Introduction to Algorithms, Second Edition

Chapter 31: Number-Theoretic Algorithms

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in part to the invention of cryptographic schemes based on large prime numbers. The feasibility of these schemes rests on our ability to find large primes easily, while their security rests on our inability to factor the product of large primes. This chapter presents some of the number theory and associated algorithms that underlie such applications.

Section 31.1 introduces basic concepts of number theory, such as divisibility, modular equivalence, and unique factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers. Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then studies the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax \equiv b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6 considers powers of a given number a, modulo n, and presents a repeated-squaring algorithm for efficiently computing $a^b \bmod n$, given a, b, and n. This operation is at the heart of efficient primality testing and of much modern cryptography. Section 31.7 then describes the RSA public-key cryptosystem. Section 31.8 examines a randomized primality test that can be used to find large primes efficiently, an essential task in creating keys for the RSA cryptosystem. Finally, Section 31.9 reviews a simple but effective heuristic for factoring small integers. It is a curious fact that factoring is one problem people may wish to be intractable, since the security of RSA depends on the difficulty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a "large input" typically means an input containing "large integers" rather than an input containing "many integers" (as for sorting). Thus, we shall measure the size of an input in terms of the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer inputs $a_1, a_2, \ldots, a_k$ is a ***polynomial-time algorithm*** if it runs in time polynomial in $\lg a_1, \lg a_2, \ldots, \lg a_k$, that is, polynomial in the lengths of its binary-encoded inputs.

In most of this book, we have found it convenient to think of the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. By counting the number of such arithmetic operations that an algorithm performs, we have a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes convenient to measure how many ***bit operations*** a number-theoretic algorithm requires. In this model, a multiplication of two β-bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, the operation of dividing a β-bit integer by a shorter integer, or the operation of taking the remainder of a β-bit integer when divided by a shorter integer, can be performed in time $\Theta(\beta^2)$ by simple algorithms. (See Exercise 31.1-11.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β-bit integers has a running time of , and the fastest known method has a running time of $\Theta(\beta \lg \beta \lg \lg \beta)$. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we shall use this bound as a basis for our analyses.

In this chapter, algorithms are generally analyzed in terms of both the number of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set $\mathbf{Z} = \{..., -2, -1, 0, 1, 2,..\}$ of integers and the set $\mathbf{N} = \{0, 1, 2, ...\}$ of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is a central one in the theory of numbers. The notation $d \mid a$ (read "d ***divides*** a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \leq |a|$. If $d \mid a$, then we also say that a is a ***multiple*** of d. If d does not divide a, we write $d \nmid a$.

If $d \mid a$ and $d \geq 0$, we say that d is a ***divisor*** of a. Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of an integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every integer a is divisible by the ***trivial divisors*** 1 and a. Nontrivial divisors of a are also called ***factors*** of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is said to be a ***prime*** number (or, more simply, a ***prime***). Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71.

Exercise 31.1-1 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is said to be a ***composite*** number (or, more simply, a ***composite***). For example, 39

In most of this book, we have found it convenient to think of the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. By counting the number of such arithmetic operations that an algorithm performs, we have a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes convenient to measure how many ***bit operations*** a number-theoretic algorithm requires. In this model, a multiplication of two β-bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, the operation of dividing a β-bit integer by a shorter integer, or the operation of taking the remainder of a β-bit integer when divided by a shorter integer, can be performed in time $\Theta(\beta^2)$ by simple algorithms. (See Exercise 31.1-11.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β-bit integers has a running time of $\Theta(\beta^{\lg 3})$, and the fastest known method has a running time of $\Theta(\beta \lg \beta \lg \lg \beta)$. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we shall use this bound as a basis for our analyses.

In this chapter, algorithms are generally analyzed in terms of both the number of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set $\mathbf{Z} = \{..., -2, -1, 0, 1, 2,..\}$ of integers and the set $\mathbf{N} = \{0, 1, 2, ...\}$ of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is a central one in the theory of numbers. The notation $d \mid a$ (read "d ***divides*** a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \le |a|$. If $d \mid a$, then we also say that a is a ***multiple*** of d. If d does not divide a, we write $d \nmid a$.

If $d \mid a$ and $d \ge 0$, we say that d is a ***divisor*** of a. Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of an integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every integer a is divisible by the ***trivial divisors*** 1 and a. Nontrivial divisors of a are also called ***factors*** of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is said to be a ***prime*** number (or, more simply, a ***prime***). Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71.

Exercise 31.1-1 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is said to be a ***composite*** number (or, more simply, a ***composite***). For example, 39

is composite because 3 | 39. The integer 1 is said to be a ***unit*** and is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n, the integers can be partitioned into those that are multiples of n and those that are not multiples of n. Much number theory is based upon a refinement of this partition obtained by classifying the nonmultiples of n according to their remainders when divided by n. The following theorem is the basis for this refinement. The proof of this theorem will not be given here (see, for example, Niven and Zuckerman [231]).

Theorem 31.1: (Division theorem)

For any integer a and any positive integer n, there are unique integers q and r such that $0 \le r < n$ and $a = qn + r$.

The value $q = \lfloor a/n \rfloor$ is the ***quotient*** of the division. The value $r = a \bmod n$ is the ***remainder*** (or ***residue***) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$.

The integers can be divided into n equivalence classes according to their remainders modulo n. The ***equivalence class modulo n*** containing an integer a is

$[a]_n = \{a + kn : k \quad \mathbf{Z}\}$.

For example, $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$; other denotations for this set are $[-4]_7$ and $[10]_7$. Using the notation defined on page 52, we can say that writing $a \quad [b]_n$ is the same as writing $a \equiv b \pmod{n}$. The set of all such equivalence classes is

(31.1) $\mathbf{Z}_n = \{[a]_n : 0 \le a \le n - 1\}$.

One often sees the definition

(31.2) $\mathbf{Z}_n = \{0, 1, \ldots, n - 1\}$,

which should be read as equivalent to equation (31.1) with the understanding that 0 represents $[0]_n$, 1 represents $[1]_n$, and so on; each class is represented by its least nonnegative element. The underlying equivalence classes must be kept in mind, however. For example, a reference to -1 as a member of $\mathbf{Z}_n$ is a reference to $[n - 1]_n$, since $-1 \equiv n - 1 \pmod{n}$.

Common divisors and greatest common divisors

If d is a divisor of a and d is also a divisor of b, then d is a ***common divisor*** of a and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor of any two integers.

An important property of common divisors is that

(31.3) $d \mid a$ and $d \mid b$ implies $d \mid (a + b)$ and $d \mid (a - b)$.

More generally, we have that

(31.4) $d \mid a$ and $d \mid b$ implies $d \mid (ax + by)$

for any integers x and y. Also, if $a \mid b$, then either $|a| \leq |b|$ or $b = 0$, which implies that

(31.5) $a \mid b$ and $b \mid a$ implies $a = \pm b$.

The ***greatest common divisor*** of two integers a and b, not both zero, is the largest of the common divisors of a and b; it is denoted gcd(a, b). For example, gcd(24, 30) = 6, gcd(5, 7) = 1, and gcd(0, 9) = 9. If a and b are not both 0, then gcd(a, b) is an integer between 1 and min(|a|, |b|). We define gcd(0, 0) to be 0; this definition is necessary to make standard properties of the gcd function (such as equation (31.9) below) universally valid.

The following are elementary properties of the gcd function:

(31.6) $\gcd(a, b) = \gcd(b, a)$,
(31.7) $\gcd(a, b) = \gcd(-a, b)$,
(31.8) $\gcd(a, b) = \gcd(|a|, |b|)$,
(31.9) $\gcd(a, 0) = |a|$,
(31.10) $\gcd(a, ka) = |a|$ for any $k \in \mathbf{Z}$.

The following theorem provides an alternative and useful characterization of gcd(a, b).

Theorem 31.2

If a and b are any integers, not both zero, then gcd(a, b) is the smallest positive element of the set {$ax + by : x, y$ **Z**} of linear combinations of a and b.

Proof Let s be the smallest positive such linear combination of a and b, and let $s = ax + by$ for some x, y Z. Let $q = \lfloor a/s \rfloor$. Equation (3.8) then implies

$$
\begin{aligned}
a \bmod s &= a - qs \\
&= a - q(ax + by) \\
&= a(1 - qx) + b(-qy),
\end{aligned}
$$

and so a mod s is a linear combination of a and b as well. But, since a mod $s < s$, we have that a mod $s = 0$, because s is the smallest positive such linear combination. Therefore, $s \mid a$ and, by analogous reasoning, $s \mid b$. Thus, s is a common divisor of a and b, and so gcd(a, b) $\geq s$. Equation (31.4) implies that gcd(a, b) $\mid s$, since gcd(a, b) divides both a and b and s is a linear combination of a and b. But gcd(a, b) $\mid s$ and $s > 0$ imply that gcd(a, b) $\leq s$. Combining gcd(a, b) $\geq s$ and gcd(a, b) $\leq s$ yields gcd(a, b) = s; we conclude that s is the greatest common divisor of a and b.

Corollary 31.3

For any integers a and b, if $d \mid a$ and $d \mid b$ then $d \mid \gcd(a, b)$.

Proof This corollary follows from equation (31.4), because $\gcd(a, b)$ is a linear combination of a and b by Theorem 31.2.

Corollary 31.4

For all integers a and b and any nonnegative integer n,

$\gcd(an, bn) = n \gcd(a, b)$.

Proof If $n = 0$, the corollary is trivial. If $n > 0$, then $\gcd(an, bn)$ is the smallest positive element of the set $\{anx + bny\}$, which is n times the smallest positive element of the set $\{ax + by\}$.

Corollary 31.5

For all positive integers n, a, and b, if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

Proof The proof is left as Exercise 31.1-4.

Relatively prime integers

Two integers a, b are said to be ***relatively prime*** if their only common divisor is 1, that is, if $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since the divisors of 8 are 1, 2, 4, and 8, while the divisors of 15 are 1, 3, 5, and 15. The following theorem states that if two integers are each relatively prime to an integer p, then their product is relatively prime to p.

Theorem 31.6

For any integers a, b, and p, if both $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then $\gcd(ab, p) = 1$.

Proof It follows from Theorem 31.2 that there exist integers x, y, x', and y' such that

$$ax + py = 1 ,$$
$$bx' + py' = 1 .$$

Multiplying these equations and rearranging, we have

$ab(x\,x') + p(ybx' + y'ax + pyy') = 1.$

Since 1 is thus a positive linear combination of ab and p, an appeal to Theorem 31.2 completes the proof.

We say that integers $n_1, n_2, \ldots, n_k$ are ***pairwise relatively prime*** if, whenever $i \neq j$, we have $\gcd(n_i, n_j) = 1$.

Unique factorization

An elementary but important fact about divisibility by primes is the following.

Theorem 31.7

For all primes p and all integers a, b, if $p \mid ab$, then $p \mid a$ or $p \mid b$ (or both).

Proof Assume for the purpose of contradiction that $p \mid ab$ but that $p \nmid a$ and $p \nmid b$. Thus, $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, since the only divisors of p are 1 and p, and by assumption p divides neither a nor b. Theorem 31.6 then implies that $\gcd(ab, p) = 1$, contradicting our assumption that $p \mid ab$, since $p \mid ab$ implies $\gcd(ab, p) = p$. This contradiction completes the proof.

A consequence of Theorem 31.7 is that an integer has a unique factorization into primes.

Theorem 31.8: (Unique factorization)

A composite integer a can be written in exactly one way as a product of the form

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

where the p_i are prime, $p_1 < p_2 < \cdots < p_r$, and the e_i are positive integers.

Proof The proof is left as Exercise 31.1-10.

As an example, the number 6000 can be uniquely factored as $2^4 \cdot 3 \cdot 5^3$.

Exercises 31.1-1

Prove that there are infinitely many primes. (*Hint:* how that none of the primes $p_1, p_2, ..., p_k$ divide $(p_1 p_2 \cdots p_k) + 1$.)

Exercises 31.1-2

Prove that if $a \mid b$ and $b \mid c$, then $a \mid c$.

Exercises 31.1-3

Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

Exercises 31.1-4

Prove Corollary 31.5.

Exercises 31.1-5

Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a, b, and primes p,

$(a + b)^p \equiv a^p + b^p \pmod{p}$.

Exercises 31.1-6

Prove that if a and b are any integers such that $a \mid b$ and $b > 0$, then

$(x \bmod b) \bmod a = x \bmod a$

for any x. Prove, under the same assumptions, that

$x \equiv y \bmod b)$ implies $x \equiv y \pmod{a}$

for any integers x and y.

Exercises 31.1-7

For any integer $k > 0$, we say that an integer n is a ***kth power*** if there exists an integer a such that $a^k = n$. We say that $n > 1$ is a ***nontrivial power*** if it is a kth power for some integer $k > 1$. Show how to determine if a given β-bit integer n is a nontrivial power in time polynomial in β.

Exercises 31.1-8

Prove equations (31.6)–(31.10).

Exercises 31.1-9

Show that the gcd operator is associative. That is, prove that for all integers a, b, and c,

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

Exercises 31.1-10:

Prove Theorem 31.8.

Exercises 31.1-11

Give efficient algorithms for the operations of dividing a β-bit integer by a shorter integer and of taking the remainder of a β-bit integer when divided by a shorter integer. Your algorithms should run in time $O(\beta^2)$.

Exercises 31.1-12

Give an efficient algorithm to convert a given β-bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes time $M(\beta)$, then binary-to-decimal conversion can be performed in time $\Theta(M(\beta) \lg \beta)$.

(*Hint:* Use a divide-and-conquer approach, obtaining the top and bottom halves of the result with separate recursions.)

31.2 Greatest common divisor

In this section, we describe Euclid's algorithm for computing the greatest common divisor of two integers efficiently. The analysis of running time brings up a surprising connection with the Fibonacci numbers, which yield a worst-case input for Euclid's algorithm.

