GCD}(m \bmod n, n)$.

Proof. Lemma 7.10 establishes that the *set* of common divisors of $\langle n, m \rangle$ is identical to the set of common divisors of $\langle n, m \bmod n \rangle$. Therefore the *maxima* of these two sets of divisors—that is, $\text{GCD}(n, m)$ and $\text{GCD}(m \bmod n, n)$—are also equal. □

Putting it together: the correctness of the Euclidean algorithm

Using this corollary, we can now prove the correctness of the Euclidean algorithm:

Theorem 7.12: Correctness of the Euclidean algorithm.
For arbitrary positive integers n and m with $n \leq m$, we have $\textbf{Euclid}(n, m) = \text{GCD}(n, m)$.

Proof. We'll proceed by strong induction on n, the smaller input. Define the property

$$P(n) = \text{for any } m \geq n, \text{ we have } \textbf{Euclid}(n, m) = \text{GCD}(n, m).$$

We'll prove that $P(n)$ holds for all integers $n \geq 1$.

Base case ($n = 1$). $P(1)$ follows because both $\text{GCD}(1, m) = 1$ and $\textbf{Euclid}(1, m) = 1$: for any m, the *only* positive integer divisor of 1 is 1 itself (and indeed $1 \mid m$), and thus $\text{GCD}(1, m) = 1$. Observe that $\textbf{Euclid}(1, m) = 1$, too, because $m \bmod 1 = 0$ for any m.

Inductive case ($n \geq 2$). We assume the inductive hypotheses—that $P(n')$ holds for any $1 \leq n' < n$—and must prove $P(n)$. Let $m \geq n$ be arbitrary. There are two subcases, based on whether $n \mid m$ or $n \nmid m$:

Case I: $n \mid m$. In other words, $m = cn$ for an integer c, which means that $m \bmod n = 0$ and thus, by inspection of the algorithm, $\textbf{Euclid}(n, m) = n$. Because $n \mid n$ (and there is no $d > n$ that divides n evenly), indeed n is the GCD of n and $m = cn$.

Case II: $n \nmid m$. Because $n \nmid m$—that is, because $m \bmod n \neq 0$—we have

$$\begin{aligned} \textbf{Euclid}(n, m) &= \textbf{Euclid}(m \bmod n, n) && \textit{by inspection of the algorithm} \\ &= \text{GCD}(m \bmod n, n) && \textit{by the inductive hypothesis } P(m \bmod n) \\ &= \text{GCD}(n, m). && \textit{by Corollary 7.11} \end{aligned}$$

Note that $(m \bmod n) \leq n - 1$ by the definition of mod (*anything* mod n is less than n), so we can invoke the inductive hypothesis $P(m \bmod n)$ in the second step of this proof. □

Theorem 7.12 establishes the *correctness* of the Euclidean algorithm, but we introduced this algorithm because the brute-force algorithm (simply testing every candidate divisor d) was too slow. So we'd better analyze its running time. Indeed, the Euclidean algorithm *is* very efficient:

Theorem 7.13: Efficiency of Euclidean Algorithm.
For arbitrary positive integers n and m with $n \leq m$, the recursion tree of $\textbf{Euclid}(n, m)$ has depth at most $\log n + \log m$.

(The ability to efficiently compute $\text{GCD}(n, m)$ using the Euclidean algorithm—assuming we use the efficient algorithm to compute $m \bmod n$ from Exercises 7.11–7.16, at least—will be crucial in the RSA cryptographic system in Section 7.5.) You'll prove Theorem 7.13 by induction in Exercise 7.34—and you'll show that the recursion tree can be as deep as $\Omega(\log n + \log m)$, using the Fibonacci numbers, in Exercise 7.37.

Problem-solving tip: In Theorem 7.13, it's not obvious what quantity upon which to perform induction—after all, there are two input variables, n and m. It is often useful to combine multiple inputs into a single "measure of progress" toward the base case—perhaps performing induction on the quantity $n + m$ or the quantity $n \cdot m$.

Computer Science Connections

Converting Between Bases, Binary Representation, and Generating Strings

For a combination of historical and anatomical reasons—we have ten fingers and ten toes!—we generally use a *base ten*, or *decimal*, system to represent numbers. ("Decimal" comes from the Latin *decim* "ten"—and it's suggestive that *digit* is ambiguous in English between "place in a number" and "finger or toe.") Moving from right to left, there's a ones place, a tens place, a hundreds place, and so forth; thus 2048 denotes $8 \cdot 1 + 4 \cdot 10 + 0 \cdot 100 + 2 \cdot 1000$.

This representation is an example of a *positional system,* in which each place/position has a value, and the symbol in that position tells us how many of that value the number has. Some ancient cultures used non-decimal positional systems, some of which survive to the present day: for example, the Sumarians and Babylonians used a base 60 system—and, even today, 60 seconds make a minute, and 60 minutes make an hour.

In general, to represent a number n in base $b \geq 2$, we write a sequence of elements of $\{0, 1, \ldots, b-1\}$—say $[d_k d_{k-1} \cdots d_2 d_1 d_0]_b$. (We'll write the base explicitly as a subscript, for clarity.) Moving from right to left, the ith position is "worth" b^i, so this number's value is $\sum_{i=0}^{k} b^i d_i$. For example,

$$[1234]_5 = 4 \cdot 5^0 + 3 \cdot 5^1 + 2 \cdot 5^2 + 1 \cdot 5^3 = 4 + 15 + 50 + 125 = 194$$
$$[1234]_8 = 4 \cdot 8^0 + 3 \cdot 8^1 + 2 \cdot 8^2 + 1 \cdot 8^3 = 4 + 24 + 128 + 512 = 668.$$

We can use modular arithmetic to quickly convert an integer to an arbitrary base b. (For simplicity, it's easiest to think about the input n as being written in base 10, but it's not harder to convert from an arbitrary base instead.) To start, notice that

$$\left(\sum_{i=0}^{k} b^i d_i\right) \bmod b = d_0.$$

(The value $b^i d_i$ is divisible by b for any $i \geq 1$.) Thus, to represent n in base b, we have no choice: we must have $d_0 := n \bmod b$. Similarly, $\left(\sum_{i=0}^{k} b^i d_i\right) \bmod b^2 = bd_1 + d_0$; thus we must choose $d_1 := \frac{n-d_0}{b} \bmod b$. (Note that $n - d_0$ must be divisible by b, because of our choice of d_0.) An algorithm following this strategy is shown in Figure 7.4. (We could also have written this algorithm without using division; see Exercise 7.5.)

baseConvert(n, b):

Input: integers n and $b \geq 2$

Output: n, represented in base b

```
1 i := 0
2 while n > 0:
3     d_i := n mod b
4     n := (n − d_i)/b
5     i := i + 1
6 return [d_i d_{i−1} ··· d_1 d_0]_b
```

$d_i := n \bmod 2$

n	i	d_i
145	0	1
72	1	0
36	2	0
18	3	0
9	4	1
4	5	0
2	6	0
1	7	1
0	8	—

Figure 7.4 A base-conversion algorithm, and an example—here, converting 145 (in base 10) to binary (base 2). For each iteration of **baseConvert**$(145, 2)$, the values of n, i, and d_i are shown; thus 145 can be written (reading from the bottom up) as $[10010001]_2$.

We can use the base conversion algorithm in Figure 7.4 to convert decimal numbers (base 10) into *binary* (base 2), the internal representation in computers. Or we can convert into *octal* (base 8) or *hexadecimal* (base 16), two other frequently used representations for numbers in programming. But we can also use **baseConvert** for seemingly

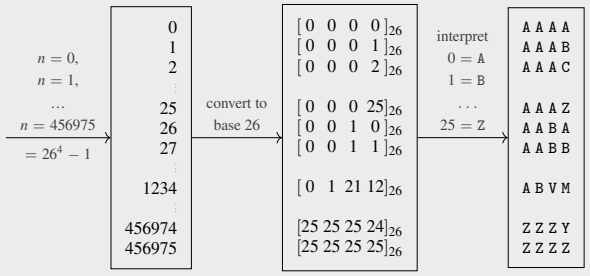


Figure 7.5 Generating all 4-letter strings using **baseConvert**.

unrelated problems. Consider the task of enumerating all 4-letter strings from the alphabet. The "easy" way to write a program to accomplish this task, with four nested loops, is painful to write—and it becomes utterly unwieldy if we needed all 10-letter strings instead. But, instead, let's count from 0 up to $26^4 - 1$—there are 26^4 different 4-letter strings—and convert each number into base 26. We can then translate each number into a sequence of letters, with the ith digit acting as an index into the alphabet that tells us which letter to put in position i. See Figure 7.5.

EXERCISES

7.1 *Using paper and pencil only,* follow the proof of Theorem 7.1 or use the **mod-and-div** algorithm (see Figure 7.6) to compute integers $r \in \{0, 1, \ldots, k-1\}$ and d such that $kd + r = n$, for $k = 17$ and $n = 202$.

7.2 Repeat Exercise 7.1 for $k = 99$ and $n = 2017$.

7.3 Repeat Exercise 7.1 for $k = 99$ and $n = -2017$.

7.4 Let $k \geq 1$ and n be integers. In proving Theorem 7.1, we showed that there exist integers r and d such that $0 \leq r < k$ and $kd + r = n$. We stated but did not prove that r and d are unique. Prove that they are. In other words, for $r, r', d, d' \in \mathbb{Z}$, prove that if $0 \leq r < k$ and $0 \leq r' < k$ and $n = dk + r = d'k + r'$, then $d' = d$ and $r' = r$.

7.5 The algorithm **baseConvert** on p. 7-16, which performs base conversion, is written using division. Modify the algorithm so that it uses only addition, subtraction, mod, multiplication, and comparison.

A repdigit$_b$ *is a number n that, when represented in base b (see p. 7-16), consists of the same symbol written over and over, repeated at least twice. For example,* 666 *is a repdigit*$_{10}$*: when you write* $[666]_{10}$*, it's the same digit ("*6*") repeated (in this case, three times). One way of understanding that* 666 *is a repdigit*$_{10}$ *is that* $666 = 6 + 60 + 600 = 6 \cdot 10^0 + 6 \cdot 10^1 + 6 \cdot 10^2$*. We can write* $[40]_{10}$ *as* $[130]_5$ *because* $40 = 0 + 3 \cdot 5 + 1 \cdot 5^2$*, or as* $[101000]_2$ *because* $40 = 1 \cdot 2^3 + 1 \cdot 2^5$*. So* 40 *is not a repdigit*$_{10}$*, repdigit*$_5$*, or repdigit*$_2$*. But* 40 *is a repdigit*$_3$*, because* $40 = [1111]_3$.

7.6 Prove that every number $n \geq 3$ is a repdigit$_b$ for some base $b \geq 2$, where $n = [11 \cdots 1]_b$.

7.7 Prove that every even number $n > 6$ is a repdigit$_b$ for some base $b \geq 2$, where $n = [22 \cdots 2]_b$.

7.8 Prove that *no* odd number n is a repdigit$_b$ of the form $[22 \cdots 2]_b$, for any base b.

7.9 Write $R(n)$ to denote the number of bases b, for $2 \leq b \leq n-1$, such that n is a repdigit$_b$. Conjecture a condition on n such that $R(n) = 1$, and prove your conjecture.

7.10 Recall the **mod-and-div**(n, m) algorithm, reproduced in Figure 7.6, that computes n mod k and $\lfloor \frac{n}{k} \rfloor$ by repeatedly subtracting k from n until the result is less than k. As written, the **mod-and-div** algorithm fails when given a negative value of n. Follow Case II of Theorem 7.1's proof to extend the algorithm for $n < 0$ too.

7.11 The **mod-and-div** algorithm is slow—this algorithm computes an integer d such that $nd \leq m < n(d+1)$ by performing *linear search* for d. A faster version of this algorithm, called **mod-and-div-faster**, finds d using *binary search* instead; again, see Figure 7.6. The code for **mod-and-div-faster** as written uses division, by averaging *lo* and *hi*. Modify the algorithm so that it uses only addition, subtraction, multiplication, and comparison.

7.12 The code for **mod-and-div-faster** as written uses $hi := n + 1$ as the initial upper bound. Why is this assignment an acceptable for the correctness of the algorithm? Explain briefly.

mod-and-div(n, k):
Input: integers $n \geq 0$ and $k \geq 1$
Output: n mod k and $\lfloor \frac{n}{k} \rfloor$
1 $r := n$; $d := 0$
2 **while** $r \geq k$:
3 $\quad r := r - k$; $d := d + 1$
4 **return** r, d

mod-and-div-faster(n, k):
Input: integers $n \geq 0$ and $k \geq 1$
Output: n mod k and $\lfloor \frac{n}{k} \rfloor$
1 $lo := 0$; $hi := n + 1$
2 **while** $lo < hi - 1$:
3 $\quad mid := \lfloor (lo + hi)/2 \rfloor$
4 $\quad$ **if** $mid \cdot k \leq n$ **then**
5 $\quad\quad lo := mid$
6 $\quad$ **else**
7 $\quad\quad hi := mid$
8 **return** $(n - k \cdot lo), lo$

mod-exp(b, e, n):
Input: integers $n \geq 1$, b, and $e \geq 0$
Output: b^e mod n
1 **if** $e = 0$ **then**
2 $\quad$ **return** 1
3 **else if** e is even **then**
4 $\quad result :=$ **mod-exp**$(b, \frac{e}{2}, n)$
5 $\quad$ **return** $(result \cdot result)$ mod n
6 **else**
7 $\quad result :=$ **mod-exp**$(b, e - 1, n)$
8 $\quad$ **return** $(b \cdot result)$ mod n

Figure 7.6 Two algorithms to compute n mod k and $\lfloor \frac{n}{k} \rfloor$ (one reminder and one faster version), and modular exponentiation via repeated squaring.

7.13 Describe an algorithm that finds a better upper bound *hi*, by repeatedly doubling *hi* until it's large enough.

7.14 Let k be arbitrary. Describe an input n for which the doubling search from Exercise 7.13 yields a significant improvement on the running time of the algorithm for inputs k and n.

7.15 *(programming required.)* Implement, in a programming language of your choice, all three of these algorithms (**mod-and-div**, **mod-and-div-faster**, and the doubling-search modified version of **mod-and-div-faster** from Exercise 7.13) to compute $n \bmod k$ and $\lfloor \frac{n}{k} \rfloor$.

7.16 Run the three algorithms from Exercise 7.15 to compute the following values: $2^{32} \bmod 202$, $2^{32} \bmod 2020$, and $2^{32} \bmod 3^{15}$. How do their speeds compare?

7.17 Prove Theorem 7.4.2: for integers $k > 0$, a, and b, we have $a + b \bmod k = [(a \bmod k) + (b \bmod k)] \bmod k$. Begin your proof as follows: *We can write $a = ck + r$ and $b = dk + t$ for $r, t \in \{0, \ldots, k-1\}$ (as guaranteed by Theorem 7.1).* Then use **mod-and-div** and Lemma 7.3.

7.18 Prove that $a \bmod b = (a \bmod bc) \bmod b$ for all positive integers a, b, and c.

7.19 Prove Theorem 7.7.1: $a \mid 0$ for any positive integer a.

7.20 Prove Theorem 7.7.2: $1 \mid a$ for any positive integer a.

7.21 Prove Theorem 7.7.6: for all positive integers a, b, and c, if $a \mid b$ and $a \mid c$, then $a \mid (b + c)$.

7.22 Prove Theorem 7.7.7: for all positive integers a, b, and c, if $a \mid c$, then $a \mid bc$.

7.23 Consider the "repeated squaring" algorithm **mod-exp** for modular exponentiation shown in Figure 7.6. Observe that this algorithm computes $b^e \bmod n$ with a recursion tree of depth $\Theta(\log e)$. Use this algorithm to compute $3^{80} \bmod 5$ *without using a calculator*. (You should never have to keep track of a number larger than 5 except for the exponent itself when you're doing these calculations!)

7.24 Write down a recurrence relation representing the number of multiplications done by **mod-exp**(b, e, n). Prove, using this recurrence, that the number of multiplications done is between $\log e$ and $2 \log e$.

7.25 *(programming required.)* Implement **mod-exp** in a programming language of your choice. Also implement a version of **mod-exp** that computes b^e and then, after that computation is complete, takes the result $\bmod n$. Compare the speeds of these two algorithms in computing $3^k \bmod 5$, for $k = 80$, $k = 800$, $k = 8000$, …, $k = 8{,}000{,}000$. Explain.

There's a category of numerical tricks often called "divisibility rules" that you may have seen—quick ways of testing whether a given number is evenly divisible by some small k. The test for whether an integer n is divisible by 3 *is this:* add up the *digits* of n; n is divisible by 3 if and only if this sum is divisible by 3. *For example,* 6,007,023 *is divisible by* 3 *because* $6+0+0+7+0+2+3 = 18$, *and* $3 \mid 18$. *(Indeed* $3 \cdot 2{,}002{,}341 = 6{,}007{,}023$*.) This test relies on the following claim: for any sequence* $\langle x_0, x_1, \ldots, x_{n-1} \rangle \in \{0, 1, \ldots, 9\}^n$*, we have*

$$\left[\sum_{i=0}^{n-1} 10^i x_i \right] \bmod 3 \;=\; \left[\sum_{i=0}^{n-1} x_i \right] \bmod 3.$$

(For example, 6,007,023 *is represented as* $x_0 = 3$, $x_1 = 2$, $x_2 = 0$, $x_3 = 7$, $x_4 = 0$, $x_5 = 0$, *and* $x_6 = 6$*.)*

7.26 Prove that the test for divisibility by 3 is correct. First prove that $10^i \bmod 3 = 1$ for any integer $i \geq 0$; then prove the stated claim. Your proof should make heavy use of the properties in Theorem 7.4.

7.27 The divisibility test for 9 is to add up the digits of the given number, and test whether that sum is divisible by 9. State and prove the condition that ensures that this test is correct.

7.28 *Using paper and pencil only,* use the Euclidean algorithm to compute the GCDs of $n = 111$ and $m = 202$.

7.29 Do the same for $n = 333$ and $m = 2017$.

7.30 Do the same for $n = 156$ and $m = 360$.

7.31 *(programming required.)* Implement the Euclidean algorithm in a language of your choice.

7.32 *(programming required.)* Early in Section 7.2.4, we discussed a brute-force algorithm to compute GCD(n,m): try every $d \in \{1,2,\ldots,\min(n,m)\}$ and return the largest d such that $d \mid n$ and $d \mid m$. Implement this algorithm, and compare its performance to the Euclidean algorithm as follows: for both algorithms, find the largest n for which you can compute GCD$(n,n-1)$ in less than 1 second on your computer.

7.33 Let's analyze the running time of the Euclidean algorithm for GCDs, to prove Theorem 7.13. Let n and m be arbitrary positive integers with $n \leq m$. Prove that $m \bmod n \leq \frac{m}{2}$. *(Hint: what happens if $n \leq \frac{m}{2}$? What happens if $\frac{m}{2} < n \leq m$?)*

7.34 Using Exercise 7.33, prove that the Euclidean algorithm terminates within $O(\log n + \log m)$ recursive calls. (Actually it's possible to prove a bound that's tighter by a constant factor, but this result is good enough for asymptotic work.)

Now let's show that, in fact, the Euclidean algorithm generates a recursion tree of depth $\Omega(\log n + \log m)$ in the worst case—specifically, when **Euclid**(f_n, f_{n+1}) *is run on consecutive Fibonacci numbers f_n, f_{n+1}.*

7.35 Show that, for all $n \geq 3$, we have $f_n \bmod f_{n-1} = f_{n-2}$, where f_i is the ith Fibonacci number. (Recall from Definition 6.21 that $f_1 = 1, f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 3$.)

7.36 Prove that, for all $n \geq 3$, **Euclid**(f_{n-1}, f_n) generates a recursion tree of depth $n-2$.

7.37 Using Exercise 7.36 and the fact that $f_n \leq 2^n$ (Exercise 6.95), argue that the running time of the Euclidean algorithm is $\Omega(\log n + \log m)$ in the worst case.

7.3 Primality and Relative Primality

> Why is it that we entertain the belief that for every purpose odd numbers are the most effectual?
>
> ---
>
> Pliny the Elder (23–79)
> Book XXVIII ("Remedies Derived From Living Creatures")
> *The Natural History* (c. 79)

Now that we've reviewed divisibility (and the related notions of factors, divisors, and multiples) in Section 7.2, we'll continue with a brief review of another concept from Chapter 2: the definition of *prime numbers*. We'll then introduce the related notion of *relatively prime* integers—pairs of numbers that share no common divisors aside from 1—and a few applications and extensions of both definitions.

7.3.1 Primality (A Reminder) and Relative Primality (An Introduction)

We begin with a reminder of the definitions from Chapter 2:

Definition 7.14: Primes and composites [reprise].
An integer $p \geq 2$ is called *prime* if the only positive integers that evenly divide it are 1 and p itself. An integer $n \geq 2$ that is not prime is called *composite*. (Note that 1 is neither prime nor composite.)

For example, the integers 2, 3, 5, and 7 are all prime, but 4 (which is divisible by 2) and 6 (which is divisible by 2 and 3) are composite. It's also worth recalling two results that we saw in Chapter 4:

There are infinitely many prime numbers. Example 4.31 gave a proof by contradiction to show that there is no largest prime. (That result is attributed to Euclid—the same Euclid whose algorithm we encountered in Section 7.2.)

The smallest divisor of a composite number isn't too big. Theorem 4.32 showed that any composite number $n \geq 2$ is divisible by some factor $d \leq \sqrt{n}$. (That is, $n \geq 2$ is prime if and only if $d \nmid n$ for every $d \in \{2, 3, \ldots, \sqrt{n}\}$.)

We used the latter result to give an algorithm for the *primality testing problem* that performs $\sqrt{n}$ divisibility tests. (The primality testing problem: we are given an integer $n \geq 2$, and we have to figure out whether n is prime or composite.) This algorithm simply exhaustively tests whether n is divisible by any of the candidate divisors between 2 and $\sqrt{n}$.

> **Taking it further:** The faster divisibility algorithm that you developed in Exercises 7.11–7.16 will allow us to test primality in $\Theta(\sqrt{n} \cdot \log^k n)$ steps, for some constant k: faster than the naïve algorithm, but still not efficient. There *are* faster algorithms for primality testing that require only polylogarithmically many operations—that is, $O(\log^k n)$, for some fixed k—to test whether n is prime. See, for example, p. 7-53 for a discussion of a *randomized* algorithm that efficiently tests for primality, which requires only $O(\log^k n)$ steps to test whether n is prime, although it does have a small (provably small!) probability of making a mistake. There

are also deterministic algorithms to solve this problem in polylogarithmic time, though they're substantially more complicated than this randomized algorithm.