We restrict ourselves in this section to nonnegative integers. This restriction is justified by equation (31.8), which states that gcd(a, b) = gcd($|a|$, $|b|$).

In principle, we can compute gcd(a, b) for positive integers a and b from the prime factorizations of a and b. Indeed, if

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.11}$$
$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \tag{31.12}$$

with zero exponents being used to make the set of primes $p_1, p_2, \ldots, p_r$ the same for both a and b, then, as Exercise 31.2-1 asks you to show,

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} . \tag{31.13}$$

As we shall show in Section 31.9, however, the best algorithms to date for factoring do not run in polynomial time. Thus, this approach to computing greatest common divisors seems unlikely to yield an efficient algorithm.

Euclid's algorithm for computing greatest common divisors is based on the following theorem.

Theorem 31.9: (GCD recursion theorem)

For any nonnegative integer a and any positive integer b,

gcd(a, b) = gcd(b, a mod b).

Proof We shall show that gcd(a, b) and gcd(b, a mod b) divide each other, so that by equation (31.5) they must be equal (since they are both nonnegative).

We first show that gcd(a, b) | gcd(b, a mod b). If we let d = gcd(a, b), then $d \mid a$ and $d \mid b$. By equation (3.8), (a mod b) = $a - qb$, where $q = \lfloor a/b \rfloor$. Since (a mod b) is thus a linear combination of a and b, equation (31.4) implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, Corollary 31.3 implies that $d \mid \gcd(b, a \bmod b)$ or, equivalently, that

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \tag{31.14}$$

Showing that gcd(b, a mod b) | gcd(a, b) is almost the same. If we now let d = gcd(b, a mod b), then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$, we have that a is a

linear combination of b and (a mod b). By equation (31.4), we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ by Corollary 31.3 or, equivalently, that

(31.15) $\gcd(b, a \bmod b) \mid \gcd(a, b)$.

Using equation (31.5) to combine equations (31.14) and (31.15) completes the proof.

Euclid's algorithm

The *Elements* of Euclid (*circa* 300 B.C.) describes the following gcd algorithm, although it may be of even earlier origin. Euclid's algorithm is expressed as a recursive program based directly on Theorem 31.9. The inputs a and b are arbitrary nonnegative integers.

```
EUCLID(a, b)
1  if b = 0
2    then return a
3    else return EUCLID(b, a mod b)
```

As an example of the running of EUCLID, consider the computation of gcd(30, 21):

EUCLID(30, 21) = EUCLID(21,9)
= EUCLID(9,3)
= EUCLID(3,0)
= 3.

In this computation, there are three recursive invocations of EUCLID.

The correctness of EUCLID follows from Theorem 31.9 and the fact that if the algorithm returns a in line 2, then $b = 0$, so equation (31.9) implies that $\gcd(a, b) = \gcd(a, 0) = a$. The algorithm cannot recurse indefinitely, since the second argument strictly decreases in each recursive call and is always nonnegative. Therefore, EUCLID always terminates with the correct answer.

The running time of Euclid's algorithm

We analyze the worst-case running time of EUCLID as a function of the size of a and b. We assume with no loss of generality that $a > b \geq 0$. This assumption can be justified by the observation that if $b > a \geq 0$, then EUCLID(a, b) immediately makes the recursive call EUCLID(b, a). That is, if the first argument is less than the second argument, EUCLID spends one recursive call swapping its arguments and then proceeds. Similarly, if $b = a > 0$, the procedure terminates after one recursive call, since a mod $b = 0$.

The overall running time of EUCLID is proportional to the number of recursive calls it makes. Our analysis makes use of the Fibonacci numbers F_k, defined by the recurrence (3.21).

Lemma 31.10

If $a > b \geq 1$ and the invocation EUCLID(a, b) performs $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.

Proof The proof is by induction on k. For the basis of the induction, let $k = 1$. Then, $b \geq 1 = F_2$, and since $a > b$, we must have $a \geq 2 = F_3$. Since $b > (a \bmod b)$, in each recursive call the first argument is strictly larger than the second; the assumption that $a > b$ therefore holds for each recursive call.

Assume inductively that the lemma is true if k - 1 recursive calls are made; we shall then prove that it is true for k recursive calls. Since $k > 0$, we have $b > 0$, and EUCLID(a, b) calls EUCLID(b, $a \bmod b$) recursively, which in turn makes k - 1 recursive calls. The inductive hypothesis then implies that $b \geq F_{k+1}$ (thus proving part of the lemma), and $(a \bmod b) \geq F_k$. We have

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor\, b) \\ &\leq a\ , \end{aligned}$$

since $a > b > 0$ implies $\lfloor a/b \rfloor \geq 1$. Thus,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2}\ . \end{aligned}$$

The following theorem is an immediate corollary of this lemma.

Theorem 31.11: (Lamé's theorem)

For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call EUCLID(a, b) makes fewer than k recursive calls.

We can show that the upper bound of Theorem 31.11 is the best possible. Consecutive Fibonacci numbers are a worst-case input for EUCLID. Since EUCLID(F_3, F_2) makes exactly one recursive call, and since for $k > 2$ we have $F_{k+1} \bmod F_k = F_{k-1}$, we also have

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, (F_{k+1} \bmod F_k)) \\ &= \gcd(F_k, F_{k-1}). \end{aligned}$$

Thus, EUCLID(F_{k+1}, F_k) recurses *exactly* k - 1 times, meeting the upper bound of Theorem 31.11.

Since F_k is approximately $\phi^k/\sqrt{5}$, where φ is the golden ratio $(1 + \sqrt{5})/2$defined by equation (3.22), the number of recursive calls in EUCLID is $O(\lg b)$. (See Exercise 31.2-5 for a tighter bound.) It follows that if EUCLID is applied to two β-bit numbers, then it will perform $O(\beta)$ arithmetic operations and $O(\beta^3)$ bit operations (assuming that multiplication and division of β-bit numbers take $O(\beta^2)$ bit operations). Problem 31-2 asks you to show an $O(\beta^2)$ bound on the number of bit operations.

The extended form of Euclid's algorithm

We now rewrite Euclid's algorithm to compute additional useful information. Specifically, we extend the algorithm to compute the integer coefficients x and y such that

(31.16) $d = \gcd(a, b) = ax + by$.

Note that x and y may be zero or negative. We shall find these coefficients useful later for the computation of modular multiplicative inverses. The procedure EXTENDED-EUCLID takes as input a pair of nonnegative integers and returns a triple of the form (d, x, y) that satisfies equation (31.16).

```
EXTENDED-EUCLID(a, b)
1  if b = 0
2     then return (a, 1, 0)
3  (d′, x′, y′) ← EXTENDED-EUCLID(b, a mod b)
4  (d, x, y) ← (d′, y′, x′- ⌊a/b⌋ y′)
5  return (d, x, y)
```

Figure 31.1 illustrates the execution of EXTENDED-EUCLID with the computation of gcd(99, 78).

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	—	3	1	0

Figure 31.1: An example of the operation of EXTENDED-EUCLID on the inputs 99 and 78. Each line shows one level of the recursion: the values of the inputs a and b, the computed value $\lfloor a/b \rfloor$, and the values d, x, and y returned. The triple (d, x, y) returned becomes the triple (d', x', y') used in the computation at the next higher level of recursion. The call EXTENDED-EUCLID(99, 78) returns (3, -11, 14), so gcd(99, 78) = 3 and gcd(99, 78) = 3 = 99 · (-11) + 78 · 14.

The EXTENDED-EUCLID procedure is a variation of the EUCLID procedure. Line 1 is equivalent to the test "$b = 0$" in line 1 of EUCLID. If $b = 0$, then EXTENDED-EUCLID returns not only $d = a$ in line 2, but also the coefficients $x = 1$ and $y = 0$, so that $a = ax + by$. If $b \neq 0$, EXTENDED-EUCLID first computes (d', x', y') such that $d' = \gcd(b, a \bmod b)$ and

$$d' = bx' + (a \bmod b)y' . \tag{31.17}$$

As for EUCLID, we have in this case $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. To obtain x and y such that $d = ax + by$, we start by rewriting equation (31.17) using the equation $d = d'$ and equation (3.8):

$$\begin{aligned} d &= bx' + (a - \lfloor a/b \rfloor\, b)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor\, y'). \end{aligned}$$

Thus, choosing $x = y'$ and $y = x' - \lfloor a/b \rfloor\, y'$ satisfies the equation $d = ax + by$, proving the correctness of EXTENDED-EUCLID.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

Exercises 31.2-1

Prove that equations (31.11) and (31.12) imply equation (31.13).

Exercises 31.2-2

Compute the values (d, x, y) that the call EXTENDED-EUCLID(899, 493) returns.

Exercises 31.2-3

Prove that for all integers a, k, and n,

$\gcd(a, n) = \gcd(a + kn, n)$.

Exercises 31.2-4

Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

Exercises 31.2-5

If $a > b \geq 0$, show that the invocation EUCLID(a, b) makes at most $1 + \log_\varphi b$ recursive calls. Improve this bound to $1 + \log_\varphi(b/\gcd(a, b))$.

Exercises 31.2-6

What does EXTENDED-EUCLID(F_{k+1}, F_k) return? Prove your answer correct.

Exercises 31.2-7

Define the gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, ..., a_n) = \gcd(a_0, \gcd(a_1, a_2, ..., a_n))$. Show that the gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers $x_0, x_1, ..., x_n$ such that $\gcd(a_0, a_1, ..., a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, ..., a_n\}))$.

Exercises 31.2-8

Define $\text{lcm}(a_1, a_2, ..., a_n)$ to be the ***least common multiple*** of the n integers $a_1, a_2, ..., a_n$, that is, the least nonnegative integer that is a multiple of each a_i. Show how to compute $\text{lcm}(a_1, a_2, ..., a_n)$ efficiently using the (two-argument) gcd operation as a subroutine.

Exercises 31.2-9

Prove that n_1, n_2, n_3, and n_4 are pairwise relatively prime if and only if

$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$.

Show more generally that $n_1, n_2, ..., n_k$ are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

31.3 Modular arithmetic

Informally, we can think of modular arithmetic as arithmetic as usual over the integers, except that if we are working modulo n, then every result x is replaced by the element of $\{0, 1, ..., n - 1\}$ that is equivalent to x, modulo n (that is, x is replaced by $x \bmod n$). This informal model is sufficient if we stick to the operations of addition, subtraction, and multiplication. A more formal model for modular arithmetic, which we now give, is best described within the framework of group theory.

Finite groups

A ***group*** $(S, \ )$ is a set S together with a binary operation defined on S for which the following properties hold.

1. **Closure:** For all $a, b \ S$, we have $a \ b \ S$.
2. **Identity:** There is an element $e \ S$, called the ***identity*** of the group, such that $e \ a = a \ e = a$ for all $a \ S$.
3. **Associativity:** For all $a, b, c \ S$, we have $(a \ b) \ c = a \ (b \ c)$.
4. **Inverses:** For each $a \ S$, there exists a unique element $b \ S$, called the ***inverse*** of a, such that $a \ b = b \ a = e$.

As an example, consider the familiar group $(\mathbf{Z}, +)$ of the integers $\mathbf{Z}$ under the operation of addition: 0 is the identity, and the inverse of a is $-a$. If a group $(S, \ )$ satisfies the ***commutative law*** $a \ b = b \ a$ for all $a, b \ S$, then it is an ***abelian group***. If a group $S, \ )$ satisfies $|S| < \infty$, then it is a ***finite group***.

The groups defined by modular addition and multiplication

We can form two finite abelian groups by using addition and multiplication modulo n, where n is a positive integer. These groups are based on the equivalence classes of the integers modulo n, defined in Section 31.1.

To define a group on $\mathbf{Z}_n$, we need to have suitable binary operations, which we obtain by redefining the ordinary operations of addition and multiplication. It is easy to define addition and multiplication operations for $\mathbf{Z}_n$, because the equivalence class of two integers uniquely determines the equivalence class of their sum or product. That is, if $a \equiv a' \pmod n$ and $b \equiv b' \pmod n$, then

$$a + b \equiv a' + b' \pmod n ,$$
$$ab \equiv a'b' \pmod n .$$

Thus, we define addition and multiplication modulo n, denoted $+_n$ and $\cdot_n$, as follows:

$$[a]_n +_n [b]_n = [a + b]_n , \tag{31.18}$$
$$[a]_n \cdot_n [b]_n = [ab]_n .$$

(Subtraction can be similarly defined on $\mathbf{Z}_n$ by $[a]_n -_n [b]_n = [a-b]_n$, but division is more complicated, as we shall see.) These facts justify the common and convenient practice of using the least nonnegative element of each equivalence class as its representative when

performing computations in $\mathbf{Z}_n$. Addition, subtraction, and multiplication are performed as usual on the representatives, but each result x is replaced by the representative of its class (that is, by x mod n).

Using this definition of addition modulo n, we define the ***additive group modulo n*** as $(\mathbf{Z}_n, +_n)$. This size of the additive group modulo n is $|\mathbf{Z}_n| = n$. Figure 31.2(a) gives the operation table for the group $(\mathbf{Z}_6, +_6)$.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figure 31.2: Two finite groups. Equivalence classes are denoted by their representative elements. *(a)* The group $(Z_6, +_6)$. *(b)* The group $(\mathbf{Z}^*_{15}, \cdot_{15})$.

Theorem 31.12

The system $(\mathbf{Z}_n, +_n)$ is a finite abelian group.

Proof Equation (31.18) shows that $(\mathbf{Z}_n, +_n)$ is closed. Associativity and commutativity of $+_n$ follow from the associativity and commutativity of $+$:

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\ &= [(a + b) + c]_n \\ &= [a + (b + c)]_n \\ &= [a]_n +_n [b + c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n),\end{aligned}$$

$$\begin{aligned}[a]_n +n [b]_n &= [a + b]_n \\ &= [b + a]_n \\ &= [b]_n +_n [a]_n.\end{aligned}$$

The identity element of $(\mathbf{Z}_n, +_n)$ is 0 (that is, $[0]_n$). The (additive) inverse of an element a (that is, of $[a]_n$) is the element $-a$ (that is, $[-a]_n$ or $[n - a]_n$), since $[a]_n +_n [-a]_n = [a - a]_n = [0]_n$.

Using the definition of multiplication modulo n, we define the ***multiplicative group modulo n*** as $(\mathbf{Z}^*_n, \cdot_n)$. The elements of this group are the set $\mathbf{Z}^*_n$ of elements in $\mathbf{Z}_n$ that are relatively prime to n:

$$\mathbf{Z}^*_n = \{[a]_n \in \mathbf{Z}_n : \gcd(a, n) = 1\} .$$

To see that $\mathbf{Z}_n^*$ is well defined, note that for $0 \le a < n$, we have $a \equiv (a + kn) \pmod{n}$ for all integers k. By Exercise 31.2-3, therefore, $\gcd(a, n) = 1$ implies $\gcd(a + kn, n) = 1$ for all integers k. Since $[a]_n = \{a + kn : k \quad \mathbf{Z}\}$, the set $\mathbf{Z}_n^*$ is well defined. An example of such a group is

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

where the group operation is multiplication modulo 15. (Here we denote an element $[a]_{15}$ as a; for example, we denote $[7]_{15}$ as 7.) Figure 31.2(b) shows the group $(\mathbf{Z}_{15}^*, \cdot_{15})$. For example, $8 \cdot 11 \equiv 13 \bmod 15$), working in $\mathbf{Z}_{15}^*$. The identity for this group is 1.

Theorem 31.13

The system $(\mathbf{Z}_n^*, \cdot_n)$ is a finite abelian group.

Proof Theorem 31.6 implies that $(\mathbf{Z}_n^*, \cdot_n)$ is closed. Associativity and commutativity can be proved for $\cdot_n$ as they were for $+_n$ in the proof of Theorem 31.12. The identity element is $[1]_n$. To show the existence of inverses, let a be an element of $\mathbf{Z}_n^*$ and let (d, x, y) be the output of EXTENDED-EUCLID(a, n). Then $d = 1$, since $a \in \mathbf{Z}_n^*$, and

$$ax + ny = 1$$

or, equivalently,

$$ax \equiv 1 \pmod{n}.$$

Thus, $[x]_n$ is a multiplicative inverse of $[a]_n$, modulo n. The proof that inverses are uniquely defined is deferred until Corollary 31.26.

As an example of computing multiplicative inverses, suppose that $a = 5$ and $n = 11$. Then EXTENDED-EUCLID(a, n) returns $(d, x, y) = (1, -2, 1)$, so that $1 = 5 \cdot (-2) + 11 \cdot 1$. Thus, -2 (i.e., 9 mod 11) is a multiplicative inverse of 5 modulo 11.

When working with the groups $(\mathbf{Z}_n, +_n)$ and $(\mathbf{Z}_n^*, \cdot_n)$ in the remainder of this chapter, we follow the convenient practice of denoting equivalence classes by their representative elements and denoting the operations $+_n$ and $\cdot_n$ by the usual arithmetic notations + and · (or juxtaposition) respectively. Also, equivalences modulo n may also be interpreted as equations in $\mathbf{Z}_n$. For example, the following two statements are equivalent:

$$ax \equiv b \pmod{n} ,$$
$$[a]_n \cdot_n [x]_n = [b]_n .$$

As a further convenience, we sometimes refer to a group $(S, \ )$ merely as S when the operation is understood from context. We may thus refer to the groups $(\mathbf{Z}_n, +_n)$ and $(\mathbf{Z}_n^*, \cdot_n)$ as $\mathbf{Z}_n$ and $\mathbf{Z}_n^*$, respectively.

The (multiplicative) inverse of an element a is denoted $(a^{-1} \bmod n)$. Division in $\mathbf{Z}_n^*$ is defined by the equation $a/b \equiv ab^{-1} \pmod{n}$. For example, in $\mathbf{Z}_{15}^*$ we have that $7^{-1} \equiv 13 \pmod{15}$, since $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, so that $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

The size of $\mathbf{Z}_n^*$ is denoted $\varphi(n)$. This function, known as ***Euler's phi function***, satisfies the equation

$$\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) , \tag{31.19}$$

where p runs over all the primes dividing n (including n itself, if n is prime). We shall not prove this formula here. Intuitively, we begin with a list of the n remainders $\{0, 1, ..., n - 1\}$ and then, for each prime p that divides n, cross out every multiple of p in the list. For example, since the prime divisors of 45 are 3 and 5,

$$\begin{aligned} \phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 . \end{aligned}$$

If p is prime, then $\mathbf{Z}_p^* = \{1, 2, \ldots, p - 1\}$, and

$$\phi(p) = p - 1 . \tag{31.20}$$

If n is composite, then $\varphi(n) < n - 1$.

Subgroups

If $(S, \ )$ is a group, $S' \quad S$, and $(S', \ )$ is also a group, then $(S', \ )$ is said to be a ***subgroup*** of $(S, \ )$. For example, the even integers form a subgroup of the integers under the operation of addition. The following theorem provides a useful tool for recognizing subgroups.

Theorem 31.14: (A nonempty closed subset of a finite group is a subgroup)

If $(S, \ )$ is a finite group and S' is any nonempty subset of S such that $a \quad b \quad S'$ for all $a, b \quad S'$, then $(S', \ )$ is a subgroup of $(S, \ )$.

Proof The proof is left as Exercise 31.3-2.

For example, the set $\{0, 2, 4, 6\}$ forms a subgroup of $\mathbf{Z}_8$, since it is nonempty and closed under the operation + (that is, it is closed under $+_8$).

The following theorem provides an extremely useful constraint on the size of a subgroup; we omit the proof.

Theorem 31.15: (Lagrange's theorem)

If $(S, \ )$ is a finite group and $(S', \ )$ is a subgroup of $(S, \ )$, then $|S'|$ is a divisor of $|S|$.

A subgroup S' of a group S is said to be a ***proper*** subgroup if $S' \neq S$. The following corollary will be used in our analysis of the Miller-Rabin primality test procedure in Section 31.8.

Corollary 31.16

If S' is a proper subgroup of a finite group S, then $|S'| \leq |S|/2$.

Subgroups generated by an element

Theorem 31.14 provides an interesting way to produce a subgroup of a finite group $(S, \ )$: choose an element a and take all elements that can be generated from a using the group operation. Specifically, define $a^{(k)}$ for $k \geq 1$ by

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k} .$$

For example, if we take $a = 2$ in the group $\mathbf{Z}_6$, the sequence $a^{(1)}$, $a^{(2)}$, ...is 2, 4, 0, 2, 4, 0, 2, 4, 0, ...

In the group $\mathbf{Z}_n$, we have $a^{(k)} = ka \bmod n$, and in the group $\mathbf{Z}_n^*$, we have $a^{(k)} = a^k \bmod n$. The ***subgroup generated by a***, denoted $\ a \ $ or $(\ a \ , \ )$, is defined by

$$a \ = \{a^{(k)} : k \geq 1\}.$$

We say that a ***generates*** the subgroup $\ a \ $ or that a is a ***generator*** of $\ a \ $. Since S is finite, $\ a \ $ is a finite subset of S, possibly including all of S. Since the associativity of $\ $ implies

$$a^{(i)} \ \ a^{(j)} = a^{(i+j)},$$

$\ a \ $ is closed and therefore, by Theorem 31.14, $\ a \ $ is a subgroup of S. For example, in $\mathbf{Z}_6$, we have

$$0 \ \ = \{0\} ,$$
$$1 \ \ = \{0,1,2,3,4,5\} ,$$
$$2 \ \ = \{0,2,4\} .$$

Similarly, in $\mathbf{Z}_7^*$, we have

1 = {1} ,
2 = {1,2,4} ,
3 3 = {1,2,3,4,5,6} .

The ***order*** of a (in the group S), denoted ord(a), is defined as the smallest positive integer t such that $a^{(t)} = e$.

Theorem 31.17

For any finite group (S,) and any a S, the order of an element is equal to the size of the subgroup it generates, or ord(a) = | a |.

Proof Let t = ord(a). Since $a^{(t)} = e$ and $a^{(t+k)} = a^{(t)}$ $a^{(k)} = a^{(k)}$ for $k \geq 1$, if $i \geq > t$, then $a^{(i)} >$ = for some $j < i$. Thus, no new elements are seen after $a^{(t)}$, so that a = $\{a^{(1)}, a^{(2)}, ..., a^{(t)}\}$ and | a | $\leq t$. To show that | a | $\geq t$, suppose for the purpose of contradiction that $a^{(i)} = a^{(j)}$ for some i, j satisfying $1 \leq i < j \leq t$. Then, $a^{(i+k)} = a^{(j+k)}$ for $k \geq 0$. But this implies that $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, a contradiction, since $i + (t - j) < t$ but t is the least positive value such that $a^{(t)} = e$. Therefore, each element of the sequence $a^{(1)}, a^{(2)}, ..., a^{(t)}$ is distinct, and | a | $\geq t$. We conclude that ord(a) = | a |.

Corollary 31.18

The sequence $a^{(1)}, a^{(2)}, ...$ is periodic with period t = ord(a); that is, $a^{(i)} = a^{(j)}$ if and only if $i \equiv j$ (mod t).

It is consistent with the above corollary to define $a^{(0)}$ as e and $a^{(i)}$ as $a^{(i \bmod t)}$, where t = ord(a), for all integers i.

Corollary 31.19

If (S,) is a finite group with identity e, then for all a S,

$a^{(|S|)} = e$.

Proof Lagrange's theorem implies that ord(a) | |S|, and so |S| $\equiv 0$ (mod t), where t = ord(a). Therefore, $a^{(|S|)} = a^{(0)} = e$.

Exercises 31.3-1

Draw the group operation tables for the groups $(\mathbf{Z}_4, +_4)$ and $(\mathbf{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence Σ between their elements such that $a + b \equiv c \pmod 4$ if and only if $\Sigma(a) \cdot \Sigma(b) \equiv \Sigma(c) \pmod 5$.

Exercises 31.3-2

Prove Theorem 31.14.

Exercises 31.3-3

Show that if p is prime and e is a positive integer, then

$\varphi(p^e) = p^{e-1}(p - 1)$.

Exercises 31.3-4

Show that for any $n > 1$ and for any $a \in \mathbf{Z}_n^*$, the function $f_a : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$ defined by $f_a(x) = ax \bmod n$ is a permutation of $\mathbf{Z}_n^*$.

Exercises 31.3-5

List all subgroups of $\mathbf{Z}_9$ and of $\mathbf{Z}_{13}^*$.

31.4 Solving modular linear equations

We now consider the problem of finding solutions to the equation

(31.21) $ax \equiv b \pmod n$,

where $a > 0$ and $n > 0$. There are several applications for this problem; for example, we will use it as part of the procedure for finding keys in the RSA public-key cryptosystem in Section 31.7. We assume that a, b, and n are given, and we are to find all values of x, modulo n, that satisfy equation (31.21). There may be zero, one, or more than one such solution.

Let $\langle a \rangle$ denote the subgroup of $\mathbf{Z}_n$ generated by a. Since $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, equation (31.21) has a solution if and only if $b \in \langle a \rangle$. Lagrange's theorem (Theorem 31.15) tells us that $|\langle a \rangle|$ must be a divisor of n. The following theorem gives us a precise characterization of $\langle a \rangle$.

Theorem 31.20

For any positive integers a and n, if $d = \gcd(a, n)$, then

(31.22) $\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}$,

in $\mathbf{Z}_n$, and thus

$|\ a\ | = n/d$.

Proof We begin by showing that d a . Recall that EXTENDED-EUCLID(a, n) produces integers x' and y' such that $ax' + ny' = d$. Thus, $ax' \equiv d \pmod{n}$, so that d a .

Since d a , it follows that every multiple of d belongs to a , because any multiple of a multiple of a is itself a multiple of a. Thus, a contains every element in $\{0, d, 2d, ..., ((n/d) - 1)d\}$. That is, d a .

We now show that a d . If m a , then $m = ax \bmod n$ for some integer x, and so $m = ax + ny$ for some integer y. However, $d \mid a$ and $d \mid n$, and so $d \mid m$ by equation (31.4). Therefore, m d .

Combining these results, we have that a = d . To see that $|\ a\ | = n/d$, observe that there are exactly n/d multiples of d between 0 and n - 1, inclusive.

Corollary 31.21

The equation $ax \equiv b \pmod{n}$ is solvable for the unknown x if and only if $\gcd(a, n) \mid b$.

Corollary 31.22

The equation $ax \equiv b \pmod{n}$ either has d distinct solutions modulo n, where $d = \gcd(a, n)$, or it has no solutions.

Proof If $ax \equiv b \pmod{n}$ has a solution, then b a . By Theorem 31.17, $\mathrm{ord}(a) = |\ a\ |$, and so Corollary 31.18 and Theorem 31.20 imply that the sequence $ai \bmod n$, for $i = 0, 1, ...$, is periodic with period $|\ a\ | = n/d$. If b a , then b appears exactly d times in the sequence $ai \bmod n$, for $i = 0, 1, ..., n - 1$, since the length-(n/d) block of values a is repeated exactly d times as i increases from 0 to n - 1. The indices x of the d positions for which $ax \bmod n = b$ are the solutions of the equation $ax \equiv b \pmod{n}$.

Theorem 31.23

Let $d = \gcd(a, n)$, and suppose that $d = ax' + ny'$ for some integers x' and y' (for example, as computed by EXTENDED-EUCLID). If $d \mid b$, then the equation $ax \equiv b \pmod{n}$ has as one of its solutions the value x_0, where

$x_0 = x'(b/d) \bmod n$.