Prime numbers turn out to be useful in all sorts of settings, and it will sometimes turn out to be valuable to compute a large collection of primes all at once. Of course, we can always generate more than one prime number by using a primality-testing algorithm (like the one we just suggested) more than once, until enough numbers have passed the test. But some of the work that we do in figuring out whether n is prime actually turns out to be helpful in figuring out whether $n' > n$ is prime. An algorithm called the *Sieve of Eratosthenes*, which computes a list of *all* prime numbers up to a given integer, exploits this redundancy to save some computation. (The Sieve of Eratosthenes is named after Eratosthenes, a Greek scholar who lived in the 3rd century BCE. Eratosthenes is also credited as the first person to calculate the size of the earth—people were a lot less specialized back then.)

The Sieve generates its list of prime numbers by successively eliminating ("sieving") all multiples of each discovered prime: for example, once we know that 2 is prime and that 4 is a multiple of 2, we will never have to test whether $4 \mid n$ in determining whether n is prime. (If n isn't prime because $4 \mid n$, then n is also divisible by 2—that is, 4 is never the smallest integer greater than 1 that evenly divides n, so we never have to bother testing 4 as a candidate divisor.) See Exercises 7.38–7.42 and Figure 7.17.

Taking it further: The Sieve of Eratosthenes is one of the earliest known algorithms, dating back to about 200 BCE. (The date isn't clear, in part because none of Eratosthenes's work survived; the algorithm was reported, and attributed to Eratosthenes, by Nicomachus about 300 years later.) The Euclidean algorithm for greatest common divisors from Section 7.2, which dates from c. 300 BCE, is one of the few older algorithms that are known. For more, see [72].

The distribution of the primes

For a positive integer n, let $primes(n)$ denote the number of prime numbers less than or equal to n. Thus, for example, we have

$$\begin{aligned} 0 &= primes(1) \\ 1 &= primes(2) \\ 2 &= primes(3) = primes(4) \\ 3 &= primes(5) = primes(6), \text{ and} \\ 4 &= primes(7) = primes(8) = primes(9) = primes(10). \end{aligned}$$

Or, to state it recursively: we have $primes(1) = 0$, and, for $n \geq 2$, we have

$$primes(n) = \begin{cases} primes(n-1) & \text{if } n \text{ is composite} \\ 1 + primes(n-1) & \text{if } n \text{ is prime.} \end{cases}$$

Figure 7.7 displays the value of $primes(n)$ for moderately small n. An additional fact that we'll state without proof is the *Prime Number Theorem*—also illustrated in Figure 7.7—which describes the behavior of $primes(n)$ for large n:

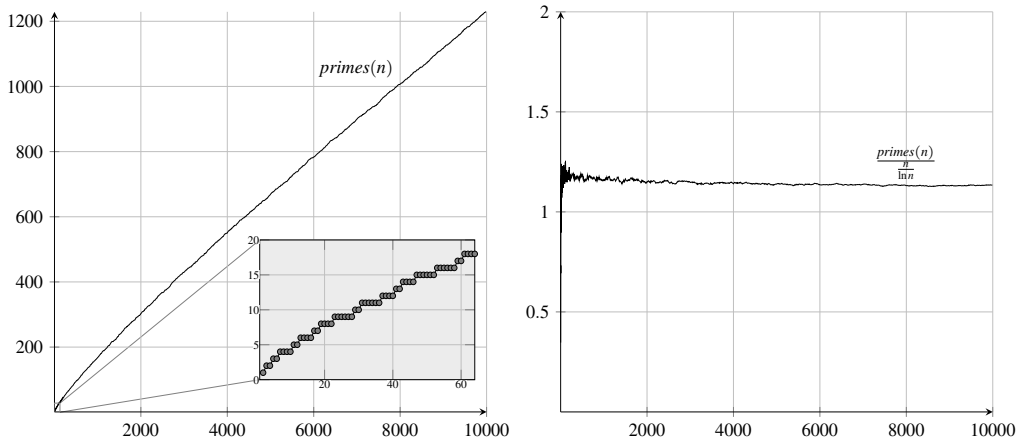


Figure 7.7 The distribution of primes. The Prime Number Theorem states that the ratio $primes(n)/\frac{n}{\ln n}$, on the right, slowly converges to 1.

Theorem 7.15: Prime Number Theorem.
Let $primes(n)$ denote the number of primes less than or equal to n. As n gets large, the ratio between $primes(n)$ and $\frac{n}{\ln n}$ approaches 1.

Formal proofs of the Prime Number Theorem are complicated beasts—far more complicated that we'll want to deal with here!—but even an intuitive understanding of the theorem is useful. Informally, this theorem says that, given an integer n, approximately a $\frac{1}{\ln n}$ fraction of the numbers "close to" n are prime. (See Exercise 7.45.)

Example 7.8: Using the Prime Number Theorem.
Using the estimate $primes(n) \approx \frac{n}{\ln n}$, calculate (approximately) how many 10-digit integers are prime.

Solution. By definition, there are exactly $primes(999{,}999{,}999)$ primes with 9 or fewer digits, and there are exactly $primes(9{,}999{,}999{,}999)$ primes with 10 or fewer digits. Thus the number of 10-digit primes is

$$\begin{aligned} primes(9{,}999{,}999{,}999) - primes(999{,}999{,}999) &\approx \frac{9{,}999{,}999{,}999}{\ln 9{,}999{,}999{,}999} - \frac{999{,}999{,}999}{\ln 999{,}999{,}999} \\ &\approx 434{,}294{,}499 - 48{,}254{,}956 \\ &= 386{,}039{,}543. \end{aligned}$$

Thus, roughly 386 million of the 9 billion 10-digit numbers (about 4.3%) are prime. (Exercise 7.46 asks you to consider how far off this estimate is.)

Problem-solving tip: Back-of-the-envelope calculations are often great as plausibility checks: although the Prime Number Theorem doesn't state a formal bound on how different $primes(n)$ and $\frac{n}{\ln n}$ are, you can see whether a solution to a problem "smells right" with an approximation like this one.

The density of the primes is potentially interesting for its own sake, but there's also a practical reason that we'll care about the Prime Number Theorem. In the RSA cryptosystem (see Section 7.5), one of the first steps of the protocol involves choosing two large prime numbers p and q. The bigger p and q are, the more secure the encryption, so we would want p and q to be pretty big—say, both approximately 2^{2048}. The Prime Number Theorem tells us that, roughly, one out of every $\ln 2^{2048} \approx 1420$ integers around 2^{2048} is prime. Thus, we can find a prime in this range by repeatedly choosing a random integer n of the right size and testing n for primality, using some efficient primality testing algorithm. (More about testing algorithms soon.) Approximately one out of every 1420 integers we try will turn out to be prime, so on average we'll only need to try about 2840 values of n before we find primes to use as p and q.

Prime factorization

Recall that any integer can be *factored* into the product of primes. For example, we can write $2001 = 3 \cdot 23 \cdot 29$ and $202 = 2 \cdot 101$ and $507 = 3 \cdot 13 \cdot 13$ and $55057 = 55057$. (All of 2, 3, 13, 23, 29, 101, and 55057 are prime.) The *Fundamental Theorem of Arithmetic* (Theorem 5.11) states that any integer n can be factored into a product of primes—and that, up to reordering, there is a *unique* prime factorization of n. (In other words, any two prime factorizations of an integer n *can* differ in the ordering of the factors—for example, $202 = 101 \cdot 2$ and $202 = 2 \cdot 101$—but they can differ *only* in ordering.) We proved the "there exists" part of the theorem in Example 5.12 using induction; a bit later in this section, we'll prove uniqueness. (The proof uses some properties of prime numbers that are most easily seen using an extension of the Euclidean algorithm that we'll introduce shortly; we'll defer the proof until we've established those properties.)

Relative primality

An integer n is prime if it has no divisors except 1 and n itself. Here we will introduce a related concept for *pairs* of integers—two numbers that do not *share* any divisors except 1:

Definition 7.16: Relative primality.
Two positive integers n and m are called *relatively prime* if $\text{GCD}(n, m) = 1$—that is, if 1 is the only positive integer that evenly divides both n and m.

Here are a few small examples:

Example 7.9: Some relatively prime integers.
The integers 21 and 25 are relatively prime, as $21 = 3 \cdot 7$ and $25 = 5 \cdot 5$ have no common divisor (other than 1). Similarly, 5 and 6 are relatively prime, as are 17 and 35. (But 12 and 21 are not relatively prime, because they're both divisible by 3.)

You'll prove a number of useful facts about relatively prime numbers in the exercises—for example, a prime number p and any integer n are relatively prime unless $p \mid n$; and, more generally, two numbers are relatively prime if and only if their prime factorizations do not share any factors. (The "usefulness" of these facts will show up when we look at cryptographic systems.)

Taking it further: Let $f(x)$ be a polynomial. Polynomials have some useful properties, and here's one of the special characteristics of prime numbers: some of these useful properties of $f(x)$ continue to hold *if we take the result of evaluating the polynomial* mod p *for some prime number* p. In particular, if $f(x)$ is a polynomial of degree k, then either $f(a) \equiv_p 0$ *for every* $a \in \{0, 1, \ldots, p-1\}$ or there are at most k values $a \in \{0, 1, \ldots, p-1\}$ such that $f(a) \equiv_p 0$. (We saw this property in Section 2.5.3 when we didn't take the result modulo the prime p.) As a consequence, if we have two polynomials $f(x)$ and $g(x)$ of degree k, then if f and g are not equivalent modulo p, then there are at most k values of $a \in \{0, 1, \ldots, p-1\}$ for which $f(a) \equiv_p g(a)$.

We can use the fact that polynomials of degree k "behave" in the same way modulo p (with respect to the number of roots, and the number of places that two polynomials agree) to give efficient solutions to two problems: *secret sharing,* in which n people wish to "distribute" shares of a secret so that any k of them can reconstruct the secret (but no set of $k-1$ can); and a form of *error-correcting codes,* as we discussed in Section 4.2. The basic idea will be that by using a polynomial $f(x)$ and evaluating $f(x)$ mod p for a prime p, we'll be able to use *small* numbers (less than p) to accomplish everything that we'd be able to accomplish by evaluating $f(x)$ without the modulus. See the discussions of secret sharing on p. 7-36 and of Reed–Solomon codes on p. 7-38.

7.3.2 A Structural Fact and the Extended Euclidean Algorithm

Given an integer $n \geq 2$, quickly determining whether n is prime seems tricky: we've seen some easy algorithms for this problem, but they're pretty slow. And, though there *are* efficient but complicated algorithms for primality testing, we haven't seen (and, really, nobody knows) a genuinely simple algorithm that's also efficient. On the other hand, the analogous question about relative primality—*given integers n and m, are n and m relatively prime?*—is easy. In fact, we already know everything we need to solve this problem efficiently, just from the definition: n and m are relatively prime if and only if their GCD is 1, which occurs if and only if $\textbf{Euclid}(n, m) = 1$. So we can efficiently test whether n and m are relatively prime by testing whether $\textbf{Euclid}(n, m) = 1$.

We will start this section with a structural property about GCDs. (Right now it shouldn't be at all clear what this claim has to with anything in the last paragraph—but stick with it! The connection will come along soon.) Here's the claim:

Lemma 7.17: There are multiples of n and m that add up to $\text{GCD}(n, m)$.
Let n and m be any positive integers, and let $r = \text{GCD}(n, m)$. Then there exist integers x and y such that $xn + ym = r$.

Here are a few examples of the multiples guaranteed by this lemma:

n	m	$x \cdot n$	$+$	$y \cdot m$	$=$	$\text{GCD}(n,m)$		
5	6	$(-1) \cdot 5$	$+$	$1 \cdot 6$	$=$	$-5+6$	$=1$	$=\text{GCD}(5,6)$
17	35	$33 \cdot 17$	$+$	$(-16) \cdot 35$	$=$	$561-560$	$=1$	$=\text{GCD}(17,35)$
12	21	$2 \cdot 12$	$+$	$(-1) \cdot 21$	$=$	$24-21$	$=3$	$=\text{GCD}(12,21)$
48	1024	$(-21) \cdot 48$	$+$	$1 \cdot 1024$	$=$	$-1008+1024$	$=16$	$=\text{GCD}(48,1024)$
16	48	$1 \cdot 16$	$+$	$0 \cdot 48$	$=$	$16+0$	$=16$	$=\text{GCD}(16,48)$

Figure 7.8 Some examples of Lemma 7.17. For each pair of integers n and m, we've chosen integers x and y such that $xn + ym = \text{GCD}(n,m)$.

Example 7.10: Some examples of Lemma 7.17.
For several pairs of integers n and m, values of x and y such that $xn + ym = \text{GCD}(n,m)$ are shown in Figure 7.8. Note that there are multiple options; for the pair $\{17,35\}$ in Figure 7.8, for example, we could have chosen $x = -2$ and $y = 1$ instead of $x = 33$ and $y = -16$, as $-2 \cdot 17 + 1 \cdot 35$ and $33 \cdot 17 + (-16) \cdot 35$ are both equal to $1 = \text{GCD}(17,35)$.

Note that the integers x and y whose existence is guaranteed by Lemma 7.17 are not necessarily positive! (In fact, in Example 7.10 the only time that we didn't have a negative coefficient for one of the numbers was for the pair $\{16,48\}$, where $\text{GCD}(16,48) = 16 = 1 \cdot 16 + 0 \cdot 48$.) Also, as Example 7.10 illustrates, there may be more than one pair of values for x and y that satisfy Lemma 7.17—in fact, you'll show in Exercise 7.58 that there are *always* infinitely many values of $\{x,y\}$ that satisfy the lemma.

Although, if you stare at it long enough, Example 7.10 might give a *tiny* hint about why Lemma 7.17 is true, a proof still seems distant. But, in fact, we'll be able to prove the claim based what looks like a digression: a mild extension to the Euclidean algorithm. For a little bit of a hint as to how, let's look at one more example of the Euclidean algorithm, but interpreting it as a guide to find the integers in Lemma 7.17:

Example 7.11: An example of Lemma 7.17, using the Euclidean algorithm.
Let's find integers x and y such that $91x + 287y = \text{GCD}(91,287)$. By running **Euclid**$(91,287)$, we make the recursive calls **Euclid**$(14,91)$ and **Euclid**$(7,14)$, which returns 7. (See Figure 7.9 for a reminder.) So

$$\text{GCD}(91,287) = 7. \tag{A}$$

Remember from the definition of mod (Definition 7.2) that we can write $m = \lfloor \frac{m}{n} \rfloor \cdot n + (m \bmod n)$ for any integers m and n. Or, by rearranging,

$$m \bmod n = m - \lfloor \tfrac{m}{n} \rfloor \cdot n$$

Specifically, for the first two calls to **Euclid** ($n = 91$ and $m = 287$, and $n = 14$ and $m = 91$), we have

$$14 = 287 \bmod 91 = 287 - \lfloor \tfrac{287}{91} \rfloor \cdot 91 = 287 - 3 \cdot 91 \tag{B}$$

$$7 = \ 91 \bmod 14 = 91 - \lfloor \tfrac{91}{14} \rfloor \cdot 14 \quad = 91 - 6 \cdot 14. \tag{C}$$

Now, assembling all of these facts, we have that

$$\text{GCD}(91, 287) \overset{\text{by (A)}}{=} 7 \overset{\text{by (C)}}{=} 91 - 6 \cdot 14 \overset{\text{by (B)}}{=} 91 - 6 \cdot [287 - 3 \cdot 91] \overset{\text{collecting like terms}}{=} -6 \cdot 287 + 19 \cdot 91.$$

Thus $x = -6$ and $y = 19$ satisfy the requirement that $91x + 287y = \text{GCD}(91, 287)$.

The Extended Euclidean algorithm

The *Extended Euclidean algorithm,* shown in Figure 7.9, follows the outline of Example 7.11, applying these algebraic manipulations recursively. Lemma 7.17 will follow from a proof that this extended version of the Euclidean algorithm actually *computes* three integers x, y, r such that $\text{GCD}(n, m) = r = xn + ym$. Here are two examples:

Example 7.12: Running the Extended Euclidean Algorithm.
Evaluating **extended-Euclid**$(12, 18)$ recursively computes **extended-Euclid**$(6, 12)$, which returns $x = 1$, $y = 0$, and $r = 6$. The else case of the algorithm tells us that our result is $\langle y - \lfloor \frac{m}{n} \rfloor \cdot x, x, r \rangle$ where $m = 18$ and $n = 12$. Plugging these values into the formula, we see that **extended-Euclid**$(12, 18)$ returns $\langle -1, 1, 6 \rangle$—and, indeed, $\text{GCD}(12, 18) = 6$ and $-1 \cdot 12 + 1 \cdot 18 = 6$. (See Figure 7.10a.)

Figure 7.10b shows a slightly more complicated example: **extended-Euclid**$(18, 30)$, whose result is $x = 2$, $y = -1$, and $r = 6$. Again, we have $\text{GCD}(18, 30) = 6$ and $2 \cdot 18 + -1 \cdot 30 = 36 - 30 = 6$, just as required.

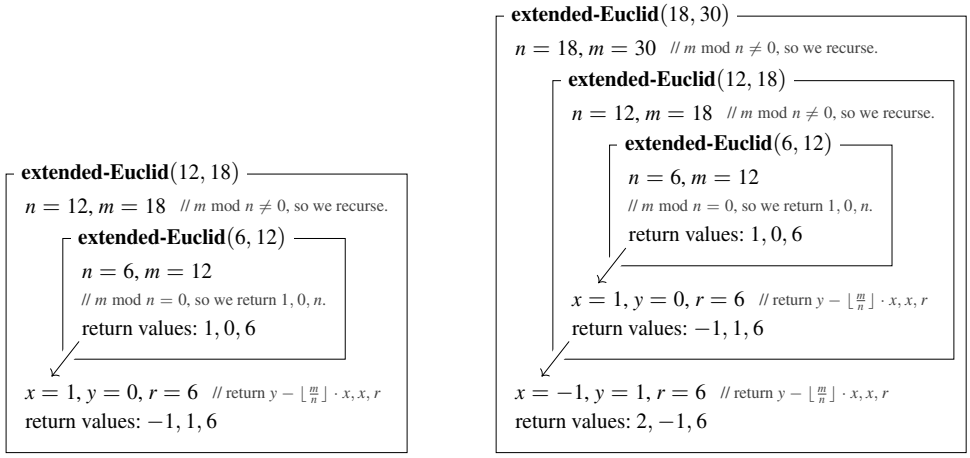
We're now ready to state the correctness of the Extended Euclidean algorithm:

Theorem 7.18: Correctness of the Extended Euclidean Algorithm.
For arbitrary positive integers n and m with $n \leq m$, **extended-Euclid**(n, m) returns three integers x, y, r such that $r = \text{GCD}(n, m) = xn + ym$.

```
extended-Euclid(n, m):
Input: positive integers n and m ≥ n
Output: x, y, r ∈ ℤ where GCD(n, m) = r = xn + ym
1 if m mod n = 0 then
2     return 1, 0, n                    // 1 · n + 0 · m = n = GCD(n, m)
3 else
4     x, y, r := extended-Euclid(m mod n, n)
5     return y − ⌊m/n⌋ · x, x, r
```

```
Euclid(n, m):
Input: positive integers n and m ≥ n
Output: GCD(n, m)
1 if m mod n = 0 then
2     return n
3 else
4     return Euclid(m mod n, n)
```

Figure 7.9 The Extended Euclidean algorithm (and a reminder of the Euclidean algorithm).



(a) Computing **extended-Euclid**$(12, 18)$. (b) Computing **extended-Euclid**$(18, 30)$.

Figure 7.10 Two calls to **extended-Euclid**. (Note that the first recursive call made in (b) computes the value of **extended-Euclid**$(12, 18)$, which is exactly what's shown in (a).)

The proof, which is fairly straightforward by induction, is left to you as Exercise 7.60. And once you've proven this theorem, Lemma 7.17—which merely stated that *there exist* integers x, y, r with $r = \text{GCD}(n, m) = xn + ym$ for any n and m—is immediate.

> *Problem-solving tip:* A nice way, particularly for computer scientists, to prove a theorem of the form "there exists x such that $P(x)$" is to *actually give algorithm that computes such an x!* That's how Theorem 7.18 establishes Lemma 7.17.

Note also that the Extended Euclidean algorithm is an efficient algorithm—you already proved in Exercise 7.34 that the depth of the recursion tree for **Euclid**(n, m) is upper bounded by $O(\log n + \log m)$, and the running time of **extended-Euclid**(n, m) is asymptotically the same as **Euclid**(n, m). (The only quantity that we need to use in **extended-Euclid** that we didn't need in **Euclid** is $\lfloor \frac{m}{n} \rfloor$, but we already had to find m mod n in **Euclid**—so if we use a faster version of **mod-and-div**(n, m) to compute m mod n, then we "for free" also get the value of $\lfloor \frac{m}{n} \rfloor$.)

7.3.3 The Uniqueness of Prime Factorization

Lemma 7.17—that there are multiples of n and m that add up to $\text{GCD}(n, m)$—and the Extended Euclidean algorithm (which computes those coefficients) will turn out to be helpful in proving some facts that are apparently unrelated to greatest common divisors. Here's a claim about divisibility related to prime numbers in that vein, which we'll be able to use to prove that prime factorizations are unique:

Lemma 7.19: When a prime divides a product.
Let p be prime, and let a and b be integers. Then $p \mid ab$ if and only if $p \mid a$ or $p \mid b$.

Proof. We'll proceed by mutual implication.

For the backward direction, assume $p \mid a$. (The case for $p \mid b$ is strictly analogous.) Then $a = kp$ for some integer k, and thus $ab = kpb$, which is obviously divisible by p.

For the forward direction, assume that $p \mid ab$ and suppose that $p \nmid a$. We must show that $p \mid b$. Because p is prime and $p \nmid a$, we know that $\text{GCD}(p, a) = 1$ (see Exercise 7.47), and, in particular, **extended-Euclid**(p, a) returns the GCD 1 and two integers n and m such that $1 = pm + an$. Multiplying both sides by b yields $b = pmb + anb$, and thus

$$\begin{aligned} b \bmod p &= (pmb + anb) \bmod p \\ &= (pmb \bmod p + anb \bmod p) \bmod p && \textit{Theorem 7.4.2} \\ &= (0 + anb \bmod p) \bmod p && \textit{Theorem 7.7.7} \\ &= (0 + 0) \bmod p && \textit{$p \mid ab$ by assumption, and Theorem 7.7.7 again} \\ &= 0. \end{aligned}$$

That is, we've shown that if $p \nmid a$, then $p \mid b$. (And $\neg x \Rightarrow y$ is equivalent to $x \vee y$.) □

We can use Lemma 7.19 to prove that an integer's prime factorization is unique. (We'll prove only the uniqueness part of the Prime Factorization Theorem here; see Example 5.12 for the "there exists a prime factorization" part.)

Theorem 7.20: Prime Factorization Theorem [Reprise].
Let $n \in \mathbb{Z}^{\geq 1}$ be any positive integer. There exist $k \geq 0$ prime numbers $p_1, p_2, \ldots, p_k$ such that $n = \prod_{i=1}^{k} p_i$. Further, up to reordering, the prime numbers $p_1, p_2, \ldots, p_k$ are unique.

Taking it further: Back when we defined prime numbers, we were very careful to specify that 1 *is neither prime nor composite.* You may well have found this insistence to be silly and arbitrary and pedantic—after all, the only positive integers that evenly divide 1 are 1 and, well, 1 itself, so it sure seems like 1 ought to be prime. But there was a good reason that we chose to exclude 1 from the list of primes: *it makes the uniqueness of prime factorization true!* If we'd listed 1 as a prime number, there would be many different ways to prime factor, say, 202: for example, $202 = 2 \cdot 101$ and $202 = 1 \cdot 2 \cdot 101$ and $202 = 1 \cdot 1 \cdot 2 \cdot 101$, and so forth. So we'd have to have restated the theorem about uniqueness of prime factorization ("…is unique up to reordering *and the number of times that we multiply by* 1"), which is a much more cumbersome statement. This theorem *is* the reason that 1 is not defined as a prime number, in this book or in any other mathematical treatment. (There's an embedded problem-solving tip—and life tip—in this discussion: if you have the power to define something, then you genuinely get to choose how to define it. So if you can make a choice in the definition that makes life better, *do it!*)

Proof of Theorem 7.20 (uniqueness). We'll proceed by strong induction on n.

For the base case ($n = 1$), we can write 1 as the product of zero prime numbers—recall that $\prod_{i \in \varnothing} i = 1$—and this representation is unique. (The product of one or more primes is greater than 1, as all primes are at least 2.)

For the inductive case ($n \geq 2$), we assume the inductive hypotheses, namely that any $n' < n$ has a unique prime factorization. We must prove that the prime factorization of n is also unique. We consider two subcases:

Case I: n is prime. Then the statement holds immediately: the only prime factorization is $p_1 = n$. (Suppose that there were a different way of prime factoring n, as $n = \prod_{i=1}^{\ell} q_i$ for prime numbers $\langle q_1, q_2, \ldots, q_\ell \rangle$. We'd have to have $\ell \geq 2$ for this factorization to differ from $p_1 = n$, but then each q_i satisfies $q_i > 1$ and $q_i < n$ and $q_i \mid n$—contradicting what it means for n to be prime.)

Case II: n is composite. Then suppose that $p_1, p_2, \ldots, p_k$ and $q_1, q_2, \ldots, q_\ell$ are two sequences of prime numbers such that $n = \prod_{i=1}^{k} p_i = \prod_{i=1}^{\ell} q_i$. Without loss of generality, assume that both sequences are sorted in increasing order, so that $p_1 \leq p_2 \leq \cdots \leq p_k$ and $q_1 \leq q_2 \leq \cdots \leq q_\ell$. We must prove that these two sequences are actually equal.

Case IIA: $p_1 = q_1$. Define $n' = \frac{n}{p_1} = \frac{n}{q_1} = \prod_{i=2}^{k} p_i = \prod_{i=2}^{\ell} q_i$ as the product of all the other prime numbers (excluding the primes p_1 and $q_1 = p_1$). By the inductive hypothesis, n' has a unique prime factorization, and thus $p_2, p_3, \ldots, p_k$ and $q_2, q_3, \ldots, q_\ell$ are identical.

Case IIB: $p_1 \neq q_1$. Without loss of generality, suppose that $p_1 < q_1$. But we know that $p_1 \mid n$, and therefore $p_1 \mid \prod_{i=1}^{\ell} q_i$. By Lemma 7.19, there exists an i such that $p_1 \mid q_i$. But $2 \leq p_1 < q_1 \leq q_i$. This contradicts the assumption that q_i was prime. □

Taking it further: How difficult is it to factor a number n? Does there exist an efficient algorithm for factoring—that is, one that computes the prime factorization of n in a number of steps that's proportional to $O(\log^k n)$ for some k? *We don't know.* But it is generally believed that the answer is *no,* that factoring large numbers cannot be done efficiently. The (believed) difficulty of factoring is a crucial pillar of widely used cryptographic systems, including the ones that we'll encounter in Section 7.5. There *are* known algorithms that factor large numbers efficiently on so-called *quantum computers* (see p. 10-22)—but nobody knows how to build large quantum computers. And, while there's no known efficient algorithm for factoring large numbers on classical computers, there's also no proof of hardness for this problem. (And most modern cryptographic systems count on the difficulty of the factoring problem—which is only a conjecture!)

7.3.4 The Chinese Remainder Theorem

We'll close this section with another ancient result about modular arithmetic, called the *Chinese Remainder Theorem,* from around 1750 years ago. (The name of the Chinese Remainder Theorem comes from its early discovery by the Chinese mathematician Sun Tzu, who lived around the 5th century. This Sun Tzu, to be clear, is a different Sun Tzu than the one who wrote *The Art of War* about 800 years prior.)

Here's the basic idea. If n is some nonnegative integer, then knowing that, say, when n is divided by 7 its remainder is 4 gives you a small clue about n's value: one seventh of integers have the right value mod 7. Knowing n mod 2 and n mod 13 gives you more clues. The Chinese Remainder Theorem says that knowing n mod k for enough values of k will (almost) let you figure out the value of n exactly—at least, if those values of k are all relatively prime. Here's a concrete example:

Example 7.13: An example of the Chinese Remainder Theorem.
What nonnegative integers n satisfy the following conditions?

$$n \bmod 2 = 0 \qquad n \bmod 3 = 2 \qquad n \bmod 5 = 1.$$

Solution. Suppose $n \in \{0, 1, \ldots, 29\}$. Then there are only six values for which $n \bmod 5 = 1$, namely $0 + 1 = 1$ and $5 + 1 = 6$ and $10 + 1 = 11$ and $15 + 1 = 16$ and $20 + 1 = 21$ and $25 + 1 = 26$. Of these, the only even values are 6, 16, and 26. And $6 \bmod 3 = 0$, $16 \bmod 3 = 1$, and $26 \bmod 3 = 2$. Thus $n = 26$.

Notice that, for any integer k, we have $k \equiv_b k + 30$ for all three moduli $b \in \{2, 3, 5\}$. Therefore any $n \equiv_{30} 26$ will satisfy the given conditions.

The basic point of Example 7.13 is that every value of $n \in \{0, \ldots, 29\}$ has a unique "profile" of remainders mod 2, 3, and 5. (See Figure 7.11.) Crucially, every one of the 30 possible profiles of remainders occurs in Figure 7.11, and no profile appears more than once. (The fact that there are exactly 30 possible profiles follows from the Product Rule for counting; see Section 9.2.1.)

The Chinese Remainder Theorem states the general property that's illustrated in these particular tables: each "remainder profile" occurs once and only once. Here is a formal statement of the theorem. We refer to a constraint of the form $x \bmod n = a$ as a *congruence,* following Definition 7.5. We also write $\mathbb{Z}_k$ to denote the set $\{0, 1, \ldots, k-1\}$.

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
$n \bmod 2 =$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$n \bmod 3 =$	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
$n \bmod 5 =$	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

$n =$	0	6	12	18	24	10	16	22	28	4	20	26	2	8	14	15	21	27	3	9	25	1	7	13	19	5	11	17	23	29
$n \bmod 2 =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$n \bmod 3 =$	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2
$n \bmod 5 =$	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

Figure 7.11 The remainders of all $n \in \{0, 1, \ldots, 29\}$, modulo 2, 3, and 5—sorted by n (above) and by the remainders (below).

Input: relatively prime $n, m \in \mathbb{Z}$; $a \in \mathbb{Z}_n$; $b \in \mathbb{Z}_m$.
Output: x such that $x \bmod m = a$ and $x \bmod n = b$.
1 $c, d, r :=$ **extended-Euclid**(n, m)
2 **return** $x := (adm + bcn) \bmod nm$

Ensure that $m \geq n$ by swapping n and m if necessary. Also, the value r returned by **extended-Euclid** in Line 1 *must* be 1, because n and m are assumed to be relatively prime.

Figure 7.12 An algorithm for the Chinese Remainder Theorem.

Theorem 7.21: Chinese Remainder Theorem: two congruences.
Let n and m be any two relatively prime integers. For any $a \in \mathbb{Z}_n$ and $b \in \mathbb{Z}_m$, there exists one and only one integer $x \in \mathbb{Z}_{nm}$ such that $x \bmod n = a$ and $x \bmod m = b$.

Proof. To show that there exists an integer x satisfying $x \bmod n = a$ and $x \bmod m = b$, we'll give a proof by construction—specifically, we'll *compute* the value of x given the values of $\{a, b, n, m\}$. The two-line algorithm is shown in Figure 7.12. We must argue that $x \bmod n = a$ and $x \bmod m = b$. Note that $\text{GCD}(n, m) = 1$ because n and m are relatively prime by assumption. Thus, by the correctness of the Extended Euclidean algorithm, we have

$$cn + dm = 1. \tag{1}$$

Multiplying both sides of (1) by a, we know that

$$acn + adm = a. \tag{2}$$

Recall that we defined $x := (adm + bcn) \bmod nm$. Let's now show that $x \bmod n = a$:

$$\begin{aligned}
x \bmod n &= (adm + bcn) \bmod nm \bmod n && \textit{definition of } x \\
&= (adm + bcn) \bmod n && \textit{Exercise 7.18} \\
&= (adm + 0) \bmod n && bcn \bmod n = 0 \textit{ because } n \mid bcn \\
&= (adm + acn) \bmod n && acn \bmod n = 0 \textit{ because } n \mid acn \textit{ too!} \\
&= a \bmod n && \textit{(2)} \\
&= a. && a \in \{0, 1, \ldots, n-1\} \textit{ by assumption, so } a \bmod n = a
\end{aligned}$$

We can argue that $x = adm + bcn \equiv_m bdm + bcn \equiv_m b$ completely analogously, where the last equivalence follows by multiplying both sides of (1) by b instead. Thus we've established that *there exists* an $x \in \mathbb{Z}_{nm}$ with $x \bmod n = a$ and $x \bmod m = b$ (because we *computed* such an x).

To prove that there is a *unique* such x, suppose that $x \bmod n = x' \bmod n$ and $x \bmod m = x' \bmod m$ for two integers $x, x' \in \mathbb{Z}_{nm}$. We will prove that $x = x'$. Because $x \bmod n = x' \bmod n$, we know that $(x - x') \bmod n = 0$, or, in other words, that $n \mid (x - x')$. By similar reasoning, we know that $m \mid (x - x')$. By Exercise 7.70 and the fact that n and m are relatively prime, then, we know that $nm \mid (x - x')$. And because both $x, x' \in \mathbb{Z}_{nm}$, we've therefore shown that $x = x'$. □

Some examples

Here are two concrete examples of using the Chinese Remainder Theorem (and, specifically, of using the algorithm from Figure 7.12):

Example 7.14: The Chinese Remainder Theorem, in action.
Let's find the integer $x \in \mathbb{Z}_{30}$ that satisfies $x \bmod 5 = 4$ and $x \bmod 6 = 5$. Note that 5 and 6 are relatively prime, and **extended-Euclid**$(5,6)$ returns $\langle -1, 1, 1\rangle$. (And indeed $5 \cdot -1 + 6 \cdot 1 = 1 = \text{GCD}(5,6)$.) Thus we compute x from the values of $\langle n, m, a, b, c, d\rangle = \langle 5, 6, 4, 5, -1, 1\rangle$ as

$$x := (\overset{4}{\searrow}adm \overset{1}{\downarrow} + \overset{6}{\swarrow} \overset{5}{\searrow}bcn \overset{-1}{\downarrow}) \overset{5}{\swarrow} \bmod 30 = (24 - 25) \bmod 30 = -1 \bmod 30 = 29.$$

And, indeed, $29 \bmod 5 = 4$ and $29 \bmod 6 = 5$.

Example 7.15: A second example of the Chinese Remainder Theorem.
We are told that $x \bmod 7 = 1$ and $x \bmod 9 = 5$. What is the value of x?

Solution. Running **extended-Euclid**$(7,9)$ yields $\langle 4, -3, 1\rangle$. The algorithm in Figure 7.12 computes $x := adm + bcn \bmod nm$, where $n = 7$ and $m = 9$ are the given moduli; $a = 1$ and $b = 5$ are the given remainders; and $c = 4$ and $d = -3$ are the computed multipliers from **extended-Euclid**. Thus

$$x := (1 \cdot -3 \cdot 9) + (5 \cdot 4 \cdot 7) \bmod 7 \cdot 9 = 113 \bmod 63 = 50.$$

Indeed, $50 \bmod 7 = 1$ and $50 \bmod 9 = 5$. Thus $x \equiv_{63} 50$.

Generalizing to k congruences

We've now shown the Chinese Remainder Theorem for two congruences, but Example 7.13 had *three* constraints ($x \bmod 2$, $x \bmod 3$, and $x \bmod 5$). In fact, the generalization of the Chinese Remainder Theorem to k congruences, for any $k \geq 1$, is also true—again, as long as the moduli are *pairwise relatively prime* (that is, *any* two of the moduli share no common divisors).

We can prove this generalization fairly directly, using induction and the two-congruence case. The basic idea will be to repeatedly use Theorem 7.21 to combine a pair of congruences into a single congruence, until there are no pairs left to combine. Here's a concrete example:

Example 7.16: The Chinese Remainder Theorem, with 3 congruences.
Let's describe the values of x that satisfy the congruences

$$x \bmod 2 = 1 \qquad x \bmod 3 = 2 \qquad x \bmod 5 = 4. \tag{$*$}$$

To do so, we first identify values of y that satisfy the first two congruences, ignoring the third. Note that 2 and 3 are relatively prime, and **extended-Euclid**$(2,3) = \langle -1, 1, 1\rangle$. Thus, $y \bmod 2 = 1$ and $y \bmod 3 = 2$ if and only if

$$y \bmod (2 \cdot 3) = (1 \cdot 1 \cdot 3 + 2 \cdot -1 \cdot 2) \bmod (2 \cdot 3) = 5.$$

In other words, $y \in \mathbb{Z}_6$ satisfies the congruences $y \bmod 2 = 1$ and $y \bmod 3 = 2$ *if and only if* y satisfies the single congruence $y \bmod 6 = 5$. Thus the values of x that satisfy $(*)$ are precisely the values of x that satisfy

$$x \bmod 6 = 5 \qquad\qquad x \bmod 5 = 4. \qquad (\dagger)$$

In Example 7.14, we showed that values of x that satisfy $(\dagger)$ are precisely those with $x \bmod 30 = 29$.

Now, using the idea from this example, we'll prove the general version of the theorem:

Theorem 7.22: Chinese Remainder Theorem: General version.
Let $n_1, n_2, \ldots, n_k$ be a collection of integers that are pairwise relatively prime, for some $k \geq 1$. Let $N = \prod_{i=1}^{k} n_i$. Then, for any $\langle a_1, \ldots, a_k \rangle$ with each $a_i \in \mathbb{Z}_{n_i}$, there exists one and only one integer $x \in \mathbb{Z}_N$ such that $x \bmod n_i = a_i$ for all $1 \leq i \leq k$.

Proof. We proceed by induction on k. For the base case ($k = 1$), there's only one constraint, namely $x \bmod n_1 = a_1$. And $x = a_1$ is the only element of $\mathbb{Z}_N = \mathbb{Z}_{n_1}$ that satisfies this congruence.

For the inductive case ($k \geq 2$), we assume the inductive hypothesis, namely that there exists a unique $x \in \mathbb{Z}_M$ satisfying any set of $k-1$ congruences whose moduli have product M. To make use of this assumption, we will convert the k given congruences into $k-1$ equivalent congruences, as shown in Figure 7.13. More formally, we must show that there exists one and only one value of $x \in \mathbb{Z}_N$ satisfying Constraint Set #1. Theorem 7.21 and Exercise 7.77 show that precisely the same values of x satisfy Constraint Set #1 and Constraint Set #2, so it suffices to prove that there exists one and only one value of $x \in \mathbb{Z}_N$ satisfying Constraint Set #2. The product of the moduli is the same for both the Constraint Sets: $N = n_1 \cdot n_2 \cdot n_3 \cdots n_k$ for #1, and $(n_1 n_2) \cdot n_3 \cdots n_k$ for #2. Furthermore, in Exercise 7.69 you'll prove that $n_1 n_2$ is also relatively prime to every other n_i. Thus we can apply the inductive hypothesis to Constraint Set #2, which establishes that there's a unique $x \in \mathbb{Z}_N$ that satisfies Constraint Set #2—and therefore a unique $x \in \mathbb{Z}_N$ that satisfies Constraint Set #1. □

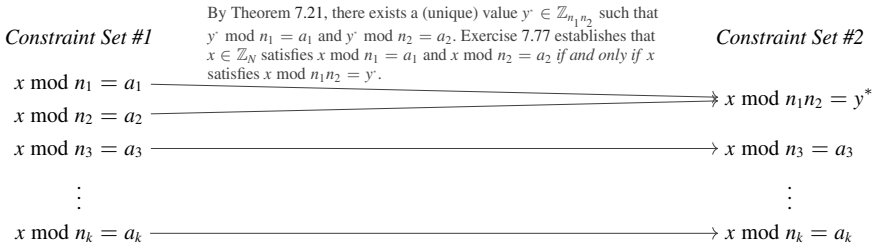


Figure 7.13 A sketch of the proof of Theorem 7.22. We convert the k congruences on the left to the $k-1$ congruences on the right. Exactly the same values of x satisfy the two sets of congruences.

(This proof of the general version of the Chinese Remainder Theorem is an inductive argument, based on the 2-congruence version. But we could also give a version of the proof that directly echoes Theorem 7.21's proof. See Exercise 7.108.)

Taking it further: One interesting implication of the Chinese Remainder Theorem is that we could choose to represent integers efficiently in a very different way from binary representation, instead using something called *modular representation*. In modular representation, we store an integer n as a sequence of values of n mod b, for a set of relatively prime values of b. To be concrete, consider the set $\{11, 13, 15, 17, 19\}$, and let $N = 11 \cdot 13 \cdot 15 \cdot 17 \cdot 19 = 692{,}835$ be their product. The Chinese Remainder Theorem tells us that we can uniquely represent any $n \in \mathbb{Z}_N$ as

$$\langle n \bmod 11, n \bmod 13, n \bmod 15, n \bmod 17, n \bmod 19 \rangle.$$

For example, $2^{17} = \langle 7, 6, 2, 2, 10 \rangle$, and $17 = \langle 6, 4, 2, 0, 17 \rangle$. Perhaps surprisingly, the representation of $2^{17} + 17$ is $\langle 2, 10, 4, 2, 8 \rangle$ and $17 \cdot 2^{17} = \langle 9, 11, 4, 0, 18 \rangle$, which are really nothing more than the result of doing component-wise addition/multiplication (modulo that component's corresponding modulus):

	mod 11	13	15	17	19	
	$\langle$ 7,	6,	2,	2,	10	$\rangle$
$+\langle$	6,	4,	2,	0,	17	$\rangle$
$= \langle$	13,	10,	4,	2,	27	$\rangle$
$\equiv \langle$	2,	10,	4,	2,	8	$\rangle$

and

	mod 11	13	15	17	19	
	$\langle$ 7,	6,	2,	2,	10	$\rangle$
$\cdot\langle$	6,	4,	2,	0,	17	$\rangle$
$= \langle$	42,	24,	4,	0,	170	$\rangle$
$\equiv \langle$	9,	11,	4,	0,	18	$\rangle$.

This representation has some advantages over the normal binary representation: the numbers in each component stay small, and multiplying k pairs of 5-bit numbers is significantly faster than multiplying one pair of $5k$-bit numbers. (Also, the components can be calculated in parallel!) But there are some other operations that are slowed down by this representation. (See Exercises 7.147–7.148.)

Computer Science Connections

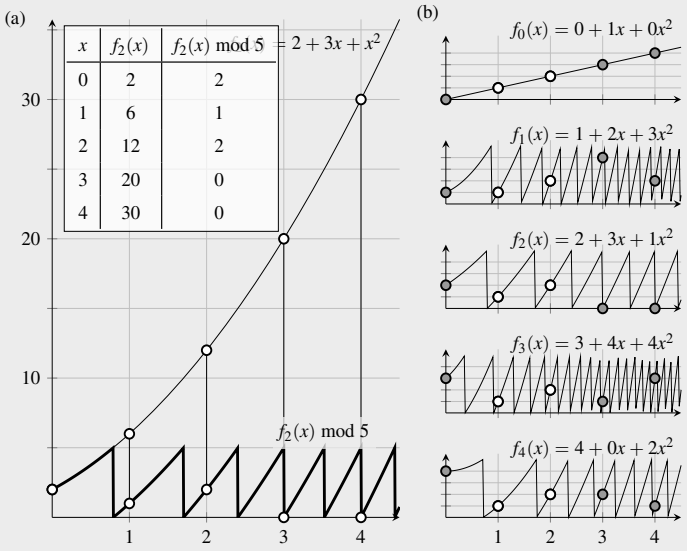
Secret Sharing

Although encryption/decryption is probably the most natural cryptographic problem, there are many other important problems in with related but different requirements—some kind of desired communication that adheres to some type of security specification. Here we'll look at a different cryptographic problem—using a solution due to Adi Shamir (the S of the RSA cryptosystem, which we'll see in Section 7.5) [113]. Imagine a shared resource, collectively owned by some group, that they wish to keep secure—for example, the launch codes for the U.S.'s nuclear weapons. In the post-apocalyptic world in which you're imagining these codes being used, where many top officials are probably dead, we'll need to ensure that any, say, $k = 3$ of the cabinet members (out of the $n = 15$ cabinet positions) can launch the weapons. But you'd also like to guarantee that no single rogue secretary can destroy the world!

In *secret sharing*, we seek a scheme by which we distribute "shares" of the secret $s \in S$ to a group of n people such that two properties hold:

(1) If any k of these n people cooperate, then—by combining their k shares of the secret—they can compute the secret s (preferably efficiently).

(2) If any $k' < k$ of these n people cooperate, then by combining their k' shares they learn *nothing* about the secret s. (Informally, to "learn nothing" about the secret means that no k' shares of the secret allow one to infer that s comes from any particular $S' \subset S$. Note that just "splitting up the bits" of the secret violates condition 2.)



x	$f_2(x)$	$f_2(x)$ mod 5
0	2	2
1	6	1
2	12	2
3	20	0
4	30	0

Figure 7.14 (a) A function $f(x) = a + bx + cx^2$ as a tool for secret sharing: the secret is $f(0) = a$; the four secret shares are $f(i)$ mod 5 for $i \in \{1, 2, 3, 4\}$. (b) Even knowing $f(1) \equiv_5 1$ and $f(2) \equiv_5 2$, we don't know $f(0)$ mod 5; there are polynomials consistent with $f(0) \equiv_5 m$ for every $m \in \{0, 1, 2, 3, 4\}$. Here we see five different polynomials, where $f_i(0) \equiv_5 i$, $f_i(1) \equiv_5 1$, and $f_i(2) \equiv_5 2$.