Proof We have

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \quad \text{(because } ax' \equiv d \pmod{n}\text{)} \\ &\equiv b \pmod{n}, \end{aligned}$$

and thus x_0 is a solution to $ax \equiv b \pmod{n}$.

Theorem 31.24

Suppose that the equation $ax \equiv b \pmod{n}$ is solvable (that is, $d \mid b$, where $d = \gcd(a, n)$) and that x_0 is any solution to this equation. Then, this equation has exactly d distinct solutions, modulo n, given by $x_i = x_0 + i(n/d)$ for $i = 0, 1, \ldots, d - 1$.

Proof Since $n/d > 0$ and $0 \le i(n/d) < n$ for $i = 0, 1, \ldots, d - 1$, the values $x_0, x_1, \ldots, x_{d-1}$ are all distinct, modulo n. Since x_0 is a solution of $ax \equiv b \pmod{n}$, we have $ax_0 \bmod n = b$. Thus, for $i = 0, 1, \ldots, d - 1$, we have

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \qquad \text{(because } d \mid a\text{)} \\ &= b, \end{aligned}$$

and so x_i is a solution, too. By Corollary 31.22, there are exactly d solutions, so that $x_0, x_1, \ldots, x_{d-1}$ must be all of them.

We have now developed the mathematics needed to solve the equation $ax \equiv b \pmod{n}$; the following algorithm prints all solutions to this equation. The inputs a and n are arbitrary positive integers, and b is an arbitrary integer.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED-EUCLID(a, n)
2  if d | b
3      then x0 ← x'(b/d) mod n
4           for i ← 0 to d - 1
```

```
5               do print (x_0 + i(n/d)) mod n
6     else print "no solutions"
```

As an example of the operation of this procedure, consider the equation $14x \equiv 30 \pmod{100}$ (here, $a = 14$, $b = 30$, and $n = 100$). Calling EXTENDED-EUCLID in line 1, we obtain $(d, x, y) = (2, -7, 1)$. Since $2 \mid 30$, lines 3–5 are executed. In line 3, we compute $x_0 = (-7)(15) \bmod 100 = 95$. The loop on lines 4–5 prints the two solutions 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows. Line 1 computes $d = \gcd(a, n)$ as well as two values x' and y' such that $d = ax' + ny'$, demonstrating that x' is a solution to the equation $ax' \equiv d \pmod{n}$. If d does not divide b, then the equation $ax \equiv b \pmod{n}$ has no solution, by Corollary 31.21. Line 2 checks if $d \mid b$; if not, line 6 reports that there are no solutions. Otherwise, line 3 computes a solution x_0 to $ax \equiv b \pmod{n}$, in accordance with Theorem 31.23. Given one solution, Theorem 31.24 states that the other $d - 1$ solutions can be obtained by adding multiples of (n/d), modulo n. The **for** loop of lines 4–5 prints out all d solutions, beginning with x_0 and spaced (n/d) apart, modulo n.

MODULAR-LINEAR-EQUATION-SOLVER performs $O(\lg n + \gcd(a, n))$ arithmetic operations, since EXTENDED-EUCLID performs $O(\lg n)$ arithmetic operations, and each iteration of the **for** loop of lines 4–5 performs a constant number of arithmetic operations.

The following corollaries of Theorem 31.24 give specializations of particular interest.

Corollary 31.25

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv b \pmod{n}$ has a unique solution, modulo n.

If $b = 1$, a common case of considerable interest, the x we are looking for is a ***multiplicative inverse*** of a, modulo n.

Corollary 31.26

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv 1 \pmod{n}$ has a unique solution, modulo n. Otherwise, it has no solution.

Corollary 31.26 allows us to use the notation $(a^{-1} \bmod n)$ to refer to *the* multiplicative inverse of a, modulo n, when a and n are relatively prime. If $\gcd(a, n) = 1$, then one solution to the equation $ax \equiv 1 \pmod{n}$ is the integer x returned by EXTENDED-EUCLID, since the equation

$$\gcd(a, n) = 1 = ax + ny$$

implies $ax \equiv 1 \pmod{n}$. Thus, we can compute $(a^{-1} \bmod n)$ efficiently using EXTENDED-EUCLID.

Exercises 31.4-1

Find all solutions to the equation $35x \equiv 10 \pmod{50}$.

Exercises 31.4-2

Prove that the equation $ax \equiv ay \pmod{n}$ implies $x \equiv y \pmod{n}$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

Exercises 31.4-3

Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:

3 **then** $x_0 \leftarrow x'(b/d) \bmod (n/d)$

Will this work? Explain why or why not.

Exercises 31.4-4:

Let $f(x) \equiv f_0 + f_1 x + \cdots + f_t x^t \pmod{p}$ be a polynomial of degree t, with coefficients f_i drawn from $\mathbf{Z}_p$, where p is prime. We say that $a \in \mathbf{Z}_p$ is a ***zero*** of f if $f(a) \equiv 0 \pmod{p}$. Prove that if a is a zero of f, then $f(x) \equiv (x - a)g(x) \pmod{p}$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo a prime p.

31.5 The Chinese remainder theorem

Around A.D. 100, the Chinese mathematician Sun-Tsu solved the problem of finding those integers x that leave remainders 2, 3, and 2 when divided by 3, 5, and 7 respectively. One such solution is $x = 23$; all solutions are of the form $23 + 105k$ for arbitrary integers k. The "Chinese remainder theorem" provides a correspondence between a system of equations modulo a set of pairwise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

The Chinese remainder theorem has two major uses. Let the integer n be factored as $n = n_1 n_2 \ldots n_k$, where the factors n_i are pairwise relatively prime. First, the Chinese remainder theorem is a descriptive "structure theorem" that describes the structure of $\mathbf{Z}_n$ as identical to that of the Cartesian product $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ with componentwise addition and multiplication modulo n_i in the ith component. Second, this description can often be used to yield efficient algorithms, since working in each of the systems $\mathbf{Z}_{n_i}$ can be more efficient (in terms of bit operations) than working modulo n.

Theorem 31.27: (Chinese remainder theorem)

Let $n = n_1 n_2 \ldots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

(31.23) $$a \leftrightarrow (a_1, a_2, \ldots, a_k) ,$$

where $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$, and

$$a_i = a \bmod n_i$$

for $i = 1, 2, \ldots, k$. Then, mapping (31.23) is a one-to-one correspondence (bijection) between $\mathbf{Z}_n$ and the Cartesian product $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$. Operations performed on the elements of $\mathbf{Z}_n$ can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \ldots, a_k),$$

$$b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

then

(31.24) $$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) ,$$
(31.25) $$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) ,$$
(31.26) $$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k) .$$

Proof Transforming between the two representations is fairly straightforward. Going from a to $(a_1, a_2, \ldots, a_k)$ is quite easy and requires only k divisions. Computing a from inputs $(a_1, a_2, \ldots, a_k)$ is a bit more complicated, and is accomplished as follows. We begin by defining $m_i = n/n_i$ for $i = 1, 2, \ldots, k$; thus m_i is the product of all of the n_j's other than n_i: $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. We next define

(31.27) $$c_i = m_i (m_i^{-1} \bmod n_i)$$

for $i = 1, 2, \ldots, k$. Equation (31.27) is always well defined: since m_i and n_i are relatively prime (by Theorem 31.6), Corollary 31.26 guarantees that $(m_i^{-1} \bmod n_i)$ exists. Finally, we can compute a as a function of $a_1, a_2, \ldots, a_k$ as follows:

(31.28) $$a \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n} .$$

We now show that equation (31.28) ensures that $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. Note that if $j \neq i$, then $m_j \equiv 0 \pmod{n_i}$, which implies that $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Note also that $c_i \equiv 1 \pmod{n_i}$, from equation (31.27). We thus have the appealing and useful correspondence

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

a vector that has 0's everywhere except in the ith coordinate, where it has a 1; the c_i thus form a "basis" for the representation, in a certain sense. For each i, therefore, we have

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i}, \end{aligned}$$

which is what we wished to show: our method of computing a from the a_i's produces a result a that satisfies the constraints $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. The correspondence is one-to-one, since we can transform in both directions. Finally, equations (31.24)–(31.26) follow directly from Exercise 31.1-6, since $x \bmod n_i = (x \bmod n) \bmod n_i$ for any x and $i = 1, 2, \ldots, k$.

The following corollaries will be used later in this chapter.

Corollary 31.28

If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for any integers $a_1, a_2, \ldots, a_k$, the set of simultaneous equations

$$x \equiv a_i \pmod{n_i},$$

for $i = 1, 2, \ldots, k$, has a unique solution modulo n for the unknown x.

Corollary 31.29

If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for all integers x and a,

$$x \equiv a \pmod{n_i}$$

for $i = 1, 2, \ldots, k$ if and only if

$$x \equiv a \pmod{n}.$$

As an example of the application of the Chinese remainder theorem, suppose we are given the two equations

$a \equiv 2 \pmod 5$,
$a \equiv 3 \pmod{13}$,

so that $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, and $n_2 = m_1 = 13$, and we wish to compute a mod 65, since $n = 65$. Because $13^{-1} \equiv 2 \pmod 5$ and $5^{-1} \equiv 8 \pmod{13}$, we have

$c_1 = 13(2 \bmod 5) = 26$,
$c_2 = 5(8 \bmod 13) = 40$,

and

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figure 31.3: An illustration of the Chinese remainder theorem for $n_1 = 5$ and $n_2 = 13$. For this example, $c_1 = 26$ and $c_2 = 40$. In row i, column j is shown the value of a, modulo 65, such that (a mod 5) = i and (a mod 13) = j. Note that row 0, column 0 contains a 0. Similarly, row 4,column 12 contains a 64 (equivalent to -1). Since $c_1 = 26$, moving down a row increases a by 26. Similarly, $c_2 = 40$ means that moving right by a column increases a by 40. Increasing a by 1 corresponds to moving diagonally downward and to the right, wrapping around from the bottom to the top and from the right to the left.

Thus, we can work modulo n by working modulo n directly or by working in the transformed representation using separate modulo n_i computations, as convenient. The computations are entirely equivalent.

Exercises 31.5-1

Find all solutions to the equations $x \equiv 4 \pmod 5$ and $x \equiv 5 \pmod{11}$.

Exercises 31.5-2

Find all integers x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.

Exercises 31.5-3

Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)).$$

Exercises 31.5-4

Under the definitions of Theorem 31.27, prove that for any polynomial f, the number of roots of the equation $f(x) \equiv 0 \pmod n$ is equal to the product of the number of roots of each of the equations $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, ..., $f(x) \equiv 0 \pmod{n_k}$.

31.6 Powers of an element

Just as it is natural to consider the multiples of a given element a, modulo n, it is often natural to consider the sequence of powers of a, modulo n, where $a \in \mathbf{Z}_n^*$:

(31.29) $a^0, a^1, a^2, a^3, \ldots,$

modulo n. Indexing from 0, the 0th value in this sequence is $a^0 \bmod n = 1$, and the ith value is $a^i \bmod n$. For example, the powers of 3 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5

whereas the powers of 2 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4

In this section, let $\langle a \rangle$ denote the subgroup of $\mathbf{Z}_n^*$ generated by a by repeated multiplication, and let $\mathrm{ord}_n(a)$ (the "order of a, modulo n") denote the order of a in $\mathbf{Z}_n^*$. For example, $\langle 2 \rangle = \{1, 2, 4\}$ in $\mathbf{Z}_7^*$, and $\mathrm{ord}_7(2) = 3$. Using the definition of the Euler phi function $\phi(n)$ as the size of $\mathbf{Z}_n^*$ (see Section 31.3), we now translate Corollary 31.19 into the notation of $\mathbf{Z}_n^*$ to obtain Euler's theorem and specialize it to $\mathbf{Z}_p^*$, where p is prime, to obtain Fermat's theorem.

Theorem 31.30: (Euler's theorem)

For any integer $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbf{Z}_n^* .$$

Theorem 31.31: (Fermat's theorem)

If p is prime, then

$$a^{p-1} \equiv 1 \pmod{p} \text{ for all } a \in \mathbf{Z}_p^* .$$

Proof By equation (31.20), $\phi(p) = p - 1$ if p is prime.

This corollary applies to every element in $\mathbf{Z}_p$ except 0, since $0 \notin \mathbf{Z}_p^*$. For all $a \in \mathbf{Z}_p$, however, we have $a^p \equiv a \pmod{p}$ if p is prime.

If $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*|$, then every element in $\mathbf{Z}_n^*$ is a power of g, modulo n, and we say that g is a ***primitive root*** or a ***generator*** of $\mathbf{Z}_n^*$. For example, 3 is a primitive root, modulo 7, but 2 is not a primitive root, modulo 7. If $\mathbf{Z}_n^*$ possesses a primitive root, we say that the group $\mathbf{Z}_n^*$ is ***cyclic***. We omit the proof of the following theorem, which is proven by Niven and Zuckerman [231].

Theorem 31.32

The values of $n > 1$ for which $\mathbf{Z}_n^*$ is cyclic are 2, 4, p^e, and $2p^e$, for all primes $p > 2$ and all positive integers e.

If g is a primitive root of $\mathbf{Z}_n^*$ and a is any element of $\mathbf{Z}_n^*$, then there exists a z such that $g^z \equiv a \pmod{n}$. This z is called the ***discrete logarithm*** or ***index*** of a, modulo n, to the base g; we denote this value as $\mathrm{ind}_{n,g}(a)$.

Theorem 31.33: (Discrete logarithm theorem)

If g is a primitive root of $\mathbb{Z}_n^*$, then the equation $g^x \equiv g^y \pmod{n}$ holds if and only if the equation $x \equiv y \pmod{\varphi(n)}$ holds.

Proof Suppose first that $x \equiv y \pmod{\varphi(n)}$. Then, $x = y + k\varphi(n)$ for some integer k. Therefore,

$$\begin{aligned} g^x &\equiv g^{y+k\varphi(n)} \pmod{n} \\ &\equiv g^y \cdot (g^{\varphi(n)})^k \pmod{n} \\ &\equiv g^y \cdot 1^k \pmod{n} \quad \text{(by Euler's theorem)} \\ &\equiv g^y \pmod{n}. \end{aligned}$$

Conversely, suppose that $g^x \equiv g^y \pmod{n}$. Because the sequence of powers of g generates every element of $\langle g \rangle$ and $|\langle g \rangle| = \varphi(n)$, Corollary 31.18 implies that the sequence of powers of g is periodic with period $\varphi(n)$. Therefore, if $g^x \equiv g^y \pmod{n}$, then we must have $x \equiv y \pmod{\varphi(n)}$.

Taking discrete logarithms can sometimes simplify reasoning about a modular equation, as illustrated in the proof of the following theorem.

Theorem 31.34

If p is an odd prime and $e \geq 1$, then the equation

(31.30) $$x^2 \equiv 1 \pmod{p^e}$$

has only two solutions, namely $x = 1$ and $x = -1$.

Proof Let $n = p^e$. Theorem 31.32 implies that $\mathbb{Z}_n^*$ has a primitive root g. Equation (31.30) can be written

(31.31) $$(g^{\text{ind}_{n,g}(x)})^2 \equiv g^{\text{ind}_{n,g}(1)} \pmod{n}.$$

After noting that $\text{ind}_{n,g}(1) = 0$, we observe that Theorem 31.33 implies that equation (31.31) is equivalent to

(31.32) $$2 \cdot \text{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}.$$

To solve this equation for the unknown $\text{ind}_{n,g}(x)$, we apply the methods of Section 31.4. By equation (31.19), we have $\varphi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$. Letting $d = \gcd(2, \varphi(n)) = \gcd(2, (p - 1)p^{e-1}) = 2$, and noting that $d \mid 0$, we find from Theorem 31.24 that equation (31.32) has exactly $d = 2$ solutions. Therefore, equation (31.30) has exactly 2 solutions, which are $x = 1$ and $x = -1$ by inspection.

A number x is a ***nontrivial square root of 1, modulo n***, if it satisfies the equation $x^2 \equiv 1 \pmod{n}$ but x is equivalent to neither of the two "trivial" square roots: 1 or -1, modulo n. For example, 6 is a nontrivial square root of 1, modulo 35. The following corollary to Theorem 31.34 will be used in the correctness proof for the Miller-Rabin primality-testing procedure in Section 31.8.

Corollary 31.35

If there exists a nontrivial square root of 1, modulo n, then n is composite.

Proof By the contrapositive of Theorem 31.34, if there exists a nontrivial square root of 1, modulo n, then n can't be an odd prime or a power of an odd prime. If $x^2 \equiv 1 \pmod{2}$, then $x \equiv 1 \pmod{2}$, so all square roots of 1, modulo 2, are trivial. Thus, n cannot be prime. Finally, we must have $n > 1$ for a nontrivial square root of 1 to exist. Therefore, n must be composite.

Raising to powers with repeated squaring

A frequently occurring operation in number-theoretic computations is raising one number to a power modulo another number, also known as ***modular exponentiation***. More precisely, we would like an efficient way to compute $a^b \bmod n$, where a and b are nonnegative integers and n is a positive integer. Modular exponentiation is an essential operation in many primality-testing routines and in the RSA public-key cryptosystem. The method of ***repeated squaring*** solves this problem efficiently using the binary representation of b.

Let $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ be the binary representation of b. (That is, the binary representation is $k + 1$ bits long, b_k is the most significant bit, and b_0 is the least significant bit.) The following procedure computes $a^c \bmod n$ as c is increased by doublings and incrementations from 0 to b.

```
MODULAR-EXPONENTIATION(a, b, n)
 1  c ← 0
 2  d ← 1
 3  let ⟨b_k, b_k-1, ..., b_0⟩ be the binary representation of b
 4  for i ← k downto 0
 5        do c ← 2c
 6                d ← (d · d) mod n
 7                if b_i = 1
 8                          then c ← c + 1
 9                                d ← (d · a) mod n
10  return d
```

The essential use of squaring in line 6 of each iteration explains the name "repeated squaring." As an example, for $a = 7$, $b = 560$, and $n = 561$, the algorithm computes the sequence of values modulo 561 shown in Figure 31.4; the sequence of exponents used is shown in the row of the table labeled by c.

i	9	8	7	6	5	4	3	2	1	0

b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figure 31.4: The results of MODULAR-EXPONENTIATION when computing $a^b \pmod n$, where $a = 7$, $b = 560 = 1000110000$, and $n = 561$. The values are shown after each executi of the *for* loop. The final result is 1.

The variable c is not really needed by the algorithm but is included for explanatory purposes; the algorithm maintains the following two-part loop invariant:

Just prior to each iteration of the **for** loop of lines 4–9,

1. The value of c is the same as the prefix $b_k, b_{k-1}, ..., b_{i+1}$ of the binary representation of b, and
2. $d = a^c \bmod n$.

We use this loop invariant as follows:

- **Initialization:** Initially, $i = k$, so that the prefix $b_k, b_{k-1}, ..., b_{i+1}$ is empty, which corresponds to $c = 0$. Moreover, $d = 1 = a^0 \bmod n$.
- **Maintenance:** Let c' and d' denote the values of c and d at the end of an iteration of the **for** loop, and thus the values prior to the next iteration. Each iteration updates $c' \leftarrow 2c$ (if $b_i = 0$) or $c' \leftarrow 2c + 1$ (if $b_i = 1$), so that c will be correct prior to the next iteration. If $b_i = 0$, then $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$. If $b_i = 1$, then $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. In either case, $d = a^c \bmod n$ prior to the next iteration.
- **Termination:** At termination, $i = -1$. Thus, $c = b$, since c has the value of the prefix $b_k, b_{k-1}, ..., b_0$ of b's binary representation. Hence $d = a^c \bmod n = a^b \bmod n$.

If the inputs a, b, and n are β-bit numbers, then the total number of arithmetic operations required is $O(\beta)$ and the total number of bit operations required is $O(\beta^3)$.

Exercises 31.6-1

Draw a table showing the order of every element in $\mathbf{Z}_{11}^*$. Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbf{Z}_{11}^*$.

Exercises 31.6-2

Give a modular exponentiation algorithm that examines the bits of b from right to left instead of left to right.

Exercises 31.6-3

Assuming that you know $\varphi(n)$, explain how to compute $a^{-1} \bmod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

31.7 The RSA public-key cryptosystem

A public-key cryptosystem can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. A public-key cryptosystem also enables a party to append an unforgeable "digital signature" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that must be authenticated.

The RSA public-key cryptosystem is based on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers. Section 31.8 describes an efficient procedure for finding large prime numbers, and Section 31.9 discusses the problem of factoring large integers.

Public-key cryptosystems

In a public-key cryptosystem, each participant has both a ***public key*** and a ***secret key***. Each key is a piece of information. For example, in the RSA cryptosystem,each key consists of a pair of integers. The participants "Alice" and "Bob" are traditionally used in cryptography examples; we denote their public and secret keys as P_A, S_A for Alice and P_B, S_B for Bob.

Each participant creates his own public and secret keys. Each keeps his secret key secret, but he can reveal his public key to anyone or even publish it. In fact, it is often convenient to assume that everyone's public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

The public and secret keys specify functions that can be applied to any message. Let $\mathcal{D}$ denote the set of permissible messages. For example, $\mathcal{D}$ might be the set of all finite-length bit sequences. In the simplest, and original, formulation of public-key cryptography, we require that the public and secret keys specify one-to-one functions from $\mathcal{D}$ to itself. The function corresponding to Alice's public key P_A is denoted $P_A()$, and the function corresponding to her secret key S_A is denoted $S_A()$. The functions $P_A()$ and $S_A()$ are thus permutations of $\mathcal{D}$. We assume that the functions $P_A()$ and $S_A()$ are efficiently computable given the corresponding key P_A or S_A.

The public and secret keys for any participant are a "matched pair" in that they specify functions that are inverses of each other. That is,

(31.33) $M = S_A(P_A(M))$,
(31.34) $M = P_A(S_A(M))$

for any message $M \in \mathcal{D}$. Transforming M with the two keys P_A and S_A successively, in either order, yields the message M back.

In a public-key cryptosystem, it is essential that no one but Alice be able to compute the function $S_A()$ in any practical amount of time. The privacy of mail that is encrypted and sent to Alice and the authenticity of Alice's digital signatures rely on the assumption that only Alice able to compute $S_A()$. This requirement is why Alice keeps S_A secret; if she does not, she loses her uniqueness and the cryptosystem cannot provide her with unique capabilities. The umption that only Alice can compute $S_A()$ must hold even though everyone knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. The major difficulty in designing a workable public-key cryptosystem is in figuring out how to create a system in which we can reveal a transformation $P_A()$ without thereby revealing how to compute the corresponding inverse transformation $S_A()$.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Suppose Bob wishes to send Alice a message M encrypted so that it will look like unintelligible gibberish to an eavesdropper. The scenario for sending the message goes as follows.

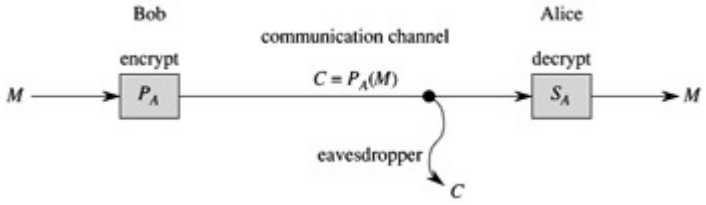

Figure 31.5: Encryption in a public key system. Bob encrypts the message M using Alice's public key P_A and transmits the resulting ciphertext $C = P_A(M)$ to Alice. An eavesdropper who captures the transmitted ciphertext gains no information about M. Alice receives C and decrypts it using her secret key to obtain the original message $M = S_A(C)$.

- Bob obtains Alice's public key P_A (from a public directory or directly from Alice).
- Bob computes the ***ciphertext*** $C = P_A(M)$ corresponding to the message M and sends C to Alice.
- When Alice receives the ciphertext C, she applies her secret key S_A to retrieve the original message: $M = S_A(C)$.

Because $S_A()$ and $P_A()$ are inverse functions, Alice can compute M from C. Because only Alice is able to compute $S_A()$, Alice is the only one who can compute M from C. The encryption of M using $P_A()$ has protected M from disclosure to anyone except Alice.

Digital signatures are similarly easy to implement within our formulation of a public-key cryptosystem. (We note that there are other ways of approaching the problem of constructing digital signatures, which we shall not go into here.) Suppose now that Alice wishes to send Bob a digitally signed response M'. The digital-signature scenario proceeds as shown in Figure 31.6.

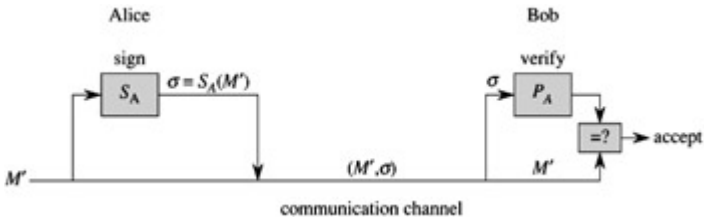

Figure 31.6: Digital signatures in a public-key system. Alice signs the message M' by appending her digital signature $\Sigma = S_A(M')$ to it. She transmits the message/signature pair (M', Σ) to Bob, who verifies it by checking the equation $M' = P_A(\Sigma)$. If the equation holds, he accepts (M', Σ) as a message that has been signed by Alice.

- Alice computes her ***digital signature*** Σ for the message M' using her secret key S_A and the equation $\Sigma = S_A(M')$.
- Alice sends the message/signature pair (M', Σ) to Bob.
- When Bob receives (M', Σ), he can verify that it originated from Alice by using Alice's public key to verify the equation $M' = P_A(\Sigma)$. (Presumably, M' contains Alice's name, so Bob knows whose public key to use.) If the equation holds, then Bob concludes that the message M' was actually signed by Alice. If the equation doesn't hold, Bob concludes either that the message M' or the digital signature Σ was corrupted by transmission errors or that the pair (M', Σ) is an attempted forgery.

Because a digital signature provides both authentication of the signer's identity and authentication of the contents of the signed message, it is analogous to a handwritten signature at the end of a written document.

An important property of a digital signature is that it is verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. For example, the message might be an electronic check from Alice to Bob. After Bob verifies Alice's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

We note that a signed message is not encrypted; the message is "in the clear" and is not protected from disclosure. By composing the above protocols for encryption and for signatures, we can create messages that are both signed and encrypted. The signer first appends his digital signature to the message and then encrypts the resulting message/signature pair with the public key of the intended recipient. The recipient decrypts the received message with his secret key to obtain both the original message and its digital signature. He can then verify the signature using the public key of the signer. The corresponding combined process using paper-based systems is to sign the paper document and then seal the document inside a paper envelope that is opened only by the intended recipient.

The RSA cryptosystem

In the ***RSA public-key cryptosystem***, a participant creates his public and secret keys with the following procedure.

1. Select at random two large prime numbers p and q such that $p \neq q$. The primes p and q might be, say, 512 bits each.
2. Compute n by the equation $n = pq$.

3. Select a small odd integer e that is relatively prime to $\varphi(n)$, which, by equation (31.19), equals $(p - 1)(q - 1)$.
4. Compute d as the multiplicative inverse of e, modulo $\varphi(n)$. (Corollary 31.26 guarantees that d exists and is uniquely defined. We can use the technique of Section 31.4 to compute d, given e and $\varphi(n)$.)
5. Publish the pair $P = (e, n)$ as his ***RSA public key***.
6. Keep secret the pair $S = (d, n)$ as his ***RSA secret key***.

For this scheme, the domain $\mathcal{D}$ is the set $\mathbf{Z}_n$. The transformation of a message M associated with a public key $P = (e, n)$ is

(31.35) $$P(M) = M^e \pmod{n}.$$

The transformation of a ciphertext C associated with a secret key $S = (d, n)$ is

(31.36) $$S(C) = C^d \pmod{n}.$$

These equations apply to both encryption and signatures. To create a signature, the signer applies his secret key to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to it, rather than to a message to be encrypted.

The public-key and secret-key operations can be implemented using the procedure MODULAR-EXPONENTIATION described in Section 31.6. To analyze the running time of these operations, assume that the public key (e, n) and secret key (d, n) satisfy $\lg e = O(1)$, $\lg d \leq \beta$, and $\lg n \leq \beta$. Then, applying a public key requires $O(1)$ modular multiplications and uses $O(\beta^2)$ bit operations. Applying a secret key requires $O(\beta)$ modular multiplications, using $O(\beta^3)$ bit operations.

Theorem 31.36: (Correctness of RSA)

The RSA equations (31.35) and (31.36) define inverse transformations of $\mathbf{Z}_n$ satisfying equations (31.33) and (31.34).

Proof From equations (31.35) and (31.36), we have that for any $M \in \mathbf{Z}_n$,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}.$$

Since e and d are multiplicative inverses modulo $\varphi(n) = (p - 1)(q - 1)$,

$$ed = 1 + k(p - 1)(q - 1)$$

for some integer k. But then, if $M \not\equiv 0 \pmod{p}$, we have

$$\begin{aligned} M^{ed} &\equiv M\,(M^{P-1})^{k(q-1)} \pmod{p} \\ &\equiv M(1)^{k(q-1)} \pmod{p} \quad \text{(by Theorem 31.31)} \\ &\equiv M \pmod{p}. \end{aligned}$$

Also, $M^{ed} \equiv M \pmod{p}$ if $M \equiv 0 \pmod{p}$. Thus,

$$M^{ed} \equiv M \pmod{p}$$

for all M. Similarly,

$$M^{ed} \equiv M \pmod{q}$$

for all M. Thus, by Corollary 31.29 to the Chinese remainder theorem,

$$M^{ed} \equiv M \pmod{n}$$

for all M.

The security of the RSA cryptosystem rests in large part on the difficulty of factoring large integers. If an adversary can factor the modulus n in a public key, then he can derive the secret key from the public key, using the knowledge of the factors p and q in the same way that the creator of the public key used them. So if factoring large integers is easy, then breaking the RSA cryptosystem is easy. The converse statement, that if factoring large integers is hard, then breaking RSA is hard, is unproven. After two decades of research, however, no easier method has been found to break the RSA public-key cryptosystem than to factor the modulus n. And as we shall see in Section 31.9, the factoring of large integers is surprisingly difficult. By randomly selecting and multiplying together two 512-bit primes, one can create a public key that cannot be "broken" in any feasible amount of time with current technology. In the absence of a fundamental breakthrough in the design of number-theoretic algorithms, and when implemented with care following recommended standards, the RSA cryptosystem is capable of providing a high degree of security in applications.

In order to achieve security with the RSA cryptosystem, however, it is advisable to work with integers that are several hundred bits long, to resist possible advances in the art of factoring. At the time of this writing (2001), RSA moduli were commonly in the range of 768 to 2048 bits. To create moduli of such sizes, we must be able to find large primes efficiently. Section 31.8 addresses this problem.

For efficiency, RSA is often used in a "hybrid" or "key-management" mode with fast non-public-key cryptosystems. With such a system, the encryption and decryption keys are identical. If Alice wishes to send a long message M to Bob privately, she selects a random key K for the fast non-public-key cryptosystem and encrypts M using K , obtaining ciphertext C. Here, C is as long as M, but K is quite short. Then, she encrypts K using Bob's public RSA key. Since K is short, computing $P_B(K)$ is fast (much faster than computing $P_B(M)$). She then transmits $(C, P_B(K))$ to Bob, who decrypts $P_B(K)$ to obtain K and then uses K to decrypt C, obtaining M.

A similar hybrid approach is often used to make digital signatures efficiently. In this approach, RSA is combined with a public ***one-way hash function*** h—a function that is easy to compute but for which it is computationally infeasible to find two messages M and M' such that $h(M) = h(M')$. The value $h(M)$ is a short (say, 160-bit) "fingerprint" of the message M. If

Alice wishes to sign a message M, she first applies h to M to obtain the fingerprint $h(M)$, which she then encrypts with her secret key. She sends $(M, S_A(h(M)))$ to Bob as her signed version of M. Bob can verify the signature by computing $h(M)$ and verifying that P_A applied to $S_A(h(M))$ as received equals $h(M)$. Because no one can create two messages with the same fingerprint, it is computationally infeasible to alter a signed message and preserve the validity of the signature.

Finally, we note that the use of ***certificates*** makes distributing public keys much easier. For example, assume there is a "trusted authority" T whose public key is known by everyone. Alice can obtain from T a signed message (her certificate) stating that "Alice's public key is P_A." This certificate is "self-authenticating" since everyone knows P_T. Alice can include her certificate with her signed messages, so that the recipient has Alice's public key immediately available in order to verify her signature. Because her key was signed by T, the recipient knows that Alice's key is really Alice's.

Exercises 31.7-1

Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

Exercises 31.7-2

Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n. (Although you are not asked to prove it, you may be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [221].)

Exercises 31.7-3:

Prove that RSA is multiplicative in the sense that

$$P_A(M_1)\,P_A(M_2) \equiv P_A(M_1\,M_2) \pmod{n}.$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1 percent of messages from $\mathbf{Z}_n$ encrypted with P_A, then he could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

31.8 Primality testing

In this section, we consider the problem of finding large primes. We begin with a discussion of the density of primes, proceed to examine a plausible (but incomplete) approach to primality testing, and then present an effective randomized primality test due to Miller and Rabin.

The density of prime numbers

For many applications (such as cryptography), we need to find large "random" primes. Fortunately, large primes are not too rare, so that it is not too time-consuming to test random integers of the appropriate size until a prime is found. The ***prime distribution function*** $\pi(n)$ specifies the number of primes that are less than or equal to n. For example, $\pi(10) = 4$, since there are 4 prime numbers less than or equal to 10, namely, 2, 3, 5, and 7. The prime number theorem gives a useful approximation to $\pi(n)$.

Theorem 31.37: (Prime number theorem)

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1 .$$

The approximation $n/\ln n$ gives reasonably accurate estimates of $\pi(n)$ even for small n. For example, it is off by less than 6% at $n = 109$, where $\pi(n) = 50{,}847{,}534$ and $n/\ln n \approx 48{,}254{,}942$. (To a number theorist, 10^9 is a small number.)

We can use the prime number theorem to estimate the probability that a randomly chosen integer n will turn out to be prime as $1/\ln n$. Thus, we would need to examine approximately $\ln n$ integers chosen randomly near n in order to find a prime that is of the same length as n. For example, to find a 512-bit prime might require testing approximately $\ln 2^{512} \approx 355$ randomly chosen 512-bit numbers for primality. (This figure can be cut in half by choosing only odd integers.)

In the remainder of this section, we consider the problem of determining whether or not a large odd integer n is prime. For notational convenience, we assume that n has the prime factorization

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.37}$$

where $r \geq 1$, $p_1, p_2, \ldots, p_r$ are the prime factors of n, and $e_1, e_2, \ldots, e_r$ are positive integers. Of course, n is prime if and only if $r = 1$ and $e_1 = 1$.

One simple approach to the problem of testing for primality is ***trial division***. We try dividing n by each integer 2, 3, ..., $\lfloor \sqrt{n} \rfloor$. (Again, even integers greater than 2 may be skipped.) It is easy to see that n is prime if and only if none of the trial divisors divides n. Assuming that each trial division takes constant time, the worst-case running time is $\Theta(\sqrt{n})$, which is exponential in the length of n. (Recall that if n is encoded in binary using β bits, then $\beta = \lceil \lg(n + 1) \rceil$, and so $\sqrt{n} = \Theta(2^{\beta/2})$.) Thus, trial division works well only if n is very small or happens to have a small prime factor. When it works, trial division has the advantage that it not only determines whether n is prime or composite, but also determines one of n's prime factors if n is composite.

In this section, we are interested only in finding out whether a given number n is prime; if n is composite, we are not concerned with finding its prime factorization. As we shall see in Section 31.9, computing the prime factorization of a number is computationally expensive. It

is perhaps surprising that it is much easier to tell whether or not a given number is prime than it is to determine the prime factorization of the number if it is not prime.

Pseudoprimality testing

We now consider a method for primality testing that "almost works" and in fact is good enough for many practical applications. A refinement of this method that removes the small defect will be presented later. Let $\mathbf{Z}_n^+$ denote the nonzero elements of $\mathbf{Z}_n$:

$$\mathbf{Z}_n^+ = \{1, 2, \ldots, n-1\} .$$

If n is prime, then $\mathbf{Z}_n^+ = \mathbf{Z}_n^*$.

We say that n is a ***base-a pseudoprime*** if n is composite and

(31.38) $$a^{n-1} \equiv 1 \pmod{n} .$$

Fermat's theorem (Theorem 31.31) implies that if n is prime, then n satisfies equation (31.38) for every a in $\mathbf{Z}_n^+$. Thus, if we can find any $a \in \mathbf{Z}_n^+$ such that n does *not* satisfy equation (31.38), then n is certainly composite. Surprisingly, the converse *almost* holds, so that this criterion forms an almost perfect test for primality. We test to see if n satisfies equation (31.38) for $a = 2$. If not, we declare n to be composite. Otherwise, we output a guess that n is prime (when, in fact, all we know is that n is either prime or a base-2 pseudoprime).

The following procedure pretends in this manner to be checking the primality of n. It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. The input n is assumed to be an odd integer greater than 2.

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2 = n - 1, n) ≠ 1 (mod n)
2     then return COMPOSITE          ▹ Definitely.
3     else return PRIME                    ▹ We hope!
```