The basic idea will be to define a polynomial $f(x)$, and distribute the value of $f(i)$ as the the ith "share" of the secret; the secret itself will be $f(0)$. Why will this be useful? Imagine that $f(x) = a + bx$. (The secret is thus $f(0) = a + b \cdot 0 = a$.) Knowing that $f(1) = 17$ tells you that $a + b = 17$, but it doesn't tell you anything about a itself: for every possible value of the secret, there's a value of b that makes $a + b = 17$. But knowing $f(1) = 17$ and $f(2) = 42$ lets you solve for $b = 25, a = -8$. If $f(x) = a + bx + cx^2$, then knowing $f(x_1)$ and $f(x_2)$ gives you two equations and three unknowns—but you *can* solve for a if you know the value of $f(x)$ for *three* different values of x. In general, knowing k values of a polynomial f of degree k lets you compute $f(0)$, but any $k - 1$ values of f are consistent with *any* value of $f(0)$. And this result remains true if, instead of using the value $f(x)$ as the share of the

secret, we instead use $f(x) \bmod p$, for some prime p. (See p. 7-38, and Figure 7.14.) Thus, concretely, to distribute shares of a secret $m \in \{0, 1, 2, 3, 4\}$, here's what we'd do:

- First, choose $a_1, \ldots, a_{k-1}$ uniformly and independently at random from the set $\{0, 1, 2, 3, 4\}$.
- Second, define the function $f(x) = m + \sum_{i=1}^{k-1} a_i x^i$.
- Then, to the ith person who's sharing the secret, send the pair $\langle i, f(i) \bmod 5 \rangle$.

Then f was built from k unknown coefficients: m, and $a_1, a_2, \ldots, a_{k-1}$. For example, if $k = 3$, no two people can reconstruct the secret—but three people *can*. Only one quadratic function passes through three given points.

Computer Science Connections

Error Correction with Reed–Solomon Codes

In Section 4.2, we discussed *error-correcting codes:* we encode a *message* m as a *codeword* $c(m)$, so that m is (efficiently) recoverable from $c(m)$, or even from a mildly corrupted codeword $c' \approx c(m)$. (Note the difference in motivation with cryptography: in error-correcting codes, we want a codeword that makes computing the original message very easy; in cryptography, we want a ciphertext that makes computing the original message very hard.) The key property that we seek is that if $m_1 \neq m_2$, then $c(m_1)$ and $c(m_2)$ are "very different," so that decoding c' simply corresponds to finding the m that minimizes the difference between c' and $c(m)$.

We've discussed *Reed–Solomon codes*, one of the classic schemes for error-correcting codes. Under Reed–Solomon codes, to encode a message $m \in \mathbb{Z}^k$, we define the polynomial $p_m(x) = \sum_{i=1}^{k} m_i x^i$, and then encode m as $\langle p_m(1), p_m(2), \ldots, p_m(n)\rangle$. (We choose n much bigger than k, to achieve the desired error-correction properties.) For example, for the messages $m_1 = \langle 1, 3, 2\rangle$ and $m_2 = \langle 3, 0, 3\rangle$, we have $p_{m_1}(x) = x + 3x^2 + 2x^3$ and $p_{m_2}(x) = 3x + 3x^3$. For $n = 6$, we have the codewords (for m_1 and m_2, respectively) $\langle 6, 30, 84, 180, 330, 546\rangle$ and $\langle 6, 30, 90, 204, 390, 666\rangle$. (See Figure 7.15.)

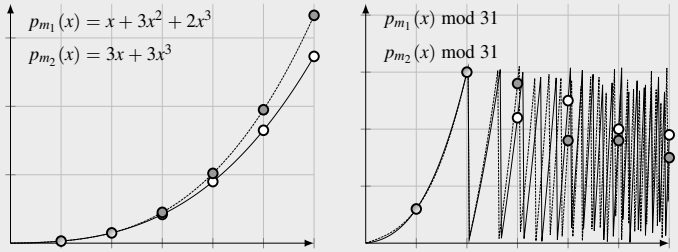


Figure 7.15 Two polynomials derived from Reed–Solomon codes, computed using both normal arithmetic and in $\mathbb{Z}_{31}$. (The polynomials have the same values for $x \in \{0, 1, 2\}$, and differ for all other values of x.)

The key point is that *two distinct polynomials of degree k agree on at most k inputs,* which means that the codewords for m_1 and m_2 will be very different. (Here $p_{m_1}(x)$ and $p_{m_2}(x)$ agree on $x \in \{1, 2\}$, but not on $x \in \{3, 4, 5, 6\}$.) The theorem upon which this difference rests is important enough to be called the *Fundamental Theorem of Algebra* (Theorem 2.58).

While this fact about Reed–Solomon codes is nice, it's already evident that the numbers in the codewords get really big—546 and 666 are very big relative to the integers in the original messages! In real Reed–Solomon codes, there's another trick that's used: every value is stored *modulo a prime.* (We encode the message m by computing $p_m(i) \bmod q$ for a prime q, for many values of i, instead of just computing $p_m(i)$.) We now encode

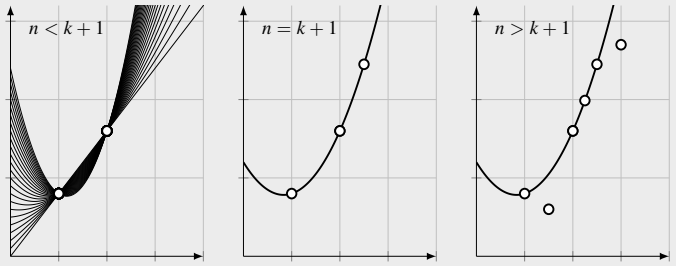


Figure 7.16 The combined message of Reed–Solomon codes and Shamir secret-sharing (p. 7-36). Imagine a degree-k polynomial p that is unknown to you (here, $k = 2$). You're given the value of $p(x)$ for n distinct values of x.
- If $n < k + 1$, you know *nothing* about the constant term of p. (Secrets kept!)
- If $n = k + 1$, you can compute every coefficient of p. (Secrets shared!)
- If $n > k + 1$, you can find p even if some points are wrong. (Errors corrected!)

(The exact same relationships hold if you're given $p(x) \bmod q$ instead of $p(x)$.)

a message $m \in \mathbb{Z}_q^k$ with a codeword in $\mathbb{Z}_q^n$. And it turns out that everything important about polynomials remains true if we take all values modulo a prime q: *two distinct polynomials of degree k agree mod q on at most k inputs,* too. (Again, see Figure 7.15.) In fact, the Shamir secret sharing scheme (p. 7-36) and error correction using Reed–Solomon codes can be viewed as part of the same continuum: evaluating a polynomial of degree k on fewer that $k+1$ points keeps a secret, evaluating it on $k+1$ points reveals a secret, and evaluating it on more than $k+1$ points allows for errors to be detected and corrected. (See Figure 7.16.)

EXERCISES

7.38 The *Sieve of Eratosthenes* returns a list of all prime numbers up to a given integer n by creating a list of candidate primes $\langle 2, 3, \ldots, n \rangle$, and repeatedly marking the first unmarked number p as prime and striking out all entries in the list that are multiples of p. (See Figure 7.17.) Write pseudocode to describe the Sieve of Eratosthenes.

7.39 Run the Sieve of Eratosthenes algorithm, by hand, to find all primes less than 100.

7.40 *(programming required.)* Implement the Sieve of Eratosthenes in a programming language of your choice. Use your program to compute all primes up to 100,000. How many are there?

7.41 *(programming required.)* Earlier, we suggested another algorithm to compute all primes up to n: for each $i = 2, 3, \ldots, n$, test whether i is divisible by any integer between 2 and $\sqrt{i}$. Implement this algorithm too, and compare their execution times for $n = 100{,}000$. What happens for $n = 500{,}000$?

7.42 Assume that each number k is crossed off by the Sieve of Eratosthenes *every time* a divisor of it is found. (For example, 6 is crossed off when 2 is the prime in question, *and* when 3 is the prime in question.) Prove that the total number of crossings-out by **sieve**(n) is $\leq H_n \cdot n$, where H_n is the nth harmonic number. (See Definition 5.8.)

7.43 Use the Prime Number Theorem to estimate the number of primes between $2^{127} + 1$ and 2^{128}.

7.44 Use the Prime Number Theorem to estimate the 2^{128}th-largest prime.

7.45 Use the Prime Number Theorem to argue that, roughly, the probability that a randomly chosen number close to n is prime is about $1/\ln n$. *(Hint: what does* $\text{primes}(n) - \text{primes}(n-1)$ *represent?)*

7.46 Using the same technique as in Example 7.8, estimate the number of 6-digit primes. Then, using the Sieve (see Exercise 7.38) or some other custom-built program, determine how far off the estimate was.

Let p be an arbitrary prime number and let a be an arbitrary nonnegative integer. Prove the following facts.

7.47 If $p \nmid a$, then $\text{GCD}(p, a) = 1$.

7.48 For any positive integer k, we have $p \mid a^k$ if and only if $p \mid a$. *(Hint: use induction and Lemma 7.19.)*

7.49 For any integers $n, m \in \{1, \ldots, p-1\}$, we have that $p \nmid nm$.

7.50 For any integer m and any prime number q distinct from p (that is, $p \neq q$), we have $m \equiv_p a$ and $m \equiv_q a$ if and only if $m \equiv_{pq} a$. *(Hint: think first about the case $a = 0$; then generalize.)*

7.51 If $0 \leq a < p$, then $a^2 \equiv_p 1$ if and only if $a \in \{1, p-1\}$. *(You may use the theorem from p. 7-38: if $f(x)$ is a polynomial of degree k, and q is a prime, then either $f(a) \bmod q = 0$ for every $a \in \mathbb{Z}_q$, or the equation $f(x) = 0$ has at most k solutions for $x \in \mathbb{Z}_q$.)*

7.52 *Using the brute force algorithm (test all candidate divisors) and paper and pencil only,* determine whether 54321 and 12345 are relatively prime.

7.53 Do the same for 209 and 323.

7.54 Do the same for 101 and 1100.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40

⋮

Figure 7.17 A few iterations of the Sieve of Eratosthenes. Primes are highlighted as they're discovered; numbers are written in light gray as they're crossed off.

7.55 Using the Extended Euclidean algorithm (executed *by hand*), compute $\text{GCD}(n,m)$ and two integers x and y such that $xn + ym = \text{GCD}(n,m)$ for $n = 60$ and $m = 93$.

7.56 Repeat for $n = 24$ and $m = 28$.

7.57 Repeat for $n = 74$ and $m = 13$.

7.58 Prove the following extension to Lemma 7.17: there are *infinitely many pairs* of integers x, y such that $xn + ym = \text{GCD}(n,m)$, for any nonnegative integers n and m.

7.59 Prove the extension of Lemma 7.17 to $k \geq 2$ integers: if $\text{GCD}(a_1, \ldots, a_k) = d$, then there exist integers $x_1, \ldots, x_k$ such that $\sum_{i=1}^{k} a_i x_i = d$. (Define $\text{GCD}(x_1, x_2, \ldots, x_k)$ as $\text{GCD}(x_1, \text{GCD}(x_2, \ldots, x_k))$ for $k \geq 3$.)

7.60 Prove Theorem 7.18 (the correctness of the Extended Euclidean algorithm) by induction on n: for arbitrary positive integers n and m with $n \leq m$, **extended-Euclid**(n,m) returns three integers x, y, r such that $r = \text{GCD}(n,m) = xn + ym$.

7.61 *(programming required.)* Write a program that implements the Extended Euclidean algorithm. (Recommended: if you did Exercises 7.11–7.16, compute m mod n and $\lfloor \frac{m}{n} \rfloor$ with a single call to **mod-and-div-faster**(m,n).)

I have a friend named Nikki, who's from New Zealand. Nikki and I went out to eat together, and I paid for both dinners. She was going to pay me back, in cash—but she had only New Zealand dollars [NZD]. (I was happy to take NZDs.) Nikki had a giant supply of 5NZD bills; I had a giant supply of 5 U.S. dollar [USD] bills. At the time, the exchange rate was 5NZD = 3USD (or close enough to 5 : 3 for two friends to call it good).

7.62 Prove that Nikki can pay me exactly 4USD in value, through only the exchange of 5NZD and 5USD bills.

7.63 In Exercise 7.62, was there something special about the number 4? Identify for which nonnegative integers x Nikki can pay me back exactly x USD in value, through only the exchange of 5NZD and 5USD bills, and prove your answer.

7.64 In Exercises 7.62–7.63, was there something special about the number 3? Suppose that, due to geopolitical turmoil and a skyrocketing of the price of wool, the 5NZD bill is now worth b USDs, for some $b \equiv_5 3$. I still have many 5USD bills, and Nikki still has the equivalent of many b USD bills. What amounts can Nikki now pay me? Prove your answer.

7.65 In an unexpected twist, I run out of U.S. dollars and Nikki runs out of New Zealand dollars. But I discover that I have a giant supply of identical Israeli Shekel notes, each of which is worth k USD. And Nikki discovers that she has a giant supply of identical Thai Baht notes, each of which is worth ℓ USD. (Assume k and ℓ are integers.) What amounts can she pay me now? Again, prove your answer.

7.66 Prove that any two consecutive integers (n and $n+1$) are always relatively prime.

7.67 Prove that any two consecutive Fibonacci numbers are always relatively prime.

7.68 Prove that two integers a and b are relatively prime if and only if there is no prime number p such that $p \mid a$ and $p \mid b$. (Notice that this claim differs from the definition of relative primality, which required that there be no *integer* $n \geq 2$ such that $n \mid a$ and $n \mid b$.)

7.69 Let a and b be relatively prime. Let c be relatively prime to both a and b. Prove that c and ab are also relatively prime.

7.70 Let a and b be relatively prime. Prove that, for any integer n, we have that both $a \mid n$ and $b \mid n$ if and only if $ab \mid n$.

7.71 Let a and b be relatively prime. Prove that, for every integer m, there exist integers x and y such that $ax + by = m$.

7.72 Describe the integers $x \in \mathbb{Z}^{\geq 0}$ that satisfy the constraints x mod $13 = 6$ and x mod $19 = 2$. More precisely, describe these integers as a set $\{a + bk : k \in \mathbb{Z}^{\geq 0}\}$, where a is smallest x satisfying the constraints, $a + b$ is the next smallest, $a + 2b$ is the next smallest, etc. (You have to figure out the values of a and b.)

7.73 Repeat for the constraints x mod $21 = 3$ and x mod $11 = 2$.

7.74 Repeat for the constraints x mod $6 = 3$ and x mod $7 = 3$.

7.75 Repeat for the constraints x mod $5 = 4$ and x mod $6 = 5$ and x mod $7 = 2$.

7.76 Repeat for the constraints x mod $5 = 4$ and x mod $6 = 5$ and x mod $7 = 3$.

7.77 Let n and m be relatively prime, and let $a \in \mathbb{Z}_n$ and $b \in \mathbb{Z}_m$. Define y^* to be the unique value in $\mathbb{Z}_{nm}$ such that $y^* \bmod n = a$ and $y^* \bmod m = b$, whose existence is guaranteed by Theorem 7.21. Prove that an integer $x \in \mathbb{Z}_{nm}$ satisfies $x \bmod n = a$ and $x \bmod m = b$ *if and only if* x satisfies $x \bmod nm = y^*$.

Show that relative primality was mandatory for the Chinese Remainder Theorem. Considering two integers n and m that are not necessarily relatively prime:

7.78 Prove that, for some $a \in \mathbb{Z}_n$ and $b \in \mathbb{Z}_m$, it may be the case that *no* $x \in \mathbb{Z}_{nm}$ satisfies $x \bmod n = a$ and $x \bmod m = b$.

7.79 Prove that, for some $a \in \mathbb{Z}_n$ and $b \in \mathbb{Z}_m$, there may be *more than one* $x \in \mathbb{Z}_{nm}$ satisfies $x \bmod n = a$ and $x \bmod m = b$.

7.4 Multiplicative Inverses

> *Tous pour un, un pour tous.*
> All for one, one for all.
>
> Alexandre Dumas (1802–1870)
> *Les Trois Mousquetaires [The Three Musketeers]* (1844)

For any integer $n \geq 2$, let $\mathbb{Z}_n$ denote the set $\{0, 1, \ldots, n-1\}$. In this section, we'll discuss *arithmetic over* $\mathbb{Z}_n$—that is, arithmetic where we think of all expressions by considering their value modulo n. For example, when $n = 9$, the expressions $4 + 6$ and $8 \cdot 7$ are equivalent to 1 and 2, respectively, because $10 \bmod 9 = 1$ and $56 \bmod 9 = 2$. When $n = 10$, the expressions $4 + 6$ and $8 \cdot 7$ are equivalent to 0 and 6, respectively.

We have already encountered addition and multiplication in the world of modular arithmetic (for example, in Theorem 7.4). But we haven't yet defined subtraction or division. (Theorem 7.4 also introduced exponentiation over $\mathbb{Z}_n$, and it turns out that, along with division, exponentiation in modular arithmetic will form the foundation of the RSA cryptographic system; see Section 7.5.) Subtraction turns out to be fairly straightforward (see Exercise 7.81), but division will be a bit trickier than $+$, $\cdot$, and $-$. In this section, we'll introduce what division over $\mathbb{Z}_n$ even means, and then discuss algorithms to perform modular division.

7.4.1 The Basic Definitions

Before we introduce any of the technical definitions, let's start with a tiny bit of intuition about why there's something potentially interesting going on with division in $\mathbb{Z}_n$. For concreteness, here's a small example in $\mathbb{Z}_9$:

Example 7.17: Halving some numbers in $\mathbb{Z}_9$.
In $\mathbb{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, where every expression's value is understood mod 9, what element of $\mathbb{Z}_9$ is half of 6? Half of 8? Half of 5?

Solution. What number is half of 6? Well, easy: it's obviously 3. (Why? Because 6 is double 3, and therefore 3 is half of 6—or, in other words, 3 is half of 6 because $3 \cdot 2$ is 6.) And what number is half of 8? Easy again: it's 4 (because $4 \cdot 2$ is 8).

Okay, what number is half of 5? The first temptation is to say that it's 2.5 (or $\frac{5}{2}$, if you're more of a fan of fractions)—but that doesn't make sense as an answer: after all, which element of $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ is 2.5?!? So the next temptation is to say that there is *no* number that's half of 5. (After all, in normal nonmodular arithmetic, there is no integer that's half of 5.) But that's not right either: there *is* an answer in $\mathbb{Z}_9$, even if it doesn't quite match our intuition. The number that's half of 5 is in fact 7(!). Why? Because $7 \cdot 2$ is 5. (Remember that we're in $\mathbb{Z}_9$, and $14 \bmod 9 = 5$.) So, in $\mathbb{Z}_9$, the number 7 is half of 5.

In fact, we can find a number in $\mathbb{Z}_9$ that's half of each element of $\mathbb{Z}_9$. For each $b \in \mathbb{Z}_9$, here is the value of $a \in \mathbb{Z}_9$ such that $2a = b$:

b	0	1	2	3	4	5	6	7	8
a	0	5	1	6	2	7	3	8	4.

(You can check that $0 \cdot 2 \equiv_9 0$, and $5 \cdot 2 \equiv_9 1$, and so forth.)

Example 7.17 illustrates the basic idea of division in $\mathbb{Z}_n$: we'll define $\frac{a}{b}$ as the number k such that $k \cdot b$ is equivalent to a in $\mathbb{Z}_n$. To make this idea formal, we'll need a few definitions about modular arithmetic. But, first, we'll go back to "normal" arithmetic, for the real numbers, and introduce the two key concepts: *identity* and *inverse*.

Problem-solving tip: When you encounter a new definition, it's often helpful to try it out in a setting that you already understand well. For example, it's easier understand Manhattan distance in $\mathbb{R}^2$ (see Example 2.40) before trying to understand it for general $\mathbb{R}^n$. In this case, you've grasped division in $\mathbb{R}$ since, what, second grade—so, before trying to make sense of the definitions for $\mathbb{Z}_n$, try to consider the analogy of each definition for $\mathbb{R}$.

Multiplicative inverses in $\mathbb{R}$

The number 1 is called the *multiplicative identity,* because it has the property that

$$x \cdot 1 = 1 \cdot x = x, \text{ for any } x \in \mathbb{R}.$$

(We've encountered identities a number of times already. Definition 2.43 introduced the identity matrix I, where $MI = IM = M$ for any matrix M. And Exercises 3.13–3.16 explored identities of logical connectives; for example, the identity of $\vee$ is False, because $p \vee \text{False} \equiv \text{False} \vee p \equiv p$ for any proposition p.)

The *multiplicative inverse* of a number x is the number by which we have to multiply x to get 1 (that is, to get the multiplicative identity) as the result. In other words, the multiplicative inverse of $x \in \mathbb{R}$ is the real number x^{-1} such that $x \cdot x^{-1} = 1$. (We generally denote the multiplicative inverse of x as x^{-1}, though it may be easier to think about the multiplicative inverse of x as $\frac{1}{x}$, because $x \cdot \frac{1}{x} = 1$. Actually the "$^{-1}$" notation is in general ambiguous between denoting inverse and denoting exponentiation with a negative exponent—though these concepts match up perfectly for the real numbers. Exercise 7.100 addresses negative exponents in modular arithmetic.) For example, the multiplicative inverse of 8 is $\frac{1}{8} = 0.125$, because $8 \cdot 0.125 = 1$.

When we think of *dividing* $y \in \mathbb{R}$ *by* $x \in \mathbb{R}$, we can instead think of this operation as *multiplying* y *by* x^{-1}. For example, we have $7/8 = 7 \cdot 8^{-1} = 7 \cdot 0.125 = 0.875$.

Not every real number has a multiplicative inverse: specifically, there is no number that yields 1 when it's multiplied by 0, so 0^{-1} doesn't exist. (And we can't divide y by 0, because 0^{-1} doesn't exist.) But for any $x \neq 0$, the multiplicative inverse of x does exist, and it's given by $x^{-1} = \frac{1}{x}$.

Multiplicative inverses in $\mathbb{Z}_n$

Now let's turn to the analogous definitions in the world of modular arithmetic, in $\mathbb{Z}_n$. Notice that 1 is still the multiplicative identity, for any modulus n: for any $x \in \mathbb{Z}_n$, it is the case that $x \bmod n = 1 \cdot x \bmod n = x \cdot 1 \bmod n$. The definition of the multiplicative inverse in $\mathbb{Z}_n$ is identical to the definition in $\mathbb{R}$:

Definition 7.23: Multiplicative inverse.
Let $n \geq 2$ be any integer, and let $a \in \mathbb{Z}_n$ be arbitrary. The *multiplicative inverse of a in $\mathbb{Z}_n$* is the number $a^{-1} \in \mathbb{Z}_n$ such that $a \cdot a^{-1} \equiv_n 1$. If there is no element $x \in \mathbb{Z}_n$ such that $ax \equiv_n 1$, then a^{-1} is undefined.