This procedure can make errors, but only of one type. That is, if it says that n is composite, then it is always correct. If it says that n is prime, however, then it makes an error only if n is a base-2 pseudoprime.

How often does this procedure err? Surprisingly rarely. There are only 22 values of n less than 10,000 for which it errs; the first four such values are 341, 561, 645, and 1105. It can be shown that the probability that this program makes an error on a randomly chosen β-bit number goes to zero as $\beta \leftarrow \infty$. Using more precise estimates due to Pomerance [244] of the number of base-2 pseudoprimes of a given size, we may estimate that a randomly chosen 512-bit number that is called prime by the above procedure has less than one chance in 10^{20} of being a base-2 pseudoprime, and a randomly chosen 1024-bit number that is called prime has less than one chance in 10^{41} of being a base-2 pseudoprime. So if you are merely trying to find a large prime for some application, for all practical purposes you almost never go wrong by choosing large numbers at random until one of them causes PSEUDOPRIME to output PRIME. But when the numbers being tested for primality are not randomly chosen, we need a better approach for testing primality. As we shall see, a little more cleverness, and some randomization, will yield a primality-testing routine that works well on all inputs.

Unfortunately, we cannot entirely eliminate all the errors by simply checking equation (31.38) for a second base number, say $a = 3$, because there are composite integers n that satisfy equation (31.38) for *all* $a \in \mathbb{Z}_n^*$. These integers are known as ***Carmichael numbers***. The first three Carmichael numbers are 561, 1105, and 1729. Carmichael numbers are extremely rare; there are, for example, only 255 of them less than 100,000,000. Exercise 31.8-2 helps explain why they are so rare.

We next show how to improve our primality test so that it won't be fooled by Carmichael numbers.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple test PSEUDOPRIME with two modifications:

- It tries several randomly chosen base values a instead of just one base value.
- While computing each modular exponentiation, it notices if a nontrivial square root of 1, modulo n, is discovered during the final set of squarings. If so, it stops and outputs COMPOSITE. Corollary 31.35 justifies detecting composites in this manner.

The pseudocode for the Miller-Rabin primality test follows. The input $n > 2$ is the odd number to be tested for primality, and s is the number of randomly chosen base values from $\mathbb{Z}_n^*$ to be tried. The code uses the random-number generator RANDOM described on page 94: RANDOM(1, n - 1) returns a randomly chosen integer a satisfying $1 \le a \le n - 1$. The code uses an auxiliary procedure WITNESS such that WITNESS(a, n) is TRUE if and only if a is a "witness" to the compositeness of n—that is, if it is possible using a to prove (in a manner that we shall see) that n is composite. The test WITNESS(a, n) is an extension of, but more effective than, the test

$$a^{n-1} \not\equiv 1 \pmod{n}$$

that formed the basis (using $a = 2$) for PSEUDOPRIME. We first present and justify the construction of WITNESS, and then show how it is used in the Miller-Rabin primality test. Let $n - 1 = 2^t u$ where $t \ge 1$ and u is odd; i.e., the binary representation of n - 1 is the binary representation of the odd integer u followed by exactly t zeros. Therefore, $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, so that we can compute $a^{n-1} \bmod n$ by first computing $a^u \bmod n$ and then squaring the result t times successively.

```
WITNESS(a, n)
1  let n - 1 = 2^t u, where t ≥ 1 and u is odd
2  x_0 ← MODULAR-EXPONENTIATION(a, u, n)
3  for i ← 1 to t
4        do x_i ← x_{i-1}^2 mod n
5           if x_i = 1 and x_{i-1} ≠ 1 and x_{i-1} ≠ n - 1
6              then return TRUE
7  if x_t ≠ 1
8     then return TRUE
9  return FALSE
```

This pseudocode for WITNESS computes $a^{n-1} \bmod n$ by first computing the value $x_0 = a^u \bmod n$ in line 2, and then squaring the result t times in a row in the **for** loop of lines 3–6. By

induction on i, the sequence $x_0, x_1, \ldots, x_t$ of values computed satisfies the equation $x_i \equiv a^{2^i q} \pmod{n}$ for $i = 0, 1, \ldots, t$, so that in particular $x_t \not\equiv a^{n-1} \pmod{n}$. Whenever a squaring step is performed on line 4, however, the loop may terminate early if lines 5–6 detect that a nontrivial square root of 1 has just been discovered. If so, the algorithm stops and returns TRUE. Lines 7–8 return TRUE if the value computed for $x_t \not\equiv a^{n-1} \pmod{n}$ is not equal to 1, just as the PSEUDOPRIME procedure returns COMPOSITE in this case. Line 9 returns FALSE if we haven't returned TRUE in lines 6 or 8.

We now argue that if WITNESS(a, n) returns TRUE, then a proof that n is composite can be constructed using a.

If WITNESS returns TRUE from line 8, then it has discovered that $x_t = a^{n-1} \bmod n \neq 1$. If n is prime, however, we have by Fermat's theorem (Theorem 31.31) that $a^{n-1} \not\equiv 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^*$. Therefore, n cannot be prime, and the equation $a^{n-1} \bmod n \neq 1$ is a proof of this fact.

If WITNESS returns TRUE from line 6, then it has discovered that x_{i-1} is a nontrivial square root of $x_i = 1$, modulo n, since we have that $x_{i-1} \not\equiv \pm 1 \pmod{n}$ yet $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$. Corollary 31.35 states that only if n is composite can there be a nontrivial square root of 1 modulo n, so that a demonstration that x_{i-1} is a nontrivial square root of 1 modulo n is a proof that n is composite.

This completes our proof of the correctness of WITNESS. If the invocation WITNESS(a, n) outputs TRUE, then n is surely composite, and a proof that n is composite can be easily determined from a and n.

At this point we briefly present an alternative description of the behavior of WITNESS as a function of the sequence $X = \langle x_0, x_1, \ldots, x_t \rangle$, which you may find useful later on, when we analyze the efficiency of the Miller-Rabin primality test. Note that if $x_i = 1$ for some $0 \le i < t$, WITNESS might not compute the rest of the sequence. If it were to do so, however, each value $x_{i+1}, x_{i+2}, \ldots, x_t$ would be 1, and we consider these positions in the sequence X as being all 1's. We have four cases:

1. $X = \langle \ldots, d \rangle$, where $d \neq 1$: the sequence X does not end in 1. Return TRUE; a is a witness to the compositeness of n (by Fermat's Theorem).
2. $X = \langle 1, 1, \ldots, 1 \rangle$: the sequence X is all 1's. Return FALSE; a is not a witness to the compositeness of n.
3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: the sequence X ends in 1, and the last non-1 is equal to -1. Return FALSE; a is not a witness to the compositeness of n.
4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, where $d \neq \pm 1$: the sequence X ends in 1, but the last non-1 is not -1. Return TRUE; a is a witness to the compositeness of n, since d is a nontrivial square root of 1.

We now examine the Miller-Rabin primality test based on the use of WITNESS. Again, we assume that n is an odd integer greater than 2.

```
MILLER-RABIN(n, s)
1  for j ← 1 to s
2       do a ← RANDOM(1, n - 1)
3          if WITNESS(a, n)
```

```
4            then return COMPOSITE    ▷ Definitely.
5  return PRIME                       ▷ Almost surely.
```

The procedure MILLER-RABIN is a probabilistic search for a proof that n is composite. The main loop (beginning on line 1) picks s random values of a from $\mathbb{Z}_n^+$(line 2). If one of the a's picked is a witness to the compositeness of n, then MILLER-RABIN outputs COMPOSITE on line 4. Such an output is always correct, by the correctness of WITNESS. If no witness can be found in s trials, MILLER-RABIN assumes that this is because there are no witnesses to be found, and therefore n is assumed to be prime. We shall see that this output is likely to be correct if s is large enough, but that there is a small chance that the procedure may be unlucky in its choice of a's and that witnesses do exist even though none has been found.

To illustrate the operation of MILLER-RABIN, let n be the Carmichael number 561, so that $n - 1 = 560 = 2^4 \cdot 35$. Supposing that $a = 7$ is chosen as a base, Figure 31.4 shows that WITNESS computes $x_0 = a^{35} = 241 \pmod{561}$ and thus computes the sequence $X = \langle 241, 298, 166, 67, 1 \rangle$. Thus, a nontrivial square root of 1 is discovered in the last squaring step, since $a^{280} \equiv 67 \pmod{n}$ and $a^{560} \equiv 1 \pmod{n}$. Therefore, $a = 7$ is a witness to the compositeness of n, WITNESS(7, n) returns TRUE, and MILLER-RABIN returns COMPOSITE.

If n is a β-bit number, MILLER-RABIN requires $O(s\beta)$ arithmetic operations and $O(s\beta^3)$ bit operations, since it requires asymptotically no more work than s modular exponentiations.

Error rate of the Miller-Rabin primality test

If MILLER-RABIN outputs PRIME, then there is a small chance that it has made an error. Unlike PSEUDOPRIME, however, the chance of error does not depend on n; there are no bad inputs for this procedure. Rather, it depends on the size of s and the "luck of the draw" in choosing base values a. Also, since each test is more stringent than a simple check of equation (31.38), we can expect on general principles that the error rate should be small for randomly chosen integers n. The following theorem presents a more precise argument.

Theorem 31.38

If n is an odd composite number, then the number of witnesses to the compositeness of n is at least $(n - 1)/2$.

Proof The proof shows that the number of nonwitnesses is at most $(n - 1)/2$, which implies the theorem.

We start by claiming that any nonwitness must be a member of $\mathbb{Z}_n^*$. Why? Consider any nonwitness a. It must satisfy $a^{n-1} \equiv 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Thus, there is a solution to the equation $ax \equiv 1 \pmod{n}$, namely a^{n-2}. By Corollary 31.21, $\gcd(a, n) \mid 1$, which in turn implies that $\gcd(a, n) = 1$. Therefore, a is a member of $\mathbb{Z}_n^*$; all nonwitnesses belong to $\mathbb{Z}_n^*$.

To complete the proof, we show that not only are all nonwitnesses contained in $\mathbb{Z}_n^*$, they are all contained in a proper subgroup B of $\mathbb{Z}_n^*$(recall that we say B is a *proper* subgroup of $\mathbb{Z}_n^*$when B is subgroup of $\mathbb{Z}_n^*$but B is not equal to $\mathbb{Z}_n^*$). By Corollary 31.16, we then have $|B| \le |\mathbb{Z}_n^*|/2$. Since

$|\mathbf{Z}_n^*| \leq n - 1$, we obtain $|B| \leq (n - 1)/2$. Therefore, the number of nonwitnesses is at most $(n - 1)/2$, so that the number of witnesses must be at least $(n - 1)/2$.

We now show how to find a proper subgroup B of $\mathbf{Z}_n^*$ containing all of the nonwitnesses. We break the proof into two cases.

Case 1: There exists an $x \in \mathbf{Z}_n^*$ such that

$x^{n-1} \not\equiv 1 \pmod{n}$.

In other words, n is not a Carmichael number. Because, as we noted earlier, Carmichael numbers are extremely rare, case 1 is the main case that arises "in practice" (e.g., when n has been chosen randomly and is being tested for primality).

Let $B = \{b \in \mathbf{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. Clearly, B is nonempty, since $1 \quad B$. Since B is closed under multiplication modulo n, we have that B is a subgroup of $\mathbf{Z}_n^*$ by Theorem 31.14. Note that every nonwitness belongs to B, since a nonwitness a satisfies $a^{n-1} \equiv 1 \pmod{n}$. Since $x \in \mathbf{Z}_n^* - B$, we have that B is a proper subgroup of $\mathbf{Z}_n^*$.

Case 2: For all $x \in \mathbf{Z}_n^*$,

(31.39) $x^{n-1} \equiv 1 \pmod{n}$.

In other words, n is a Carmichael number. This case is extremely rare in practice. However, the Miller-Rabin test (unlike a pseudo-primality test) can efficiently determine the compositeness of Carmichael numbers, as we now show.

In this case, n cannot be a prime power. To see why, let us suppose to the contrary that $n = p^e$, where p is a prime and $e > 1$. We derive a contradiction as follows. Since n is assumed to be odd, p must also be odd. Theorem 31.32 implies that $\mathbf{Z}_n^*$ is a cyclic group: it contains a generator g such that $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$. By equation (31.39), we have $g^{n-1} \equiv 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33, taking $y = 0$) implies that $n - 1 \equiv 0 \pmod{\varphi(n)}$, or

$(p - 1)p^{e-1} \mid p^e - 1$.

This is a contradiction for $e > 1$, since $(p - 1)\,p^e$-1 is divisible by the prime p but $p^e - 1$ is not. Thus, n is not a prime power.

Since the odd composite number n is not a prime power, we decompose it into a product $n_1 n_2$, where n_1 and n_2 are odd numbers greater than 1 that are relatively prime to each other. (There may be several ways to do this, and it doesn't matter which one we choose. For example, if $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we can choose $n_1 = p_1^{e_1}$ and $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Recall that we define t and u so that $n - 1 = 2^t u$, where $t \geq 1$ and u is odd, and that for an input a, the procedure WITNESS computes the sequence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

(all computations are performed modulo n).

Let us call a pair (v, j) of integers ***acceptable*** if $v \in \mathbf{Z}_n^*$, j $\{0, 1, ..., t\}$, and

$v^{2^j u} \equiv -1 \pmod{n}$.

Acceptable pairs certainly exist since u is odd; we can choose $v = n - 1$ and $j = 0$, so that $(n - 1, 0)$ is an acceptable pair. Now pick the largest possible j such that there exists an acceptable pair (v, j), and fix v so that (v, j) is an acceptable pair. Let

$$B = \{x \in \mathbf{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Since B is closed under multiplication modulo n, it is a subgroup of $\mathbf{Z}_n^*$. By Corollary 31.16, therefore, $|B|$ divides $|\mathbf{Z}_n^*|$. Every nonwitness must be a member of B, since the sequence X produced by a nonwitness must either be all 1's or else contain a -1 no later than the jth position, by the maximality of j. (If (a, j') is acceptable, where a is a nonwitness, we must have $j' \le j$ by how we chose j.)

We now use the existence of v to demonstrate that there exists a $w \in \mathbf{Z}_n^* - B$. Since $v^{2^j u} \equiv -1 \pmod{n}$, we have $v^{2^j u} \equiv -1 \pmod{n_1}$ by Corollary 31.29 to the Chinese remainder theorem. By Corollary 31.28, there is a w simultaneously satisfying the equations

$w \equiv v \pmod{n_1}$,
$w \equiv 1 \pmod{n_2}$.

Therefore,

$w^{2^j u} \equiv -1 \pmod{n_1}$,
$w^{2^j u} \equiv 1 \pmod{n_2}$.

By Corollary 31.29, $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implies $w^{2^j u} \not\equiv 1 \pmod{n}$, and $w^{2^j u} \not\equiv -1 \pmod{n_2}$ implies $w^{2^j u} \not\equiv -1 \pmod{n}$. Hence, $w^{2^j u} \not\equiv \pm 1 \pmod{n}$, and so $w \notin B$.

It remains to show that $w \in \mathbf{Z}_n^*$, which we do by first working separately modulo n_1 and modulo n_2. Working modulo n_1, we observe that since $v \in \mathbf{Z}_n^*$, we have that $\gcd(v, n) = 1$, and so also $\gcd(v, n_1) = 1$; if v doesn't have any common divisors with n, it certainly doesn't have any common divisors with n_1. Since $w \equiv v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo n_2, we observe that $w \equiv 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$. To combine these results, we use Theorem 31.6, which implies that $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbf{Z}_n^*$.

Therefore $w \in \mathbf{Z}_n^* - B$, and we finish case 2 with the conclusion that B is a proper subgroup of $\mathbf{Z}_n^*$.

In either case, we see that the number of witnesses to the compositeness of n is at least $(n - 1)/2$.

Theorem 31.39

For any odd integer $n > 2$ and positive integer s, the probability that MILLER-RABIN(n, s) errs is at most 2^{-s}.

Proof Using Theorem 31.38, we see that if n is composite, then each execution of the **for** loop of lines 1–4 has a probability of at least 1/2 of discovering a witness x to the compositeness of n. MILLER-RABIN makes an error only if it is so unlucky as to miss discovering a witness to the compositeness of n on each of the s iterations of the main loop. The probability of such a string of misses is at most 2^{-s}.

Thus, choosing $s = 50$ should suffice for almost any imaginable application. If we are trying to find large primes by applying MILLER-RABIN to *randomly chosen* large integers, then it can be argued (although we won't do so here) that choosing a small value of s (say 3) is very unlikely to lead to erroneous results. That is, for a randomly chosen odd composite integer n, the expected number of nonwitnesses to the compositeness of n is likely to be much smaller than $(n - 1)/2$. If the integer n is not chosen randomly, however, the best that can be proven is that the number of nonwitnesses is at most $(n - 1)/4$, using an improved version of Theorem 31.39. Furthermore, there do exist integers n for which the number of nonwitnesses is $(n - 1)/4$.

Exercises 31.8-1

Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1 modulo n.

Exercises 31.8-2:

It is possible to strengthen Euler's theorem slightly to the form

$a^{\lambda(n)} \equiv 1 \pmod{n}$ for all $a \in \mathbf{Z}_n^*$,

where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

(31.40) $\lambda(n) = \operatorname{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r}))$.

Prove that $\lambda(n) \mid \varphi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n - 1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$; here, $\lambda(n) = \operatorname{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. For this reason, they are not very common.

Exercises 31.8-3

Prove that if x is a nontrivial square root of 1, modulo n, then gcd(x - 1, n) and gcd(x + 1, n) are both nontrivial divisors of n.