Writing tip: Let $a \in \mathbb{Z}_n$. The notation a^{-1} doesn't explicitly indicate the modulus n anywhere, and the value of n matters! If there's *any* ambiguity about the value of n, then be sure to specify it clearly in your words surrounding the notation.

(Note that Definition 7.23 describes the multiplicative inverse as "the" a^{-1} that has the desired property. In Exercise 7.93, you'll show that there can't be two distinct values $b, c \in \mathbb{Z}_n$ where $ab \equiv_n ac \equiv_n 1$.) Here are a few examples of multiplicative inverses, and of a case where there is no multiplicative inverse:

Example 7.18: Some multiplicative inverses.
The multiplicative inverse of 2 in $\mathbb{Z}_9$ is $2^{-1} = 5$, because $2 \cdot 5 = 10 \equiv_9 1$, and the multiplicative inverse of 1 in $\mathbb{Z}_9$ is $1^{-1} = 1$, because $1 \cdot 1 \equiv_9 1$.

The multiplicative inverse of 7 in $\mathbb{Z}_{11}$ is 8 because $7 \cdot 8 = 56 \equiv_{11} 1$, and the multiplicative inverse of 7 in $\mathbb{Z}_{13}$ is 2 because $7 \cdot 2 = 14 \equiv_{13} 1$.

Example 7.19: A nonexistent multiplicative inverse.
The number 3 has no multiplicative inverse in $\mathbb{Z}_9$, as the following table shows:

$$\begin{array}{lll} 3 \cdot 0 = 0 \equiv_9 0 & 3 \cdot 3 = 9 \equiv_9 0 & 3 \cdot 6 = 18 \equiv_9 0 \\ 3 \cdot 1 = 3 \equiv_9 3 & 3 \cdot 4 = 12 \equiv_9 3 & 3 \cdot 7 = 21 \equiv_9 3 \\ 3 \cdot 2 = 6 \equiv_9 6 & 3 \cdot 5 = 15 \equiv_9 6 & 3 \cdot 8 = 24 \equiv_9 6. \end{array}$$

None of these nine entries is equivalent to 1 modulo 9, so there is no 3^{-1} in $\mathbb{Z}_9$.

Example 7.20: Multiplicative inverses in $\mathbb{Z}_7$.
Find the values of 0^{-1}, 1^{-1}, 2^{-1}, 3^{-1}, 4^{-1}, 5^{-1}, and 6^{-1} in $\mathbb{Z}_7$.

Solution. The simplest way (though not necessarily the fastest way!) to solve this problem is by building a multiplication table for $\mathbb{Z}_7$, as shown in Figure 7.18. (The entry in row a and column b of the table is the value ab mod 7—for example, $4 \cdot 5 = 20 = 2 \cdot 7 + 6$, so the entry in row 4, column 5 is the number 6.) For each row a, the value a^{-1} we seek is the column that has a 1 in it, if there is such a column in that row. (And there is a 1 in every row except $a = 0$.) Thus in $\mathbb{Z}_7$ we have $1^{-1} = 1$ and $2^{-1} = 4$ and $3^{-1} = 5$ and $4^{-1} = 2$ and $5^{-1} = 3$ and $6^{-1} = 6$—and 0^{-1} is undefined.

Taking it further: The field of mathematics called *abstract algebra* focuses on giving and analyzing very general definitions of structures that satisfy certain properties—allowing apparently disparate objects (like Boolean logic and Rubik's cubes) to be studied at the same time. For example, a *group* is a pair $\langle G, \cdot \rangle$, where G is a set of objects and $\cdot$ is a binary operator on G, where certain properties are satisfied:

- *Closure:* for any $a, b \in G$, we have $a \cdot b \in G$.
- *Associativity:* for any $a, b, c \in G$, we have $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- *Identity:* there is an *identity element* $e \in G$ with the property that $a \cdot e = e \cdot a = a$ for every $a \in G$.
- *Inverse:* for every $a \in G$, there exists $b \in G$ such that $a \cdot b = b \cdot a = e$ (where e is the identity element).

For example, $\langle \mathbb{Z}, + \rangle$ is a group. As we'll see, so too is $\langle \mathbb{Z}_p - \{0\}, \cdot \rangle$, where $\cdot$ denotes multiplication and p is any prime integer. Despite the very abstract nature of these definitions—and other more general or more specific algebraic structures, like *semigroups, rings,* and *fields*—they are a surprisingly useful way of understanding properties of $\mathbb{Z}_p$. See any good textbook on abstract algebra for more detail.

7.4.2 When Multiplicative Inverses Exist (and How to Find Them)

Examples 7.18–7.20 might inspire you to ask a question that will turn out to be both useful and reasonably simple to answer: *under what circumstances does a particular number $a \in \mathbb{Z}_n$ have a multiplicative inverse?* As we saw with arithmetic over $\mathbb{R}$, there's never a multiplicative inverse for 0 in any $\mathbb{Z}_n$ (because, for any x, we have $x \cdot 0 = 0 \not\equiv_n 1$)—but what happens for nonzero a?

To take one particular case, we just found that $2^{-1} = 5$ in $\mathbb{Z}_9$ but that 3^{-1} does not exist in $\mathbb{Z}_9$. It's worth reflecting a bit on "why" 3^{-1} failed to exist in $\mathbb{Z}_9$. There are a lot of ways to think about it, but here's one convenient way to describe what went wrong: any multiple of 3 is (obviously!) divisible by 3, and numbers divisible by 3 are never one more than multiples of 9. In other words, the only possible values of $3x \bmod 9$ are $\{0, 3, 6\}$—a set that fails to include 1. (Recall from Definition 7.23 that, for 3^{-1} to exist in $\mathbb{Z}_9$, we'd have to have been able to find an x such that $3x \equiv_9 1$.) Similarly, 6^{-1} doesn't exist in $\mathbb{Z}_9$: again, the only possible values of $6x \bmod 9$ are $\{0, 3, 6\}$, which once again does not include 1.

These observations should be reminiscent of the concepts that we discussed in Section 7.3: for a number $a \in \mathbb{Z}_n$, we seem to be unable to find a multiplicative inverse a^{-1} in $\mathbb{Z}_n$ whenever a and n share a common divisor $d > 1$. In other words, when a and n are not relatively prime, then a^{-1} fails to exist in $\mathbb{Z}_n$. (That's

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Figure 7.18 The multiplication table for $\mathbb{Z}_7$.

because any multiple xa of a will also be divisible by d, and so $xa \bmod n$ will also be divisible by d, and therefore $xa \bmod n$ will not equal 1.) In fact, not being relatively prime to n is the *only* way to fail to have a multiplicative inverse in $\mathbb{Z}_n$, as we'll prove. (Note that $0 \in \mathbb{Z}_n$ is *not* relatively prime to n, because $\text{GCD}(n, 0) \neq 1$.)

Theorem 7.24: Existence of Multiplicative Inverses.
Let $n \geq 2$ and $a \in \mathbb{Z}_n$. Then a^{-1} exists in $\mathbb{Z}_n$ if and only if n and a are relatively prime.

Proof. By definition, a multiplicative inverse of a exists in $\mathbb{Z}_n$ precisely when there exists an integer x such that $ax \equiv_n 1$. (The definition actually requires $x \in \mathbb{Z}_n$, not just $x \in \mathbb{Z}$, but see Exercise 7.99.) But $ax \equiv_n 1$ means that ax is one more than a multiple of n—that is, there exists some integer y such that $ax + yn = 1$. In other words,

$$a^{-1} \text{ exists in } \mathbb{Z}_n \text{ if and only if there exist integers } x, y \text{ such that } ax + yn = 1. \qquad (*)$$

Observe that $(*)$ echoes the form of Lemma 7.17 (and thus also echoes the output of the Extended Euclidean algorithm), and we can use this fact to prove the theorem. We'll prove the two directions of the theorem separately:

If a^{-1} exists in $\mathbb{Z}_n$, then a and n are relatively prime. We'll prove the contrapositive. Suppose that a and n are not relatively prime—that is, suppose that $\text{GCD}(a, n) = d$ for some $d > 1$. We will show that a^{-1} does not exist in $\mathbb{Z}_n$. Because $d \mid a$ and $d \mid n$, there exist integers c and k such that $a = cd$ and $n = kd$. But then, for *any* integers x and y, we have that

$$ax + yn = cdx + ykd = d(cx + yk)$$

and thus $d \mid (ax + yn)$. Thus there are no integers x, y for which $ax + yn = 1$ and therefore, by $(*)$, a^{-1} does not exist in $\mathbb{Z}_n$.

If a and n are relatively prime, then a^{-1} exists in $\mathbb{Z}_n$. Suppose that a and n are relatively prime. We'll prove that a^{-1} exists. Because a and n are relatively prime, by definition we have that $\text{GCD}(a, n) = 1$. Thus, by the correctness of the Extended Euclidean algorithm (Theorem 7.18), the output of **extended-Euclid**(a, n) is $\langle x, y, 1 \rangle$ for integers x, y such that $xa + yn = \text{GCD}(a, n) = 1$. The fact that **extended-Euclid**(a, n) outputs integers x and y such $xa + yn = 1$ means that such an x and y must exist—and so, by $(*)$, a^{-1} exists in $\mathbb{Z}_n$. □

Note that this theorem is consistent with the examples that we saw previously: we found 1^{-1} and 2^{-1} but not 3^{-1} in $\mathbb{Z}_9$ (Examples 7.18 and 7.19; 1 and 2 are relatively prime to 9, but 3 is not), and we found multiplicative inverses for all nonzero elements of $\mathbb{Z}_7$ (Example 7.20; all of $\{1, 2, \ldots, 6\}$ are relatively prime to 7).

Two implications of Theorem 7.24

There are two useful implications of this result. First, when the modulus is prime, multiplicative inverses exist for *all* nonzero elements of $\mathbb{Z}_n$, because every nonzero $a \in \mathbb{Z}_n$ and n are relatively prime for any prime number n.

Corollary 7.25. If p is prime, then every nonzero $a \in \mathbb{Z}_p$ has a multiplicative inverse in $\mathbb{Z}_p$.

(We saw an example of this corollary in Example 7.20, where we identified the multiplicative inverses of all nonzero elements in $\mathbb{Z}_7$.)

The second useful implication of Theorem 7.24 is that, whenever the multiplicative inverse of a exists in $\mathbb{Z}_n$, we can efficiently *compute* a^{-1} in $\mathbb{Z}_n$ using the Extended Euclidean algorithm—specifically, by running the algorithm in Figure 7.19a. (This problem also nicely illustrates a case in which proving a structural fact vastly improves the efficiency of a calculation—the algorithm in Figure 7.19a is *way* faster than building the entire multiplication table, as we did in Example 7.20.)

Corollary 7.26. For any $n \geq 2$ and $a \in \mathbb{Z}_n$, **inverse**(a, n) returns the value of a^{-1} in $\mathbb{Z}_n$.

Proof. We just proved that a^{-1} exists if and only if **extended-Euclid**(a, n) returns $\langle x, y, 1\rangle$. In this case, we have $xa + yn = 1$ and therefore $xa \equiv_n 1$. Defining $a^{-1} = x \bmod n$ ensures that $a \cdot (x \bmod n) \equiv_n 1$, as required. (Again, see Exercise 7.99.) □

Here are two examples, one replicating the calculation of 5^{-1} in $\mathbb{Z}_7$ from Example 7.20:

Example 7.21: 5^{-1} in $\mathbb{Z}_7$, again, and 7^{-1} in $\mathbb{Z}_9$.
To compute 5^{-1}, we run the Extended Euclidean algorithm on 5 and 7, as in Figure 7.19b, which returns $\langle 3, -2, 1\rangle$, implying that $3 \cdot 5 + -2 \cdot 7 = 1 = \text{GCD}(5, 7)$. Therefore **inverse**$(5, 7)$ returns $3 \bmod 7 = 3$. And, indeed, $3 \cdot 5 \equiv_7 1$.

In Example 7.15, we saw that **extended-Euclid**$(7, 9) = \langle 4, -3, 1\rangle$. Thus 7 and 9 are relatively prime, and 7^{-1} in $\mathbb{Z}_9$ is $4 \bmod 9 = 4$. And indeed $7 \cdot 4 = 28 \equiv_9 1$.

7.4.3 Fermat's Little Theorem

We'll now make use of the results that we've developed so far—specifically Corollary 7.25—to prove a surprising and very useful theorem, called *Fermat's Little Theorem*, which states that a^{p-1} is equivalent to 1 mod p, for any prime number p and any $a \neq 0$. (And we'll see why this result is useful for cryptography in Section 7.5.)

Taking it further: Fermat's Little Theorem is named after Pierre de Fermat (1601–1665), a French mathematician. Fermat's Little Theorem is the second-most famous theorem named after him; his more famous theorem is called *Fermat's Last Theorem,*

```
inverse(a, n):
Input: a ∈ Z_n and n ≥ 2
Output: a^{-1} in Z_n, if it exists
1 x, y, d := extended-Euclid(a, n)
2 if d = 1 then
3     return x mod n          // xa + yn = 1, so xa ≡_n 1.
4 else
5     return "no inverse for a exists in Z_n."
```

(a) The pseudocode.

To compute $5^{-1} \in \mathbb{Z}_7$, we call **inverse**$(5,7)$:

extended-Euclid$(5,7)$

  extended-Euclid$(\underbrace{7 \bmod 5}_{=2}, 5)$

    extended-Euclid$(\underbrace{5 \bmod 2}_{=1}, 2)$

    $= 1, 0, 1$

  $= -2, 1, 1$

$= 3, -2, 1.$

Therefore **inverse**$(5,7)$ returns 3 mod 7, which is 3.

(b) An example: computing that $5^{-1} \in \mathbb{Z}_7$ is 3.

Figure 7.19 An algorithm for computing multiplicative inverses, and an example.

which states the following:

For any integer $k \geq 3$, there are no positive integers x, y, z satisfying $x^k + y^k = z^k$.

There *are* integer solutions to the equation $x^k + y^k = z^k$ when $k = 2$—the so-called *Pythagorean triples*, like $\langle 3,4,5\rangle$ (where $3^2 + 4^2 = 9 + 16 = 25 = 5^2$) and $\langle 7,24,25\rangle$ (where $7^2 + 24^2 = 49 + 576 = 625 = 25^2$). But Fermat's Last Theorem states that there are no integer solutions when the exponent is larger than 2.

The history of Fermat's Last Theorem is convoluted and about as fascinating as the history of any mathematical statement can be. In the 17th century, Fermat conjectured his theorem, and scrawled—in the margin of one of his books on mathematics—the words "I have discovered a truly marvelous proof, which this margin is too narrow to contain . . . ." The conjecture, and Fermat's assertion, were found after Fermat's death—but the proof that Fermat claimed to have discovered was never found. And it seems almost certain that he did not have a correct proof of this claim. Some 350 years later, in 1995, the mathematician Andrew Wiles published a proof of Fermat's Last Theorem, building on work by a number of other 20th-century mathematicians.

The history of the Fermat's Last Theorem—including the history of Fermat's conjecture and the centuries-long quest for a proof—has been the subject of a number of books written for a nonspecialist audience; see [119] for a compelling account. This story may be as dramatic as one about the history of proofs ever gets.

Before we can prove Fermat's Little Theorem itself, we'll need a preliminary result. We will show that, for any prime p and any nonzero $a \in \mathbb{Z}_p$, the first $p-1$ nonzero multiples of a—that is, the values $\{a, 2a, 3a, \ldots, (p-1)a\}$—are precisely the $p-1$ nonzero elements of $\mathbb{Z}_p$. Or, to state this claim in a slightly different way, we will prove that the function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ defined by $f(k) = ak \bmod p$ is both one-to-one and onto (and also satisfies $f(0) = 0$). Here is a formal statement of the result:

Lemma 7.27: $\{1, 2, \ldots p-1\}$ and $\{1a, 2a, \ldots (p-1)a\}$ are equivalent mod p.
For prime p and any $a \in \mathbb{Z}_p$ where $a \neq 0$, we have

$$\{1 \cdot a \bmod p, 2 \cdot a \bmod p, \ldots, (p-1) \cdot a \bmod p\} = \{1, 2, \ldots, p-1\}.$$

Before we dive into a proof, let's check an example:

Example 7.22: $\{ai \bmod 11\}$ *vs.* $\{i \bmod 11\}$.
Consider the prime $p = 11$ and two values of a, namely $a = 2$ and $a = 5$. Then we have

i	1	2	3	4	5	6	7	8	9	10
$2i \bmod 11$	2	4	6	8	10	1	3	5	7	9
$5i \bmod 11$	5	10	4	9	3	8	2	7	1	6

Note that every number from $\{1, 2, \ldots, p\}$ appears (once and only once) in the $\{2i \bmod 11\}$ and $\{5i \bmod 11\}$ rows of this table—exactly as desired. That is,

$$\{1, 2, 3, \ldots, 10\} \equiv_{11} \{2, 4, 6, \ldots, 20\} \equiv_{11} \{5, 10, 15, \ldots, 50\}.$$

We can also observe examples of this result in the multiplication table for $\mathbb{Z}_7$. (See Figure 7.18.) In that table, every (nonzero) row $\{a, 2a, 3a, 4a, 5a, 6a\}$ contains all six numbers $\{1, 2, 3, 4, 5, 6\}$, in some order, in the six nonzero columns.

Proof of Lemma 7.27. Consider any prime p, and any nonzero $a \in \mathbb{Z}_p$. We must prove that $\{a, 2a, \ldots, (p-1)a\} \equiv_p \{1, 2, \ldots, p-1\}$.

We will first argue that the set $\{1 \cdot a \bmod p, 2 \cdot a \bmod p, \ldots, (p-1) \cdot a \bmod p\}$ contains no duplicates—that is, the value of $i \cdot a \bmod p$ is different for every i. Let $i, j \in \{1, 2, \ldots, p-1\}$ be arbitrary. We will show that $ia \equiv_p ja$ implies that $i = j$, which establishes this first claim. Suppose that $ia \equiv_p ja$. Then, multiplying both sides by a^{-1}, we have that $iaa^{-1} \equiv_p jaa^{-1}$, which immediately yields $i \equiv_p j$ because $a \cdot a^{-1} \equiv_p 1$. (Note that, because p is prime, by Corollary 7.25, we know that a^{-1} exists in $\mathbb{Z}_p$.) Therefore, for any $i, j \in \{1, 2, \ldots, 1-p\}$, if $i \neq j$ then $ai \not\equiv_p aj$.

We now need only show that $ia \bmod p \neq 0$ for any $i > 0$. But that fact is straightforward to see: $ia \bmod p = 0$ if and only if $p \mid ia$, but p is prime and $i < p$ and $a < p$, so p cannot divide ia. (See Exercise 7.49.) □

The theorem itself, and a proof

With this preliminary result in hand, we turn to Fermat's Little Theorem itself:

Theorem 7.28: Fermat's Little Theorem.
Let p be prime, and let $a \in \mathbb{Z}_p$ where $a \neq 0$. Then $a^{p-1} \equiv_p 1$.

As with the previous lemma, we'll start with a few examples of this claim, and then give a proof of the general result. (While this property admittedly might seem a bit mysterious, it turns out to follow fairly closely from Lemma 7.27, as we'll see.)

Example 7.23: Some examples of Fermat's Little Theorem.
Here are a few examples, for the prime numbers 7 and 19:

$$\begin{aligned} 2^6 \bmod 7 &= 64 \bmod 7 &&= (7 \cdot 9 + 1) \bmod 7 = 1 \\ 3^6 \bmod 7 &= 729 \bmod 7 &&= (104 \cdot 7 + 1) \bmod 7 = 1 \\ 4^{18} \bmod 19 &= 68719476736 \bmod 19 &&= (3616814565 \cdot 19 + 1) \bmod 19 = 1. \end{aligned}$$

We'll now give a proof of the theorem, making heavy use of Lemma 7.27:

Proof of Fermat's Little Theorem (Theorem 7.28). Lemma 7.27 says that $\{1, 2, \ldots, p-1\}$ is precisely the same set as $\{1 \cdot a \bmod p, 2 \cdot a \bmod p, \ldots, (p-1) \cdot a \bmod p\}$, so they have the same product:

$$1 \cdot 2 \cdot 3 \cdots (p-1) \equiv_p 1a \cdot 2a \cdot 3a \cdots (p-1)a. \tag{1}$$

Because p is prime, Corollary 7.25 implies that the multiplicative inverses $1^{-1}, 2^{-1}, \ldots, (p-1)^{-1}$ all exist in $\mathbb{Z}_p$. Multiplying both sides of (1) by the product of all $p-1$ of these multiplicative inverses—that is, multiplying by $1^{-1} \cdot 2^{-1} \cdot \cdots \cdot (p-1)^{-1}$—we have

$$1 \cdot 2 \cdot 3 \cdots (p-1) \cdot 1^{-1} \cdot 2^{-1} \cdots (p-1)^{-1} \equiv_p 1a \cdot 2a \cdot 3a \cdots (p-1)a \cdot 1^{-1} \cdot 2^{-1} \cdots (p-1)^{-1}. \tag{2}$$

Rearranging the left-hand side of (2) and replacing $b \cdot b^{-1}$ by 1 for each $b \in \{1, \ldots, p-1\}$, we simply get 1:

$$1 \equiv_p 1a \cdot 2a \cdot 3a \cdots (p-1)a \cdot \cdot 1^{-1} \cdot 2^{-1} \cdots (p-1)^{-1}. \tag{3}$$

Rearranging the right-hand side of (3) and again replacing each $b \cdot b^{-1}$ by 1, we are left only with $p-1$ copies of a:

$$1 \equiv_p a^{p-1},$$

which is exactly what we had to show. □

Note that Fermat's Little Theorem is an implication, *not* an equivalence. It states that *if* p is prime, *then* for every $a \in \{1, \ldots, p-1\}$—that is, for every p relatively prime to n—we have $a^{p-1} \equiv_p 1$. The converse does not always hold: if $a^{n-1} \equiv_n 1$ for every $a \in \mathbb{Z}_n$ that's relatively prime to n, *we cannot conclude that n is prime.* For example, $a^{560} \equiv_{561} 1$ for every $a \in \{1, 2, \ldots, 560\}$ with $\text{GCD}(a, 561) = 1$—but 561 is not prime! (See Exercise 7.112.) A number like 561, which passes the test in Fermat's Little Theorem but is not prime, is called a *Fermat pseudoprime* or a *Carmichael number.* (The latter name is in honor of Robert Carmichael (1879–1967), an American mathematician who was one of the first to discover these numbers.)

> **Taking it further:** Let $n \geq 2$ be an integer, and suppose that we need to determine whether n is prime. There's a test for primality that's implicitly suggested by Fermat's Little Theorem—for "many" different values of $a \in \mathbb{Z}_n$, test to make sure that $a^{n-1} \bmod n = 1$—but this test sometimes incorrectly identifies composite numbers as prime, because of the Carmichael

numbers. (For speed, we generally test a few randomly chosen values of $a \in \mathbb{Z}_p$ instead of trying many of them—but of course testing *fewer* values of a certainly can't prevent us from incorrectly identifying Carmichael numbers as prime.) However, there are some tests for primality that have a similar spirit but that aren't fooled by certain inputs in this way. See the discussion on p. 7-53 for a description of a randomized algorithm called the *Miller–Rabin test* that checks primality using this approach.

Computer Science Connections

Miller–Rabin Primality Testing

Quick: is 1,073,676,287 prime? What about 1,073,676,289? (Or, a bit harder: figure out whether some given 100-digit number is prime.) Fermat's Little Theorem tells us that $a^{n-1} \equiv_n 1$ for any prime n and any nonzero $a \in \mathbb{Z}_n$. This fact makes the randomized algorithm in Figure 7.20 a tempting way to test primality. Indeed, Fermat's Little Theorem implies that **fermat-isPrime?**(p) returns "probably prime" for any prime p—it never errs by accusing a prime number of being composite. But the opposite error (incorrectly asserting that a composite number is prime) *can* happen. And, unfortunately, that error happens frequently for particular values of n—specifically, for Carmichael numbers. (See the discussion after Theorem 7.28, and Exercise 7.112.) For example, $n = 118{,}901{,}521$ is composite, but the only values of a for which $a^{n-1} \not\equiv_n 1$ are multiples of 271, 541, or 811—less than 0.7% of $\{1, 2, \ldots, n-1\}$. (And there are very large Carmichael numbers that have very few prime factors—all of which are big numbers—which cause **fermat-isPrime?** to have an extremely high error rate.)

fermat-isPrime?(n, k):

Input: n is a candidate prime number; k is a "certainty parameter" telling us how many tests to perform before giving up and reporting n as prime.

```
1 repeat
2     choose a ∈ {1, 2, ..., n − 1} randomly
3 until a^{n−1} ≢_n 1 or we've tried k times
4 return "probably prime" if every a^{n−1} ≡_n 1; else return "composite"
```

Figure 7.20 A primality tester based on Fermat's Little Theorem.

In the late 1970s and early 1980s, two computer scientists, Gary Miller and Michael Rabin, developed a new primality test using modular arithmetic that doesn't get fooled for any particular input integer. (Miller's original algorithm [89] is nonrandom but relies on a still-unproven mathematical assumption; Rabin [105] later modified the test to remove the assumption—but at the cost of making it random instead.) The *Miller–Rabin primality test* in Figure 7.21 is based on the following fact (see Exercise 7.51): *if p is prime, then $a^2 \equiv_p 1$ if and only if a mod $p \in \{1, p-1\}$*. Or, taking the contrapositive,

miller-rabin-isPrime?(n, k):

Input: n is a candidate prime number; k is a "certainty parameter"

```
1 write n − 1 as 2^r d for an odd number d
2 while we've done fewer than k tests:
3     choose a random a ∈ {1, ..., n − 1}
4     σ := ⟨a^d, a^{2d}, a^{4d}, a^{8d}, ..., a^{2^r d}⟩ mod n.
5     if σ ≠ ⟨..., 1⟩ or if σ = ⟨..., x, 1, ...⟩ for any x ∉ {1, n − 1} then
6         return "composite"
7 return "probably prime"
```

Figure 7.21 Miller–Rabin primality test.

$$\text{if } a^2 \equiv_n 1 \text{ with } a \bmod p \notin \{1, n-1\}, \text{ then } n \text{ is not prime.} \tag{2}$$

The idea of Miller–Rabin is to look for an $a \in \mathbb{Z}_n$ satisfying (2). Consider a candidate prime $n \geq 3$. Thus n is odd, so $n-1$ is even, and we can write $n - 1 = 2^r d$, where d is an odd number and $r \geq 1$. Define the sequence

$$\left\langle a^d, \underbrace{a^{2d}}_{=(a^d)^2}, \underbrace{a^{4d}}_{=(a^{2d})^2}, \ldots \underbrace{a^{2^r d}}_{=(a^{2^{r-1}d})^2} \right\rangle \tag{3}$$

with each entry taken modulo n. Fermat's Little Theorem says that n is not prime if $a^{n-1} \not\equiv_n 1$, so if (3) ends with anything other

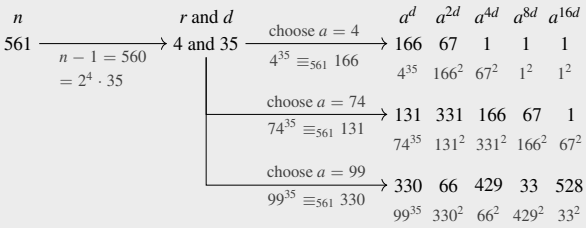


Figure 7.22 A few examples of Miller–Rabin for $n = 561$. Both $a = 4$ (because $67^2 \equiv_{561} 1$ even though $67 \not\equiv_{561} 1$ and $67 \not\equiv_{561} 560$) and $a = 99$ (because $99^{560} \not\equiv_{561} 1$) demonstrate that 561 is not prime.

than 1 we know that n is not prime. And if
there's a 1 that appears immediately after an entry x where $x \bmod n \notin \{1, n-1\}$ in (3), then we also know that n is not prime: $x^2 \equiv_n 1$ but $x \bmod n \notin \{1, n-1\}$, so (2) says that n is not prime. (See Figure 7.22 for an example.) The key fact, which we won't prove here, is that *many different values of $a \in \mathbb{Z}_n$ result in one of these two violations:*

Fact: If n is not prime, then for at least $\frac{n-1}{2}$ different nonzero values of $a \in \mathbb{Z}_n$, the sequence (3) contains a 1 following an entry $x \notin \{1, n-1\}$ or the sequence (3) doesn't end with 1. (For a proof, see [33].)

This fact then allows us to test for n's primality by trying k different randomly chosen values of a; the probability that every one of these tests fails when n is not prime is at most $1/2^k$.

EXERCISES

7.80 Following Example 7.17, identify the numbers that are half of *every* element in $\mathbb{Z}_9$. (That is, for each $a \in \mathbb{Z}_9$, find $b \in \mathbb{Z}_9$ such that $2b = a$.)

We talked extensively in this section about multiplicative inverses, but there can be inverses for other operations, too. The next few exercises explore the additive inverse *in $\mathbb{Z}_n$. Notice that the additive identity in $\mathbb{Z}_n$ is 0: for any $a \in \mathbb{Z}_n$, we have $a+0 \equiv_n 0+a \equiv_n a$. The* additive inverse *of $a \in \mathbb{Z}_n$ is typically denoted $-a$.*

7.81 Give an algorithm to find the additive inverse of any $a \in \mathbb{Z}_n$. (Be careful: the additive inverse of a has to be a value from $\mathbb{Z}_n$, so you can't just say that 3's additive inverse is negative 3!)

7.82 Using your solution to Exercise 7.81, prove that, for any $a \in \mathbb{Z}_n$, we have $-(-a) \equiv_n a$.

7.83 Using your solution to Exercise 7.81, prove that, for any $a, b \in \mathbb{Z}_n$, we have $a \cdot (-b) \equiv_n (-a) \cdot b$.

7.84 Using your solution to Exercise 7.81, prove that, for any $a, b \in \mathbb{Z}_n$, we have $a \cdot b \equiv_n (-a) \cdot (-b)$.

In regular arithmetic, for a number $x \in \mathbb{R}$, a square root *of x is a number y such that $y^2 = x$. If $x = 0$, there's only one such y, namely $y = 0$. If $x < 0$, there's no such y. If $x > 0$, there are two such values y (one positive and one negative).* (Hint for the next two exercises: think about Exercise 7.81.)

7.85 Prove or disprove: for any $n \geq 2$, there exists *one and only one* $b \in \mathbb{Z}_n$ such that $b^2 \equiv_n 0$.

7.86 Prove or disprove: for any $n \geq 2$, and for any $a \in \mathbb{Z}_n$ with $a \neq 0$, there is *not* exactly one $b \in \mathbb{Z}_n$ such that $b^2 \equiv_n a$.

7.87 Using paper and pencil (and brute-force calculation), compute 4^{-1} in $\mathbb{Z}_{11}$ (or state that the inverse doesn't exist).

7.88 Repeat for 7^{-1} in $\mathbb{Z}_{11}$.

7.89 Repeat for 0^{-1} in $\mathbb{Z}_{11}$.

7.90 Repeat for 5^{-1} in $\mathbb{Z}_{15}$.

7.91 Repeat for 7^{-1} in $\mathbb{Z}_{15}$.

7.92 Repeat for 9^{-1} in $\mathbb{Z}_{15}$.

7.93 Prove that the multiplicative inverse is unique: that is, for arbitrary $n \geq 2$ and $a \in \mathbb{Z}_n$, suppose that $ax \equiv_n 1$ and $ay \equiv_n 1$. Prove that $x \equiv_n y$.

7.94 Write down the full multiplication table (as in Figure 7.18) for $\mathbb{Z}_5$.

7.95 Do the same for $\mathbb{Z}_6$.

7.96 Do the same for $\mathbb{Z}_8$.

7.97 Prove or disprove: for arbitrary $n \geq 2$, $(n-1)^{-1} = n-1$ in $\mathbb{Z}_n$.

7.98 Prove that $(a^{-1})^{-1} = a$ for any $n \geq 2$ and $a \in \mathbb{Z}_n$: that is, prove that a is the multiplicative inverse of the multiplicative inverse of a.

7.99 Prove that, for any $n \geq 2$ and $a \in \mathbb{Z}_n$, there exists $x \in \mathbb{Z}$ with $ax \equiv_n 1$ if and only if there exists $y \in \mathbb{Z}_n$ with $ay \equiv_n 1$.

7.100 Suppose that the multiplicative inverse a^{-1} exists in $\mathbb{Z}_n$. Let $k \in \mathbb{Z}_n$ be any exponent. Prove that a^k has a multiplicative inverse in $\mathbb{Z}_n$, and, in particular, prove that the multiplicative inverse of a^k is the kth power of the multiplicative inverse of a. (That is, prove that $(a^k)^{-1} \equiv_n (a^{-1})^k$.)

7.101 *Using paper and pencil* and the algorithm based on the Extended Euclidean algorithm, compute 17^{-1} in $\mathbb{Z}_{23}$ (or explain why it doesn't exist). (See Figure 7.23 for a reminder.)

7.102 Do the same for 7^{-1} in $\mathbb{Z}_{25}$.

7.103 Do the same for 9^{-1} in $\mathbb{Z}_{33}$.

7.104 *(programming required.)* Implement **inverse**(a, n) from Figure 7.19a in a language of your choice.

extended-Euclid(n, m):
Input: positive integers n and $m \geq n$.
Output: $x, y, r \in \mathbb{Z}$ where $\text{GCD}(n, m) = r = xn + ym$
1 **if** $m \bmod n = 0$ **then**
2 **return** $1, 0, n$ // $1 \cdot n + 0 \cdot m = n = \text{GCD}(n, m)$
3 **else**
4 $x, y, r :=$ **extended-Euclid**$(m \bmod n, n)$
5 **return** $y - \left\lfloor \frac{m}{n} \right\rfloor \cdot x, x, r$

inverse(a, n):
Input: $a \in \mathbb{Z}_n$ and $n \geq 2$
Output: a^{-1} in $\mathbb{Z}_n$, if it exists
1 $x, y, d :=$ **extended-Euclid**(a, n)
2 **if** $d = 1$ **then**
3 **return** $x \bmod n$ // $xa + yn = 1$, so $xa \equiv_n 1$.
4 **else**
5 **return** "no inverse for a exists in $\mathbb{Z}_n$."

Figure 7.23 A reminder of two algorithms.

7.105 Prove or disprove the converse of Corollary 7.25: if n is composite, then there exists $a \in \mathbb{Z}_n$ (with $a \neq 0$) that does not have a multiplicative inverse in $\mathbb{Z}_n$.

7.106 Let p be an arbitrary prime number. What value does the quantity $2^{p+1} \bmod p$ have? Be as specific as you can. Explain.

7.107 It turns out that $247^{248} \bmod 249 = 4$. From this, you can conclude at least one of following: 247 is not prime; 247 is prime; 249 is not prime; or 249 is prime. Which one(s)? Explain.

7.108 Reprove the general version of the Chinese Remainder Theorem with single constructive argument, as in the 2-congruence case, instead of using induction. Namely, assume $n_1, n_2, \ldots, n_k$ are pairwise relatively prime, and let $a_i \in \mathbb{Z}_{n_i}$. Let $N = \prod_{i=1}^{k} n_i$. Let $N_i = N/n_i$ (more precisely, let N_i be the product of all n_js *except* n_i) and let d_i be the multiplicative inverse of N_i in $\mathbb{Z}_{n_i}$. Prove that $x = \sum_{i=1}^{k} a_i N_i d_i$ satisfies the congruence $x \bmod n_i = a_i$ for all i.

The totient function $\varphi : \mathbb{Z}^{\geq 1} \to \mathbb{Z}^{\geq 0}$*, sometimes called* Euler's totient function *after the 18th-century Swiss mathematician Leonhard Euler, is defined as*

$$\varphi(n) = \textit{the number of } k \textit{ such that } 1 \leq k \leq n \textit{ such that } k \textit{ and } n \textit{ have no common divisors.}$$

For example, $\varphi(6) = 2$ *because* 1 *and* 5 *have no common divisors with* 6 *(but all of* $\{2, 3, 4, 6\}$ *do share a common divisor with* 6*). There's a generalization of Fermat's Little Theorem, sometimes called the* Fermat–Euler Theorem *or* Euler's Theorem*, that states the following: if* a *and* n *are relatively prime, then* $a^{\varphi(n)} \equiv_n 1$.

7.109 Assuming the Fermat–Euler theorem, argue that Fermat's Little Theorem holds.

7.110 Assuming the Fermat–Euler theorem, argue that a^{-1} in $\mathbb{Z}_n$ is $a^{\varphi(n)-1} \bmod n$, for any $a \in \mathbb{Z}_n$ that is relatively prime to n. Verify the claim for the multiplicative inverses of $a \in \{7, 17, 31\}$ in $\mathbb{Z}_{60}$.

7.111 *(programming required.)* Implicitly, the Fermat–Euler theorem gives a different way to compute the multiplicative inverse of a in $\mathbb{Z}_n$: (1) compute $\varphi(n)$ [say by brute force, though there are somewhat faster ways—see Exercises 9.37–9.39]; and then (2) compute $a^{\varphi(n)-1} \bmod n$ [perhaps using repeated squaring; see Figure 7.6]. Implement this algorithm to compute a^{-1} in $\mathbb{Z}_n$ in a programming language of your choice.

Recall that a Carmichael number *is a composite number that passes the (bogus) primality test suggested by Fermat's Little Theorem. In other words, a Carmichael number* n *is an integer that is composite but such that, for any* $a \in \mathbb{Z}_n$ *that's relatively prime to* n*, we have* $a^{n-1} \bmod n = 1$.

7.112 *(programming required.)* Write a program to verify that 561 is (a) not prime, but (b) satisfies $a^{560} \bmod 561 = 1$ for every $a \in \{1, \ldots, 560\}$ that's relatively prime to 561. (That is, verify that 561 is a Carmichael number.)

7.113 Suppose n is a *composite* integer. Argue that there exists at least one integer $a \in \{1, 2, \ldots, n-1\}$ such that $a^{n-1} \not\equiv_n 1$. (In other words, there's always *at least one* nonzero $a \in \mathbb{Z}_n$ with $a^{n-1} \not\equiv_n 1$ when n is composite. Thus, although the probability of error in **fermat-isPrime?** from p. 7-53 may be very high for particular composite integers n, the probability of success is nonzero, at least!)

The following theorem is due to Alwin Korselt, from 1899: an integer n is a Carmichael number if and only if n is composite, squarefree, and for all prime numbers p that divide n, we have that $p - 1 \mid n - 1$. (An integer n is squarefree *if there is no integer $d \geq 2$ such that $d^2 \mid n$.)*

7.114 *(programming required.)* Use Korselt's theorem (and a program) to find all Carmichael numbers less than 10,000.

7.115 Use Korselt's theorem to prove that all Carmichael numbers are odd.

7.116 *(programming required.)* Implement the Miller–Rabin primality test (see p. 7-53) in a language of your choice.

7.5 Cryptography

> Three may keep a Secret, if two of them are dead.
>
> Benjamin Franklin (1706–1790)
> *Poor Richard's Almanack* (1735)

In the rest of this chapter, we will make use of the number-theoretic machinery that we've now developed to explore *cryptography.* Imagine that a sender, named *Alice,* is trying to send a secret message to a receiver, named *Bob.* (In a tradition that began with the paper introducing the RSA cryptosystem [106], cryptographic systems are usually described using an imagined crew of people whose names start with consecutive letters of the alphabet. We'll stick with these traditional names: Alice, Bob, Carol, etc.) The goal of cryptography is to ensure that the message itself is kept secret even if an eavesdropper—named *Eve*—overhears the transmission to Bob.

To achieve this goal, Alice does not directly transmit the message m that she wishes to send to Bob; instead, she *encrypts* m in some way. The resulting encrypted message c is what's transmitted to Bob. (The original message m is called *plaintext*; the encrypted message c that's sent to Bob is called the *ciphertext*.) Bob then *decrypts* c to recover the original message m. A diagram of the basic structure of a cryptographic system is shown in Figure 7.24.

The two obvious crucial properties of a cryptographic system are that (i) Bob can compute m from c, and (ii) Eve cannot compute m from c. (Of course, to make (i) and (ii) true simultaneously, it will have to be the case that Bob has some information that Eve doesn't have—otherwise the task would be impossible!)

One-time pads

The simplest idea for a cryptographic system is for Alice and Bob to agree on a *shared secret key* that they will use as the basis for their communication. The easiest implementation of this idea is what's called a *one-time pad.* (The *pad* in the name comes from spycraft—spies might carry physical pads of paper, where

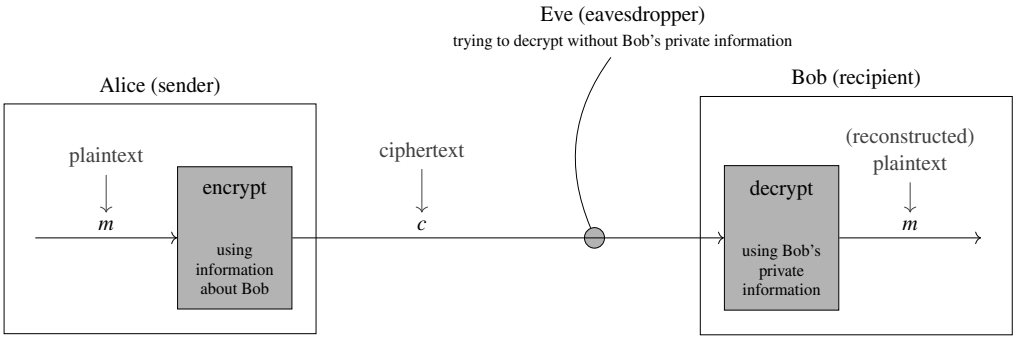


Figure 7.24 The outline of a cryptographic system.

each sheet has a fresh secret key written on it. The *one-time* in the name comes from the fact that this system is secure only if the same key is never reused, as we'll discuss.)

Here's how a one-time pad works. Alice and Bob agree in advance on an integer n, denoting the length of the message that they would like to communicate. They also agree in advance on a secret bitstring $k \in \{0,1\}^n$, where each bit $k_i \in \{0,1\}$ is chosen independently and uniformly—so that every one of the 2^n different n-bit strings has a $\frac{1}{2^n}$ chance of being chosen as k. To encrypt a plaintext message $m \in \{0,1\}^n$, Alice computes the *bitwise exclusive or* of m and k—in other words, the ith bit of the ciphertext is $m_i \oplus k_i$. To decrypt the ciphertext $c \in \{0,1\}^n$, Bob computes the bitwise XOR of c and k. Here's a small example:

Example 7.24: A One-Time Pad.
Alice and Bob agree (in advance) on the secret key $k = 10111000$.

Suppose that Alice decides that she wishes to transmit the message $m = 01101110$. To do so, Alice finds the bitwise XOR of m and k, which is $c = 11010110$. (See Figure 7.25a.)

To decrypt the ciphertext $c = 11010110$, Bob finds the bitwise XOR of c and k. (Again see Figure 7.25a.) Observe that $c \oplus k = 01101110$ is indeed precisely $m = 01101110$, as desired.

The reason that Bob can decrypt the ciphertext to recover the original message m is that, for any bits a and b, it's the case that $(a \oplus b) \oplus b = a$. (See Figure 7.25b.) The fact that Eve *cannot* recover m from c relies on the fact that, for any message m and every ciphertext c, there is precisely one secret key k such that $m \oplus k = c$. (So Eve is just as likely to see a particular ciphertext *regardless of what the message is,* and therefore she gains no information about m by seeing c. See Exercise 7.118.) Thus the one-time pad is perfectly secure as a cryptographic system—if Alice and Bob only use it once! If Alice and Bob reuse the same key to exchange many different messages, then Eve can use frequency analysis to get a handle on the key, and therefore can begin to decode the allegedly secret messages. (See Exercises 10.73–10.77 or Exercise 7.119.)

Taking it further: One of the earliest encryption schemes is now known as a *Caesar Cipher,* after Julius Caesar, who used it in his correspondence. It can be understood as a cryptographic system that uses a one-time pad more than once. The Caesar cipher works as follows. The sender and receiver agree on a *shift* x, an integer, as their secret key. The ith letter in the alphabet (from

m	0	1	1	0	1	1	1	0
k	1	0	1	1	1	0	0	0
$c = m \oplus k$	1	1	0	1	0	1	1	0
c	1	1	0	1	0	1	1	0
k	1	0	1	1	1	0	0	0
$c \oplus k$	0	1	1	0	1	1	1	0

(a) An example of encryption and decryption with a one-time pad. (See Example 7.24.)

a	b	$a \oplus b$	$(a \oplus b) \oplus b$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

(b) The truth table for $(a \oplus b) \oplus b = a$. For any message bit a and any key bit b, the decryption of the encryption is equal to the original plaintext.

Figure 7.25 The one-time pad.

A = 0 through Z = 25) will be shifted forward by x positions in the alphabet. The shift "wraps around," so that we encode letter i as letter $(i+x) \bmod 26$. For example, if $x = 3$ then A→D, L→O, Y→B, etc. To send a text message m consisting of multiple letters from the alphabet, the same shift is applied to each letter. (For convenience, we'll leave nonalphabetic characters unchanged.) For example, the ciphertext `XF BSF EJTDPWFSFE; GMFF BU PODF!` was generated with the shift $x = 1$ from the message `WE ARE DISCOVERED; FLEE AT ONCE!`. Because we've reused the same shift x for each letter of the message, the Caesar Cipher is susceptible to being broken based on frequency analysis. (In the `XF BSF EJTDPWFSFE; GMFF BU PODF!` example, F is by far the most common letter in the ciphertext—and E is by far the most common letter in English text. From these two facts, you might infer that $x = 1$ is the most probable secret key. See Exercise 7.119.)

Millennia later, the Enigma machines, the encryption system used by the Germans during World War II, was—as with Caesar—a substitution cipher, but one where the shift changed with each letter. (But not in completely unpredictable ways, as in a one-time pad!) See p. 9-75 for more.

Public-key cryptography

In addition to being single-use-only, there's another strange thing about the one-time pad: if Alice and Bob are somehow able to communicate an n-bit string securely—as they must to share the secret key k—it doesn't seem particularly impressive that they can then communicate the n-bit string m securely.

Public-key cryptography is an idea to get around this oddity. Here is the idea, in a nutshell. Every participant will have a *public key* and a *private* (or *secret) key,* which will somehow be related to the public key. A user's public key is completely public—for example, posted on the web. If Alice wishes to send a message m to Bob, then Alice will (somehow!) encrypt her message to Bob using Bob's public key, producing ciphertext c. Bob, who of course knows Bob's secret key, can decrypt c to reconstruct m; Eve, not knowing Bob's secret key, cannot decrypt c.

This idea sounds a little absurd, but we will be able to make it work. Or, at least, we will make it work *on the assumption that Eve has only limited computational power*—and on the assumption that certain computational problems, like factoring large numbers, require a lot of computational power to solve. (For example, Bob's secret key cannot be easily computable from Bob's public key—otherwise Eve could easily figure out Bob's secret key and then run whatever decryption algorithm Bob uses!)

7.5.1 The RSA Cryptosystem

The basic idea of public-key cryptography was discussed in abstract terms in the 1970s—especially by Whitfield Diffie, Martin Hellman, and Ralph Merkle—and, after some significant contributions by a number of researchers, a cryptosystem successfully implementing public-key cryptography was discovered by Ron Rivest, Adi Shamir, and Leonard Adleman [106]. The *RSA cryptosystem,* named after the first initials of their three last names, is one of the most famous, and widely used, cryptographic protocols today. The previous sections of this chapter will serve as the building blocks for the RSA system, which we'll explore in the rest of this section.

Taking it further: The RSA cryptosystem is named after its three 1978 discoverers, and the Turing Award—the highest honor in computer science, roughly equivalent to the Nobel Prize of computer science—was conferred on Rivest, Shamir, and Adleman in 2002 for this discovery. But there is also a "shadow history" of the advances in cryptography made in the second half of the 20th century. The British government's signal intelligence agency, called Government Communications Headquarters (GCHQ), had been working to solve precisely the same set of research questions about cryptography as academic researchers like R., S., and A. (GCHQ was perhaps best known for its success in World War II, in breaking the Enigma Code of the German military; see p. 9-75 for more discussion.) And it turned out that several British cryptographers at GCHQ—Clifford Cocks, James Ellis, and Malcolm Williamson—had discovered the RSA protocol several years *before* 1978. But their discovery was classified by the British government, and thus we call this protocol "RSA" instead of "CEW." See the excellent book by Simon Singh for more on the history of cryptography, including both the published and classified advances in cryptographic systems [118].

Also see the discussion on p. 7-67 of the *Diffie–Hellman key exchange protocol,* one of the first (published) modern breakthroughs in cryptography, which allows Alice and Bob to solve another apparently impossible problem: exchanging secret information while communicating only over an insecure channel.

In RSA, as for any public-key cryptosystem, we must define three algorithmic components. (These three algorithms for the RSA cryptosystem are shown in Figure 7.26; an overview of the system is shown in Figure 7.27.) They are:

- *key generation:* how do Alice and Bob construct their public/private keypairs?
- *encryption:* when Alice wishes to send a message to Bob, how does she encode it?
- *decryption:* when Bob receives ciphertext from Alice, how does he decode it?

The core idea of RSA is the following. (The details of the protocols are in Figure 7.26.) To encrypt a numerical message m for Bob, Alice will compute $c := m^e \bmod n$, where Bob's public key is $\langle e, n \rangle$. To decrypt the ciphertext c that he receives, Bob will compute $c^d \bmod n$, where Bob's private key is $\langle d, n \rangle$. (Of course, there's an important relationship among the quantities e, d, and n!)

Taking it further: It may seem strange that n is part of *both* Bob's secret key *and* Bob's public key—it's usually done this way for symmetry, but also to support *digital signatures.* When Alice sends Bob a message, she can encrypt it *using her own secret key*; Bob can then decrypt the message *using Alice's public key* to verify that Alice was indeed the person who sent the message.

An example of RSA key generation, encryption, and decryption

Later we will prove that the message that Bob decrypts is always the same as the message that Alice originally sent. But we'll start with an example. First, Bob generates a public and private key, using the protocol in Figure 7.26. (All three phases can be implemented efficiently, using techniques from this chapter; see Exercises 7.131–7.134.)

Example 7.25: Generating an RSA keypair for Bob.
For good security properties, we'd want to pick seriously large prime numbers p and q, but to make the computation easier to see we'll choose very small primes.

(1) Suppose we choose the "large" primes $p = 13$ and $q = 17$. Then $n := 13 \cdot 17 = 221$.

Key Generation:
(1) Bob chooses two large primes, p and q, and defines $n := pq$.
(2) Bob chooses $e \neq 1$ such that e and $(p-1)(q-1)$ are relatively prime.
(3) Bob computes $d := e^{-1}$ modulo $(p-1)(q-1)$.
(4) Bob publishes $\langle e, n \rangle$ as his public key; Bob's secret key is $\langle d, n \rangle$.

Encryption: If Alice wants to send message m to Bob,
(1) Alice finds Bob's public key, say $\langle e_{\text{Bob}}, n_{\text{Bob}} \rangle$, as he published it.
(2) To send message $m \in \{0, \ldots, n_{\text{Bob}} - 1\}$, Alice computes $c := m^{e_{\text{Bob}}} \bmod n_{\text{Bob}}$.
(3) Alice transmits c to Bob.

Decryption: When Bob receives ciphertext c,
(1) Bob computes $m := c^{d_{\text{Bob}}} \bmod n_{\text{Bob}}$, where $\langle d_{\text{Bob}}, n_{\text{Bob}} \rangle$ is Bob's secret key.

Figure 7.26 The RSA cryptosystem: key generation, encryption, and decryption.

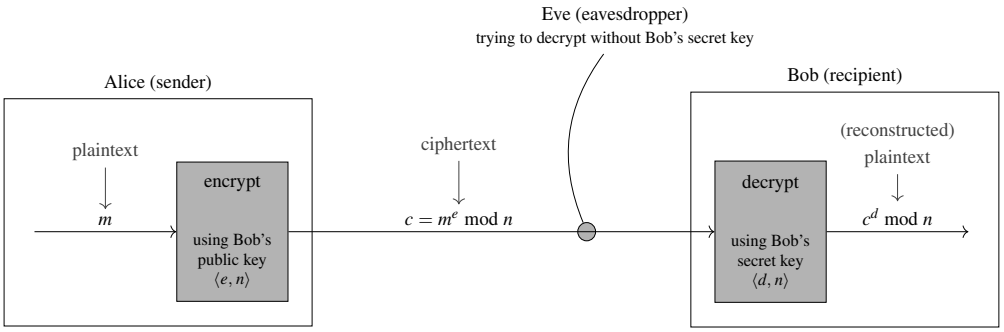


Figure 7.27 A schematic of the RSA cryptosystem, where $n = pq$ and $de \equiv_{(p-1)(q-1)} 1$, for prime numbers p and q.

(2) We now must choose a value of $e \neq 1$ that is relatively prime to $(p-1)(q-1) = 12 \cdot 16 = 192$. Note that $\text{GCD}(2, 192) = 2 \neq 1$, so $e = 2$ fails. Similarly $\text{GCD}(3, 192) = 3$ and $\text{GCD}(4, 192) = 4$. But $\text{GCD}(5, 192) = 1$. We pick $e := 5$.

(3) We now compute $d := \textbf{inverse}(e, (p-1)(q-1))$—that is, $d := e^{-1}$ in $\mathbb{Z}_{(p-1)(q-1)}$. Tracing the execution of $\textbf{inverse}(5, 192)$ just as in Example 7.21 reveals that $\textbf{extended-Euclid}(5, 192)$ returns $\langle 77, -2, 1 \rangle$, and thus that $\textbf{inverse}(5, 192)$ returns $77 \bmod 192 = 77$. (Indeed, $5 \cdot 77 = 385 = 192 \cdot 2 + 1$, so $5 \cdot 77 \equiv_{192} 1$.) So we set $d := 77$.

(4) Thus Bob's public key is $\langle e, n \rangle = \langle 5, 221 \rangle$, and Bob's secret key is $\langle d, n \rangle = \langle 77, 221 \rangle$.

Bob now publishes his public key somewhere, keeping his secret key to himself. If Alice now wishes to send a message to Bob, she uses his public key, as follows:

Example 7.26: Encrypting a message with RSA.
To send message $m = 202$ to Bob, whose public key is $\langle e, n \rangle = \langle 5, 221 \rangle$, Alice computes

$$m^e \bmod n = 202^5 \bmod 221 = 336{,}323{,}216{,}032 \bmod 221 = 206.$$

Thus she sends Bob the ciphertext $c := 206$.

When Bob receives an encrypted message, he uses his secret key to decrypt it:

Example 7.27: Decrypting a message with RSA.
When Bob, whose secret key is $\langle d, n\rangle = \langle 77, 221\rangle$, receives the ciphertext $c = 206$ from Alice, he decrypts it as

$$c^d \bmod n = 206^{77} \bmod 221.$$

Computing $206^{77} \bmod 221$ by hand is a bit tedious, but we can calculate it with "repeated squaring" (using the fact that $b^{2k} \bmod n = (b^2 \bmod n)^k \bmod n$ and $b^{2k+1} \bmod n = b \cdot (b^{2k} \bmod n) \bmod n$; see Exercises 7.23–7.25):

$$\begin{aligned}
206^{77} \bmod 221 &= 206 \cdot (206^2 \bmod 221)^{38} \bmod 221 && 206^2 \bmod 221 = 4\\
&= 206 \cdot (4^2 \bmod 221)^{19} \bmod 221 && 4^2 \bmod 221 = 16\\
&= 206 \cdot 16 \cdot (16^2 \bmod 221)^{9} \bmod 221 && 16^2 \bmod 221 = 35\\
&= 206 \cdot 16 \cdot 35 \cdot (35^2 \bmod 221)^{4} \bmod 221 && 35^2 \bmod 221 = 120\\
&= 206 \cdot 16 \cdot 35 \cdot (120^2 \bmod 221)^{2} \bmod 221 && 120^2 \bmod 221 = 35\\
&= 206 \cdot 16 \cdot 35 \cdot (35^2 \bmod 221) \bmod 221 && 35^2 \bmod 221 = 120\\
&= 206 \cdot 16 \cdot 35 \cdot 120 \bmod 221 && 206 \cdot 16 \cdot 35 \cdot 120 = 13{,}843{,}200\\
&= 202.
\end{aligned}$$

Thus Bob decrypts the ciphertext 206 as $202 = 206^{77} \bmod 221$. Indeed, then, the message that Bob receives is precisely 202—the same message that Alice sent!

We've now illustrated the full RSA protocol: generating a key, and encrypting and decrypting a message. Here's one more chance to work through the full pipeline:

Example 7.28: RSA, again, from end to end.
Bob generates a public/private keypair using the primes $p = 11$ and $q = 13$, choosing the smallest valid value of e. You encrypt the message 95 to send to Bob (using his generated public key). What ciphertext do you send to Bob?

Solution. For $\langle p, q\rangle = \langle 11, 13\rangle$, we have $pq = 143$ and $(p-1)(q-1) = 120$. Because 120 is divisible by 2, 3, 4, 5, and 6 but $\text{GCD}(120, 7) = 1$, we choose $e := 7$. We find $d := \textbf{inverse}(7, 120) = 103$. Then Bob's public key is $\langle e, n\rangle = \langle 7, 143\rangle$ and Bob's private key is $\langle d, n\rangle = \langle 103, 143\rangle$.

To send Bob the message $m = 95$, we compute $m^e \bmod n = 95^7 \bmod 143$, which is 17. Thus the ciphertext is $c := 17$. (Bob would decrypt this ciphertext as $c^d \bmod n = 17^{103} \bmod 143$—which indeed is 95.)

7.5.2 The Correctness of RSA

Examples 7.25–7.27 gave one instance of the RSA cryptosystem working properly, in the sense that **decrypt**(**encrypt**(m)) turned out to be the original message m itself—but, of course, we want this property to be true in general. Let's prove that it is. Before we give the full statement of correctness, we'll prove an intermediate lemma:

Lemma 7.29: Correctness of RSA: decrypting the ciphertext, modulo p or q.
Suppose e, d, p, q, n are all as specified in the RSA key generation protocol—that is, $n = pq$ for primes p and q, and $ed \equiv_{(p-1)(q-1)} 1$. Let $m \in \mathbb{Z}_n$ be any message. Then

$$m' := [(m^e \bmod n)^d \bmod n] \qquad \textit{(the decryption of the encryption of } m\textit{)}$$

satisfies both $m' \equiv_p m$ and $m' \equiv_q m$.

Proof. We'll prove $m' \equiv_p m$; because p and q are symmetric in the definition, the fact that $m' \equiv_q m$ follows by exactly the same logic. Recall that we chose d so that $ed \equiv_{(p-1)(q-1)} 1$, and thus we have $ed = k(p-1)(q-1)+1$ for some integer k. Hence

$$\begin{aligned}
[(m^e \bmod n)^d \bmod n] \bmod p &= (m^{ed} \bmod n) \bmod p && \textit{by Theorem 7.4.4}\\
&= (m^{k(p-1)(q-1)+1} \bmod pq) \bmod p && \textit{by definition of e, d, n, and k}\\
&= m^{k(p-1)(q-1)+1} \bmod p && \textit{by Exercise 7.18}\\
&= [m \cdot m^{k(p-1)(q-1)}] \bmod p && a^{k+1} = a \cdot a^k\\
&= [(m \bmod p) \cdot (m^{k(p-1)(q-1)} \bmod p)] \bmod p && \textit{by Theorem 7.4.3}\\
&= [(m \bmod p) \cdot ((m^{k(q-1)} \bmod p)^{p-1} \bmod p)] \bmod p. && \textit{by Theorem 7.4.4}
\end{aligned}$$

Although it's not completely obvious, we're actually almost done: we've now shown

$$\Big[(m^e \bmod n)^d \bmod n\Big] \bmod p = \Big[(m \bmod p) \cdot \big((m^{k(q-1)} \bmod p)^{p-1} \bmod p\big)\Big] \bmod p. \qquad (*)$$

If only the highlighted portion of the right-hand side of $(*)$ were equal to 1, we'd have shown exactly the desired result, because the right-hand side would then equal $[(m \bmod p) \cdot 1] \bmod p = m \bmod p$—exactly what we had to prove! And the good news is that the highlighted portion of $(*)$ matches the form of Fermat's Little Theorem: the highlighted expression is $a^{p-1} \bmod p$, where $a = m^{k(q-1)} \bmod p$, and Fermat's Little Theorem tells us $a^{p-1} \bmod p = 1$ as long as $a \not\equiv_p 0$—that is, as long as $p \nmid a$. (We'll also have to handle the case when a *is* divisible by p, but we'll be able to do that separately.)

> *Problem-solving tip:* If there's a proof outline that will establish a desired claim except in one or two special cases, then try to "break off" those special cases and handle them separately. Here we handled the "normal" case $a \not\equiv_p 0$ using Fermat's Little Theorem, and broke off the special $a \equiv_p 0$ case and handled it separately.

Here are the two cases:

Case I: $a \equiv_p 0$. Notice that $m^{k(q-1)} \bmod p = 0$ and thus that $p \mid m^{k(q-1)}$. Therefore:

$$\begin{aligned}
[(m^e \bmod n)^d \bmod n] \bmod p &= [(m \bmod p) \cdot a^{p-1} \bmod p] \bmod p && \text{by } (*) \\
&= [(m \bmod p) \cdot 0] \bmod p && \text{by the assumption that } a \equiv_p 0 \\
&= 0 \\
&= m \bmod p,
\end{aligned}$$

where the last equality follows because p is prime and $p \mid m^{k(q-1)}$; thus Exercise 7.48 tells us that $p \mid m$ as well.

Case II: $a \not\equiv_p 0$. Then we can use Fermat's Little Theorem:

$$\begin{aligned}
[(m^e \bmod n)^d \bmod n] \bmod p &= [(m \bmod p) \cdot a^{p-1} \bmod p] \bmod p && \text{by } (*) \\
&= [(m \bmod p) \cdot 1] \bmod p && \text{by Fermat's Little Theorem} \\
&= m \bmod p.
\end{aligned}$$

We've established that $[(m^e \bmod n)^d \bmod n] \bmod p = m \bmod p$ in both cases, and the lemma follows. □

Using Lemma 7.29 to do most of the work, we can now prove the main theorem:

Theorem 7.30: Correctness of RSA.
Suppose that Bob's RSA public key is $\langle e, n \rangle$ and his corresponding private key is $\langle d, n \rangle$. Let $m \in \mathbb{Z}_n$ be any message. Then

$$\textbf{decrypt}_{\text{Bob}}(\textbf{encrypt}_{\text{Bob}}(m)) = m.$$

Proof. Note that $\textbf{decrypt}_{\text{Bob}}(\textbf{encrypt}_{\text{Bob}}(m)) = (m^e \bmod n)^d \bmod n$. By Lemma 7.29,

$$(m^e \bmod n)^d \bmod n \equiv_p m \qquad \text{and} \qquad (m^e \bmod n)^d \bmod n \equiv_q m.$$

By Exercise 7.50, together these facts imply that $(m^e \bmod n)^d \bmod n \equiv_{pq} m$ as well. Because $n = pq$ and $m < n$, therefore $(m^e \bmod n)^d \bmod n = m \bmod n = m$. □

What about Eve?

When Alice encrypts a message m for Bob and transmits the corresponding RSA-encrypted ciphertext, Theorem 7.30 says that Bob is able to decrypt to recover the original message m. What's left to establish is that Eve *cannot* recover m from the information that she knows—namely, from the ciphertext $m^e \bmod n$ and from Bob's public key $\langle e, n \rangle$. (That's the desired security property of the system!)

Unfortunately, not only are we unable to *prove* this property, it's simply not true! Eve *is* able to recover m from the $m^e \bmod n$ and e and n, as follows: she factors n—that is, finds the primes p and q such that

$pq = n$—and then computes d precisely as Bob did when he generated his RSA keys. (And Eve then computes the "secret" message $(m^e \bmod n)^d \bmod n$ precisely as Bob did when he decrypted.)

But the fact that Eve *has the information necessary to recover m* doesn't mean that RSA is doomed: factoring large numbers (particularly those that are the product of two large primes, perhaps) seems to be a computationally difficult problem. Even if you know that $n = 121{,}932{,}625{,}927{,}450{,}033$ it will take you quite a while to find p and q—and the best known algorithms for factoring are not fast enough for Eve.

> **Taking it further:** The crucial property that we're using in RSA is an asymmetry in two "directions" of a problem. Taking two large prime numbers p and q and computing their product $n = pq$ is easy (that is, it can be done quickly, in polylogarithmic time). Taking a number n that happens to be the product of two primes and factoring it into $p \cdot q$ appears to be hard (that is, nobody knows how to do it quickly). Cryptographic systems have been built on a number of different problems with this kind of asymmetry; see a good textbook on cryptography for much, much more—for example, [69, 108]. See these books for a discussion of some of the ways in which "textbook RSA"—what we've described here!—is susceptible to all sorts of attacks. Industrial-level RSA implementations take all sorts of precautions that we haven't even begun to discuss. (It's also worth noting that Eve *is* aware that Alice and Bob are communicating—just not, under the hardness assumptions we're discussing, what it is that they're saying. *Steganography* [from the Greek *stego* hidden/covered; compare that to *kryptos* concealed/secret] seeks to keep secret *the existence of communication* rather than merely *the contents of the communication.* See p. 5-22.)

Notice, though, that Eve could break RSA another way, too: she only needs to find m, and she knows both the ciphertext $c = m^e \bmod n$ and Bob's public key $\langle e, n \rangle$. So Eve could discover m by computing the "eth root of c"—that is, the number x such that $x^e = c$. Unfortunately for Eve, the fact that she has to compute the eth root of $m^e \bmod n$, and not just the eth root of m^e, is crucial; this problem also seems to be computationally difficult. (See Exercise 7.141—though there's some evidence that choosing a small value of e, like $e = 3$, might weaken the security of the system.)

Note, though, that we have *not* proven that Eve is unable to efficiently break RSA encryption—for all we know, a clever student of computational number theory (you!?) will discover an efficient algorithm for factoring large numbers or computing eth roots in $\mathbb{Z}_n$. (Or, for all we know, perhaps someone already has!)

Computer Science Connections

Diffie–Hellman Key Exchange

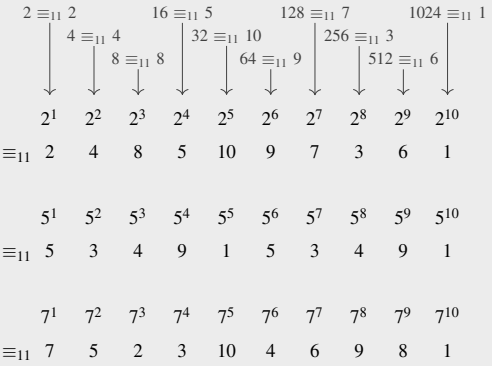


Figure 7.28 Primitive roots of $\mathbb{Z}_{11}$. $\mathbb{Z}_{11}$ has four different primitive roots $\{2, 6, 7, 8\}$, two of which are shown here: the first 10 powers of 2 and 7 (but not 5) in $\mathbb{Z}_{11}$ produce all 10 nonzero elements of $\mathbb{Z}_{11}$.

Suppose that Alice and Bob wish to establish some shared piece of secret information—perhaps to share a key to use in a one-time pad, or to use for some other cryptographic protocol. But the only communication channel available to Alice and Bob is insecure; Eve can listen to all of their communication. This problem is called the *key exchange* problem: *two parties seek to establish a shared secret while they communicate only over an insecure channel.* Like public-key cryptography (as in RSA), this task seems completely impossible—and, also like public-key cryptography, despite its apparent impossibility, this problem was solved in the 1970s. The solution that we'll describe here is called the *Diffie–Hellman key exchange protocol* [38], named after Whitfield Diffie (b. 1944) and Martin Hellman (b. 1945)—winners of the 2015 Turing Award, largely for this contribution.

Let p be prime. The key number-theoretic definition for the Diffie–Hellman protocol is what's called a *primitive root mod p,* which is an element $g \in \mathbb{Z}_p$ such that every nonzero element of $\mathbb{Z}_p$ is equivalent to a power of g. (In other words, g is a primitive root mod p if $\{g^1, g^2, \ldots, g^{p-1}\} \equiv_p \{1, 2, \ldots, p-1\}$.) See Figure 7.28 for some examples. It's a theorem of number theory that every $\mathbb{Z}_p$ for prime p has at least one primitive root.

The Diffie–Hellman protocol is shown in Figure 7.29. Alice and Bob agree on a prime p and a number g that's a primitive root mod p. Alice chooses a secret value $a \in \mathbb{Z}_p$ randomly, computes $A := g^a \bmod p$, and sends A to Bob. Bob chooses a secret value $b \in \mathbb{Z}_p$ randomly, computes $B := g^b \bmod p$, and sends B to Alice. (Note that A and B are sent over the channel, but the values of a and b themselves are never transmitted.) Alice now knows a (she picked it) and $B = g^b \bmod p$ (Bob sent it to her), so she can compute $B^a \bmod p$. Bob knows b (he picked it) and $A = g^a \bmod p$ (Alice sent it to him), so he can compute $A^b \bmod p$.

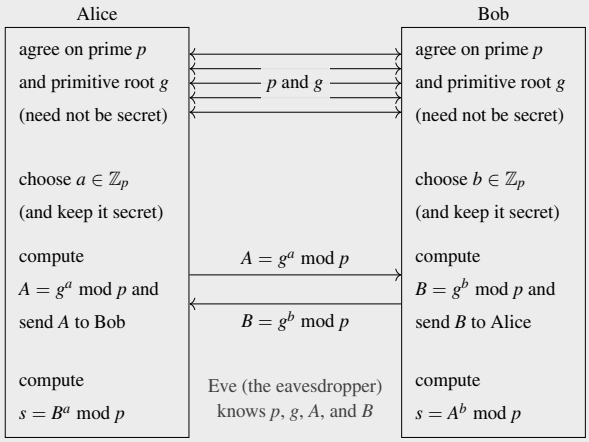


Figure 7.29 The Diffie–Hellman key exchange protocol.

Alice and Bob now have a shared piece of information, namely $g^{ab} \bmod p$, because $A^b \equiv_p (g^a)^b = g^{ab} = (g^b)^a) \equiv_p B^a$. But why is this shared piece of information a secret? Let's look at the protocol from Eve's perspective: she observes the values of p, g, $g^a \bmod p$, and $g^b \bmod p$. But it is generally believed that the problem of computing a from the values of p, g, and $g^a \bmod p$

cannot be solved efficiently. (This problem is called the *discrete logarithm problem:* it's the modular analogy of computing y from the values of x and x^y—that is, computing $\log_x(x^y)$.) The discrete log problem is believed to be difficult (as long as the prime p is of appreciable size), and thus it's believed that Eve cannot feasibly figure out the value $g^{ab} \bmod p$, shared by Alice and Bob. (For much more, see any good book on cryptography, such as [69, 108]. It's worth pointing out that, as we've stated the protocol, Diffie–Hellman is susceptible to a so-called *man-in-the-middle attack:* a malicious party ("Mallory") who has control over the channel can impersonate Bob to Alice, *and* impersonate Alice to Bob. (There are improvements to the protocol that address this issue.) Doing so allows Mallory to intercept, decrypt, and then reencrypt subsequent communications that Alice and Bob thought were secure—and they'd never know that Mallory was involved.)

EXERCISES

7.117 In our encryption/decryption scheme for one-time pads, we used *exclusive or:* plaintext m was encrypted as $m \oplus k$, and ciphertext c was decrypted as $c \oplus k$. Because $(m \oplus k) \oplus k$ is logically equivalent to m, Bob always recovers the original plaintext. But there are actually three other Boolean operators that we could have used instead of $\oplus$—that is, there are three other connectives $\circ$ such that $(m \circ k) \circ k \equiv m$. (See Figure 4.34.) Identify those three other connectives. Why are these three connectives either uninteresting or actively bad choices as alternatives to $\oplus$?

7.118 *(Requires knowledge of probability; see Chapter 10.)* For a one-time pad with an n-bit key, we have

$$\texttt{Pr}\left[\text{ciphertext} = c\right] = \sum_m \left[\texttt{Pr}\left[\text{ciphertext} = c | \text{plaintext} = m\right] \cdot \texttt{Pr}\left[\text{plaintext} = m\right]\right].$$

Prove that the probability that the ciphertext is a particular $c \in \{0,1\}^n$ is precisely $1/2^n$ *for any distribution over plaintext messages.*

7.119 *(programming required.)* As we suggested, one-time pads are secure *if* they're used only once, but using the same key more than once compromises security. I took a (famous) document written in English, and encoded it as follows: I converted each character to an ASCII value (in binary, 00000000 to 11111111), and separated this bitstring into 40-bit chunks. (Each chunk contains 5 characters, with 8 bits each.) I generated a 40-bit key, and encoded each chunk using that key. (That is: I used a one-time pad more than one time!) The encoded document starts like this:

```
1110111111011100100010011010000110111101
1110100011010101111110111010011110100001
1111010110111110100010011010100010000111
```

You can find the full encoding at `http://cs.carleton.edu/faculty/dln/one-time-pad.txt`. Figure out (a) what 40-bit key I used, and (b) what the encoded document is.

7.120 *(programming required.)* Implement one-time pads in a programming language of your choice.

7.121 Using the "large" primes $p = 19$ and $q = 23$, compute the RSA public and private keys. You may have multiple valid choices for e—if so, choose the smallest e that you can.

7.122 Repeat for $p = 31$ and $q = 37$.

7.123 Repeat for $p = 41$ and $q = 43$.

Suppose that Bob's public key is $n = 221$ and $e = 5$. (And so Bob's private key is $n = 221$ and $d = 77$.)

7.124 Compute the RSA encryption to send Bob the message $m = 42$.

7.125 Repeat for the message $m = 99$.

7.126 If Bob receives the ciphertext $c = 99$, what message was sent to Bob?

7.127 Repeat for the ciphertext $c = 17$.

7.128 *(programming required.)* Suppose that Carol's public key is $\langle e = 3, n = 1{,}331{,}191\rangle$, and the ciphertext $c = 441{,}626$. Figure out the message that was sent to Carol by factoring n.

7.129 Repeat for the public key $\langle e = 11, n = 12{,}187{,}823\rangle$, and the ciphertext $c = 7{,}303{,}892$.

7.130 Repeat for the public key $\langle e = 5, n = 662{,}983{,}829\rangle$, and the ciphertext $c = 43{,}574{,}279$.

In both key generation and encryption/decryption, the RSA cryptosystem completes a number of steps that require some nonobvious ideas to make them efficient. Luckily, we've covered those ideas at various times in previous parts of the chapter. For each of the following, explain how to compute the desired quantity efficiently (that is, with a number of primitive arithmetic operations that's $O(\log^k n)$ for the value of n in the RSA protocol, for some constant k). For some of these exercises, you'll simply be able to cite a

previously developed algorithm in a few words; in others, you'll need to combine more than one algorithm or use an algorithm in a nontrivial way.

7.131 Find a large prime number: say, find the smallest prime number greater than a given number x.

7.132 Given primes p and q, find a number $e \neq 1$ such that e and $(p-1)(q-1)$ are relatively prime.

7.133 Given primes p, q and e relatively prime to $(p-1)(q-1)$, compute e^{-1} modulo $(p-1)(q-1)$.

7.134 Given n, e, and $m \in \mathbb{Z}_n$, compute $m^e \bmod n$. (Similarly, given n, d, and c, compute $c^d \bmod n$.)

7.135 Prove that, in the RSA key-generation protocol, the number e that we choose is always odd.

7.136 Prove that, in the RSA key-generation protocol, the number d that we choose is also always odd.

Imagine the following modifications to the RSA key generation protocol. What goes wrong if we change the algorithm as described? Be precise. Is there a step of the protocol that can no longer be executed? Does Bob no longer have the information necessary to decrypt the ciphertext? Does Eve now have the power to decrypt the ciphertext?

7.137 The protocol tells us to choose two large primes p, q. But, instead, we choose *one* prime p, and set $q := p$.

7.138 The protocol tells us to choose two large primes p and q. But, instead, we choose two large numbers p and q that aren't actually prime.

7.139 The protocol tells us to choose $e \neq 1$ that's relatively prime to $(p-1)(q-1)$. But, instead, we choose $e = 1$.

7.140 The protocol tells us to choose $e \neq 1$ that's relatively prime to $(p-1)(q-1)$. But, instead, we choose an e that is not relatively prime to $(p-1)(q-1)$.

7.141 Explain precisely how to use binary search to find the eth root of m^e efficiently. Then explain precisely why this binary-search approach doesn't work to find the eth root of $m^e \bmod n$ in general.

7.142 *(programming required.)* Implement RSA key generation. Given two prime numbers p and q as input, produce a public and private RSA keypair $\langle e, n\rangle$ and $\langle d, n\rangle$. *(Hint: Exercises 7.31 and 7.104 will be helpful. To pick e, you may wish to simply try all odd numbers and use Exercise 7.31—you could make this step faster, but generally speaking this slightly slower approach will still be fast enough.)* Use the results from Exercises 7.131–7.134 to make your solutions efficient.

7.143 *(programming required.)* Implement RSA encryption and decryption. For encryption, given a public key $\langle e, n\rangle$ and a message $m \in \mathbb{Z}_n$, compute the corresponding ciphertext $c := m^e \bmod n$. Similarly, for decryption: given a private key $\langle d, n\rangle$ and a ciphertext $c \in \mathbb{Z}_n$, compute $m := c^d \bmod n$. *(Hint: Exercise 7.25 will be helpful.)* Use the results from Exercises 7.131–7.134 to make your solutions efficient.

7.144 *(programming required.)* Generally, a user of a cryptographic system wants to send *text* rather than a *number,* so you'll need to add a capacity for converting text into an integer. And RSA will only support encrypting elements of $\mathbb{Z}_n$, not $\mathbb{Z}$, so you'll actually need to convert the text into a *sequence of elements of* $\mathbb{Z}_n$. Write a pair of functions `string->intlist`(s, n) and `intlist->string`(L, n) that convert between strings of characters and a list of elements from $\mathbb{Z}_n$. You may do this conversion in many ways, but you must ensure that these operations are inverses of each other: if `string->intlist`$(s^*, n) = L^*$, then `intlist->string`$(L^*, n) = s^*$. *(Hint: the easiest approach is to view text as a sequence of ASCII symbols, each of which is an element of* $\{0, 1, \ldots, 255\}$. *Thus you can view your input text as a number written in base* 256. *Your output is a number written in base n. Use* **baseConvert** *from p. 7-16.)*

7.145 *(programming required.)* Test your implementations from Exercises 7.142–7.144 by generating keys, encrypting, and decrypting using the primes $p = 5{,}277{,}019{,}477{,}592{,}911$ and $q = 7{,}502{,}904{,}222{,}052{,}693$, and the message `"THE SECRET OF BEING BORING IS TO SAY EVERYTHING."` (Voltaire (1694–1778)).

7.146 *(programming required.)* Complete the last missing piece of your RSA implementation: prime generation. The key generation implementation from Exercise 7.142 relies on being given two prime numbers. Write a function that, given a (sufficiently large) range of possible numbers between $n_{\min}$ and $n_{\max}$, repeatedly does the following: choose a random integer between $n_{\min}$ and $n_{\max}$, and test whether it's prime using the Miller–Rabin test (see Exercise 7.116).

7.147 *(programming required.)* The Chinese Remainder Theorem tells us that $m \in \mathbb{Z}_{pq}$ is uniquely described by its value modulo p and q—that is, $m \bmod p$ and $m \bmod q$ fully describe m. Here's one way to improve the efficiency of RSA using this observation: instead of computing $m := c^d \bmod pq$ directly, instead compute $a := c^d \bmod p$ and $b := c^d \bmod q$. Then use the algorithm implicit in Theorem 7.21 to compute the value m with $m \bmod p = a$ and $m \bmod q = b$. Modify your implementation of RSA to use this idea.

7.148 Actually, instead of computing $a := c^d \bmod p$ and $b := c^d \bmod q$ in Exercise 7.147, we could have instead computed $a := c^{d \bmod p-1} \bmod p$ and $b := c^{d \bmod q-1} \bmod q$. Explain why this modification is valid. (This change can improve the efficiency of RSA, because now both the base and the exponent may be substantially smaller than they were in the regular RSA implementation.)

7.6 Chapter at a Glance

Modular Arithmetic

Given integers $k \geq 1$ and n, there exist unique integers d and r such that $0 \leq r < k$ and $kd + r = n$. The value of d is $\left\lfloor \frac{n}{k} \right\rfloor$, the (whole) number of times k goes into n; the value of r is $n \bmod k$, the remainder when we divide n by k.

Two integers a and b are *equivalent* or *congruent mod n,* written $a \equiv_n b$, if a and b have the same remainder when divided by n—that is, when $a \bmod n = b \bmod n$. For expressions taken mod n, we can always freely "reduce" mod n (subtracting multiples of n) before performing addition or multiplication. (See Theorem 7.4.)

We write $k \mid n$ to denote the proposition that $n \bmod k = 0$. If $k \mid n$, we say that k *(evenly) divides* n, that k is a *factor* of n, and that n is a *multiple* of k. See Theorem 7.7 for some useful properties of divisibility: for example, if $a \mid b$ then, for any integer c, it's also the case that a divides bc as well. The *greatest common divisor* $\text{GCD}(n, m)$ of two positive integers n and m is the largest d that evenly divides both n and m; the *least common multiple* is the smallest $d \in \mathbb{Z}^{\geq 1}$ that n and m both evenly divide. GCDs can be computed efficiently using the *Euclidean algorithm.* (See Figure 7.30.)

Primality and Relative Primality

An integer $p \geq 2$ is *prime* if the only positive integers that evenly divide it are 1 and p itself; an integer $n \geq 2$ that is not prime is called *composite*. (Note that 1 is neither prime nor composite.) Let $\textit{primes}(n)$ denote the number of prime numbers less or equal than n. The *Prime Number Theorem* states that, as n gets large, the ratio between $\textit{primes}(n)$ and $\frac{n}{\log n}$ converges (slowly!) to 1. Every positive integer can be factored into a product of zero or more prime numbers, and that factorization is unique up to the ordering of the factors.

Two positive integers n and m are called *relatively prime* if they have no common factors aside from 1—that is, if $\text{GCD}(n, m) = 1$. A tweak to the Euclidean algorithm, called the *Extended Euclidean algorithm*, takes arbitrary positive integers n and m as input, and (efficiently) computes three integers x, y, r such that

```
Euclid(n, m):
Input: positive integers n and m ≥ n
Output: GCD(n, m)
1 if m mod n = 0 then
2     return n
3 else
4     return Euclid(m mod n, n)
```

Figure 7.30 The Euclidean algorithm for GCDs.

```
extended-Euclid(n, m):
Input: positive integers n and m ≥ n.
Output: x, y, r ∈ ℤ where GCD(n, m) = r = xn + ym
1 if m mod n = 0 then
2     return 1, 0, n          // 1 · n + 0 · m = n = GCD(n, m)
3 else
4     x, y, r := extended-Euclid(m mod n, n)
5     return y − ⌊m/n⌋ · x, x, r
```

Figure 7.31 The Extended Euclidean algorithm.

$r = \text{GCD}(n, m) = xn + ym$. (See Figure 7.31.) We can determine whether n and m are relatively prime using the (Extended) Euclidean algorithm.

Let $n_1, n_2, \ldots, n_k$ be a collection of integers, any pair of which is relatively prime. Let $N = \prod_{i=1}^{k} n_i$. Writing $\mathbb{Z}_m$ to denote $\{0, 1, \ldots, m-1\}$, the *Chinese Remainder Theorem* states that, for any sequence of values $\langle a_1, \ldots, a_k \rangle$ with each $a_i \in \mathbb{Z}_{n_i}$, there exists one and only one integer $x \in \mathbb{Z}_N$ such that $x \bmod n_i = a_i$ for all $1 \leq i \leq k$.

Multiplicative Inverses

For any integer $n \geq 2$, let $\mathbb{Z}_n$ denote the set $\{0, 1, \ldots, n-1\}$. Let $a \in \mathbb{Z}_n$ be arbitrary. The *multiplicative inverse of a in $\mathbb{Z}_n$* is the number $a^{-1} \in \mathbb{Z}_n$ such that $a \cdot a^{-1} \equiv_n 1$ if any such number exists. (If no such number exists, then a^{-1} is undefined.) For example, the multiplicative inverse of 2 in $\mathbb{Z}_9$ is $2^{-1} = 5$ because $2 \cdot 5 = 10 \equiv_9 1$; the multiplicative inverse of 1 in $\mathbb{Z}_9$ is $1^{-1} = 1$ because $1 \cdot 1 \equiv_9 1$; and the multiplicative inverse of 3 in $\mathbb{Z}_9$ is undefined (because $3a \not\equiv_9 1$ for any $a \in \mathbb{Z}_9$).

Let $n \geq 2$ and $a \in \mathbb{Z}_n$. The multiplicative inverse a^{-1} exists in $\mathbb{Z}_n$ if and only if n and a are relatively prime. Furthermore, when a^{-1} exists, we can find it using the Extended Euclidean algorithm. We compute $\langle x, y, r \rangle :=$ **extended-Euclid**(a, n); when $\text{GCD}(a, n) = 1$ (as it is when a and n are relatively prime), the returned values satisfy $xa + yn = 1$, and thus $a^{-1} := x \bmod n$ is the multiplicative inverse of a in $\mathbb{Z}_n$. For a prime number p, every nonzero $a \in \mathbb{Z}_p$ has a multiplicative inverse in $\mathbb{Z}_p$.

Fermat's Little Theorem states that, for any prime p and any integer a with $p \nmid a$, the $(p-1)$st power of a must equal 1 modulo p. (That is: for prime p and nonzero $a \in \mathbb{Z}_p$, we have $a^{p-1} \equiv_p 1$. For example, because 17 is prime, Fermat's Little Theorem—or arithmetic!—tells us that $5^{16} \bmod 17 = 1$.)

Cryptography

A sender ("Alice") wants to send a private message to a receiver ("Bob"), but they can only communicate using a channel that can be overheard by an eavesdropper ("Eve"). In *cryptography,* Alice *encrypts* the message m (the "plaintext") and transmits the encrypted version c (the "ciphertext"); Bob then *decrypts* it to recover the original message m. The simplest way to achieve this goal is with a *one-time pad:* Alice and Bob agree on a shared secret bitstring k; the ciphertext is the bitwise XOR of m and k, and Bob decrypts by computing the bitwise XOR of c and k.

A more useful infrastructure is *public-key cryptography,* in which Alice and Bob do not have to communicate a secret in advance. Every user has a *public key* and a (mathematically related) *private key*; to communicate with Bob, Alice uses Bob's public key for encryption (and Bob uses his private key for decryption). The *RSA cryptosystem* is a widely used protocol for public-key cryptography; it works as follows:

- **Key generation:** Bob finds large primes p and q; he chooses an $e \neq 1$ that's relatively prime to $(p-1)(q-1)$; and he computes $d := e^{-1}$ modulo $(p-1)(q-1)$. Bob's public key is $\langle e, n \rangle$ and his private key is $\langle d, n \rangle$, where $n := pq$.
- **Encryption:** When Alice wants to send m to Bob, she encrypts m as $c := m^e \bmod n$.
- **Decryption:** Bob decrypts c as $c^d \bmod n$.

By our choices of n, p, q, d, and e, Fermat's Little Theorem allows us to prove that Bob's decryption of the encryption of message m is always the original message m itself. And, under commonly held beliefs about the difficulty of factoring large numbers (and computing "eth roots mod n"), Eve cannot compute m without spending an implausibly large amount of computation time.

Key Terms and Results

Key Terms

Modular Arithmetic

- modulus; n mod k and $\left\lfloor \frac{n}{k} \right\rfloor$
- congruence/equivalence ($\equiv_n$)
- (evenly) divides, factor, multiple
- greatest common divisor
- least common multiple
- Euclidean algorithm

Primality and Relative Primality

- prime vs. composite numbers
- Prime Number Theorem
- prime factorization
- relative primality
- Extended Euclidean algorithm
- Chinese Remainder Theorem

Multiplicative Inverses

- $\mathbb{Z}_n$
- multiplicative inverse (a^{-1} in $\mathbb{Z}_n$)
- Fermat's Little Theorem
- Carmichael number

Cryptography

- Alice, Bob, Eve
- plaintext, ciphertext
- one-time pad
- public-key cryptography
- public key; private key
- key generation; encryption/decryption
- RSA

Key Results

Modular Arithmetic

1 For any integers $k \geq 1$ and n, there exist unique integers d and r such that $0 \leq r < k$ and $kd + r = n$. (And $r = n \bmod k$ and $d = \left\lfloor \frac{n}{k} \right\rfloor$.)

2 For arbitrary positive integers n and $m \geq n$, the Euclidean algorithm efficiently computes $\text{GCD}(n, m)$.

Primality and Relative Primality

1 The *Prime Number Theorem:* as n gets large, the ratio between $\frac{n}{\log n}$ and the number of primes less than or equal to n approaches 1.

2 Every positive integer has a prime factorization (which is unique up to reordering).

3 Given positive integers n and m, the Extended Euclidean algorithm efficiently computes three integers x, y, r such that $r = \text{GCD}(n, m) = xn + ym$.

4 The *Chinese Remainder Theorem:* Suppose $n_1, n_2, \ldots, n_k$ are all relatively prime, and let $N = \prod_{i=1}^{k} n_i$. Then, for any $\langle a_1, \ldots, a_k \rangle$ with each $a_i \in \mathbb{Z}_{n_i}$, there exists a unique $x \in \mathbb{Z}_N$ such that $x \bmod n_i = a_i$ for all $1 \leq i \leq k$.

Multiplicative Inverses

1 In $\mathbb{Z}_n$, the multiplicative inverse a^{-1} of a exists if and only if n and a are relatively prime. When it does exist, we can find a^{-1} using the Extended Euclidean algorithm.

2 *Fermat's Little Theorem:* for any prime number p and any nonzero $a \in \mathbb{Z}_p$, we have $a^{p-1} \equiv_p 1$.

Cryptography

1 In the RSA cryptosystem, Alice can use Bob's public key to encrypt a message m so that Bob can decrypt it efficiently. (And, under reasonable assumptions about certain numerical problems' hardness, Eve *can't* recover m without an exorbitant amount of computation.)