31.9 Integer factorization

Suppose we have an integer n that we wish to ***factor***, that is, to decompose into a product of primes. The primality test of the preceding section would tell us that n is composite, but it usually doesn't tell us the prime factors of n. Factoring a large integer n seems to be much more difficult than simply determining whether n is prime or composite. It is infeasible with today's supercomputers and the best algorithms to date to factor an arbitrary 1024-bit number.

Pollard's rho heuristic

Trial division by all integers up to B is guaranteed to factor completely any number up to B^2. For the same amount of work, the following procedure will factor any number up to B^4 (unless we're unlucky). Since the procedure is only a heuristic, neither its running time nor its success is guaranteed, although the procedure is very effective in practice. Another advantage of the POLLARD-RHO procedure is that it uses only a constant number of memory locations. (You can easily implement Pollard-Rho on a programmable pocket calculator to find factors of small numbers.)

```
POLLARD-RHO(n)
 1  i ← 1
 2  x1 ← RANDOM(0, n - 1)
 3  y ← x1
 4  k ← 2
 5  while TRUE
 6      do i ← i + 1
 7         xi ← (xi-1^2 − 1) mod n
 8         d ← gcd(y - xi, n)
 9         if d ≠ 1 and d ≠ n
10            then print d
11         if i = k
12            then y ← xi
13                 k ← 2k
```

The procedure works as follows. Lines 1–2 initialize i to 1 and x_1 to a randomly chosen value in $\mathbf{Z}_n$. The **while** loop beginning on line 5 iterates forever, searching for factors of n. During each iteration of the **while** loop, the recurrence

(31.41) $$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n$$

is used on line 7 to produce the next value of x_i in the infinite sequence

(31.42) $$x_1, x_2, x_3, x_4, \ldots ;$$

the value of i is correspondingly incremented on line 6. The code is written using subscripted variables x_i for clarity, but the program works the same if all of the subscripts are dropped, since only the most recent value of x_i need be maintained. With this modification, the procedure uses only a constant number of memory locations.

Every so often, the program saves the most recently generated x_i value in the variable y. Specifically, the values that are saved are the ones whose subscripts are powers of 2:

$x_1, x_2, x_4, x_8, x_{16}, \ldots .$

Line 3 saves the value x_1, and line 12 saves x_k whenever i is equal to k. The variable k is initialized to 2 in line 4, and k is doubled in line 13 whenever y is updated. Therefore, k follows the sequence 1, 2, 4, 8, ... and always gives the subscript of the next value x_k to be saved in y.

Lines 8–10 try to find a factor of n, using the saved value of y and the current value of x_i . Specifically, line 8 computes the greatest common divisor $d = \gcd(y - x_i, n)$. If d is a nontrivial divisor of n (checked in line 9), then line 10 prints d.

This procedure for finding a factor may seem somewhat mysterious at first. Note, however, that POLLARD-RHO never prints an incorrect answer; any number it prints is a nontrivial divisor of n. POLLARD-RHO may not print anything at all, though; there is no guarantee that it will produce any results. We shall see, however, that there is good reason to expect POLLARD-RHO to print a factor p of n after $\Theta(\sqrt{p})$ iterations of the **while** loop. Thus, if **n** is composite, we can expect this procedure to discover enough divisors to factor n completely after approximately $n^{1/4}$ updates, since every prime factor p of n except possibly the largest one is less than $\sqrt{n}$.

We begin our analysis of the behavior of this procedure by studying how long it takes a random sequence modulo n to repeat a value. Since $\mathbf{Z}_n$ is finite, and since each value in the sequence (31.42) depends only on the previous value, the sequence (31.42) eventually repeats itself. Once we reach an x_i such that $x_i = x_j$ for some $j < i$, we are in a cycle, since $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$, and so on. The reason for the name "rho heuristic" is that, as Figure 31.7 shows, the sequence $x_1, x_2, \ldots, x_{j-1}$ can be drawn as the "tail" of the rho, and the cycle $x_j, x_{j+1}, \ldots, x_i$ as the "body" of the rho.

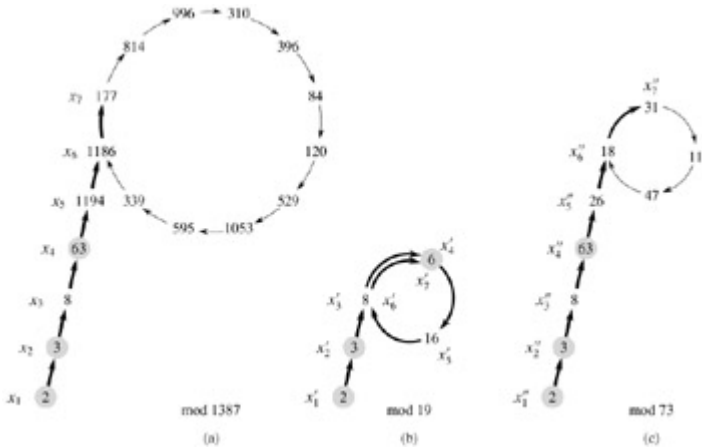

Figure 31.7: Pollard's rho heuristic. *(a)* The values produced by the recurrence $x_{i+1} \leftarrow (x_i^2 - 1) \bmod 1387$, starting with $x_1 = 2$. The prime factorization of 1387 is $19 \cdot 73$. The heavy arrows indicate the iteration steps that are executed before the factor 19 is discovered. The light arrows point to unreached values in the iteration, to illustrate the "rho" shape. The shaded values are the y values stored by POLLARD-RHO. The factor 19 is discovered upon reaching $x_7 = 177$, when $\gcd(63 - 177, 1387) = 19$ is computed. The first x value that would be repeated is 1186, but the factor 19 is discovered before this value is repeated. *(b)* The values produced

by the same recurrence, modulo 19. Every value x_i given in part (a) is equivalent, modulo 19, to the value x'_i shown here. For example, both $x_4 = 63$ and $x_7 = 177$ are equivalent to 6, modulo 19. *(c)* The values produced by the same recurrence, modulo 73. Every value x_i given in part (a) is equivalent, modulo 73, to the value x''_i shown here. By the Chinese remainder theorem, each node in part (a) corresponds to a pair of nodes, one from part (b) and one from part (c).

Let us consider the question of how long it takes for the sequence of x_i to repeat. This is not exactly what we need, but we shall then see how to modify the argument.

For the purpose of this estimation, let us assume that the function

$$f_n(x) = (x^2 - 1) \bmod n$$

behaves like a "random" function. Of course, it is not really random, but this assumption yields results consistent with the observed behavior of POLLARD-RHO. We can then consider each x_i to have been independently drawn from $\mathbf{Z}_n$ according to a uniform distribution on $\mathbf{Z}_n$. By the birthday-paradox analysis of Section 5.4.1, the expected number of steps taken before the sequence cycles is $\Theta(\sqrt{n})$.

Now for the required modification. Let p be a nontrivial factor of n such that $\gcd(p, n/p) = 1$. For example, if n has the factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we may take p to be $p_1^{e_1}$. (If $e_1 = 1$, then p is just the smallest prime factor of n, a good example to keep in mind.)

The sequence x_i induces a corresponding sequence $\langle x'_i \rangle$ modulo p, where

$$x'_i = x_i \bmod p$$

for all i.

Furthermore, because f_n is defined using only arithmetic operations (squaring and subtraction) modulo n, we shall see that one can compute x'_{i+1} from x'_i the "modulo p" view of the sequence is a smaller version of what is happening modulo n:

$$\begin{aligned}
x'_{i+1} &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2 - 1) \bmod n) \bmod p \\
&= (x_i^2 - 1) \bmod p \qquad \text{(by Exercise 31.1-6)} \\
&= ((x_i \bmod p)^2 - 1) \bmod p \\
&= ((x'_i)^2 - 1) \bmod p \\
&= f_p(x'_i) .
\end{aligned}$$

Thus, although we are not explicitly computing the sequence $\langle x'_i \rangle$, this sequence is well defined and obeys the same recurrence as the sequence x_i .

Reasoning as before, we find that the expected number of steps before the sequence $\langle x'_i \rangle$ repeats is $\Theta(\sqrt{p})$. If p is small compared to n, the sequence $\langle x'_i \rangle$ may repeat much more quickly than the sequence x_i . Indeed, the $\langle x'_i \rangle$ sequence repeats as soon as two elements of the sequence x_i

are merely equivalent modulo p, rather than equivalent modulo n. See Figure 31.7, parts (b) and (c), for an illustration.

Let t denote the index of the first repeated value in the (x'_i) sequence, and let $u > 0$ denote the length of the cycle that has been thereby produced. That is, t and $u > 0$ are the smallest values such that $x'_{t+i} = x'_{t+u+i}$ for all $i \geq 0$. By the above arguments, the expected values of t and u are both $\Theta(\sqrt{p})$. Note that if $x'_{t+i} = x'_{t+u+i}$, then $p \mid (x_{t+u+i} - x_{t+i})$. Thus, $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

Therefore, once POLLARD-RHO has saved as y any value x_k such that $k \geq t$, then $y \bmod p$ is always on the cycle modulo p. (If a new value is saved as y, that value is also on the cycle modulo p.) Eventually, k is set to a value that is greater than u, and the procedure then makes an entire loop around the cycle modulo p without changing the value of y. A factor of n is then discovered when x_i "runs into" the previously stored value of y, modulo p, that is, when $x_i \equiv y \pmod{p}$.

Presumably, the factor found is the factor p, although it may occasionally happen that a multiple of p is discovered. Since the expected values of both t and u are $\Theta(\sqrt{p})$, the expected number of steps required to produce the factor p is $\Theta(\sqrt{p})$.

There are two reasons why this algorithm may not perform quite as expected. First, the heuristic analysis of the running time is not rigorous, and it is possible that the cycle of values, modulo p, could be much larger than $\sqrt{p}$. In this case, the algorithm performs correctly but much more slowly than desired. In practice, this issue seems to be moot. Second, the divisors of n produced by this algorithm might always be one of the trivial factors 1 or n. For example, suppose that $n = pq$, where p and q are prime. It can happen that the values of t and u for p are identical with the values of t and u for q, and thus the factor p is always revealed in the same gcd operation that reveals the factor q. Since both factors are revealed at the same time, the trivial factor $pq = n$ is revealed, which is useless. Again, this problem seems to be insignificant in practice. If necessary, the heuristic can be restarted with a different recurrence of the form $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. (The values $c = 0$ and $c = 2$ should be avoided for reasons we won't go into here, but other values are fine.)

Of course, this analysis is heuristic and not rigorous, since the recurrence is not really "random." Nonetheless, the procedure performs well in practice, and it seems to be as efficient as this heuristic analysis indicates. It is the method of choice for finding small prime factors of a large number. To factor a β-bit composite number n completely, we only need to find all prime factors less than $\lfloor n^{1/2} \rfloor$, and so we expect POLLARD-RHO to require at most $n^{1/4} = 2^{\beta/4}$ arithmetic operations and at most $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ bit operations. POLLARD-RHO's ability to find a small factor p of n with an expected number $\Theta(\sqrt{p})$ of arithmetic operations is often its most appealing feature.

Exercises 31.9-1

Referring to the execution history shown in Figure 31.7(a), when does POLLARD-RHO print the factor 73 of 1387?

Exercises 31.9-2

Suppose that we are given a function f: $\mathbf{Z}_n$ $\mathbf{Z}_n$ and an initial value x_0 $\mathbf{Z}_n$. Define $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots$. Let t and $u > 0$ be the smallest values such that $x_{t+i} = x_{t+u+i}$ for $i = 0, 1, \ldots$. In the terminology of Pollard's rho algorithm, t is the length of the tail and u is the length of the cycle of the rho. Give an efficient algorithm to determine t and u exactly, and analyze its running time.

Exercises 31.9-3

How many steps would you expect POLLARD-RHO to require to discover a factor of the form p^e, where p is prime and $e > 1$?

Exercises 31.9-4:

One disadvantage of POLLARD-RHO as written is that it requires one gcd computation for each step of the recurrence. It has been suggested that we might batch the gcd computations by accumulating the product of several x_i values in a row and then using this product instead of x_i in the gcd computation. Describe carefully how you would implement this idea, why it works, and what batch size you would pick as the most effective when working on a β-bit number n.

Problems 31-1: Binary gcd algorithm

On most computers, the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving can be performed more quickly than computing remainders. This problem investigates the ***binary gcd algorithm***, which avoids the remainder computations used in Euclid's algorithm.

a. Prove that if a and b are both even, then gcd(a, b) = 2 gcd$a$/2, b/2).
b. Prove that if a is odd and b is even, then gcd(a, b) = gcd(a, b/2).
c. Prove that if a and b are both odd, then gcd(a, b) = gcd((a - b)/2, b).
d. Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in O(lg a) time. Assume that each subtraction, parity test, and halving can be performed in unit time.

Problems 31-2: Analysis of bit operations in Euclid's algorithm

a. Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b, which yields a quotient q and remainder r. Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.
b. Define $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \bmod b)$ is at most $c(\mu(a, b) - \mu(b, a \bmod b))$ for some sufficiently large constant $c > 0$.
c. Show that EUCLID(a, b) requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β-bit inputs.

Exercises 31-3: Three algorithms for Fibonacci numbers

This problem compares the efficiency of three methods for computing the nth Fibonacci number F_n, given n. Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.

a. Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.21) is exponential in n.
b. Show how to compute F_n in $O(n)$ time using memoization.
c. Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (*Hint:* Consider the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

and its powers.)

d. Assume now that adding two β-bit numbers takes $\Theta(\beta)$ time and that multiplying two β-bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

Problems 31-4: Quadratic residues

Let p be an odd prime. A number $a \in Z_p^*$ is a ***quadratic residue*** if the equation $x^2 = a \pmod p$ has a solution for the unknown x.

a. Show that there are exactly $(p - 1)/2$ quadratic residues, modulo p.
b. If p is prime, we define the ***Legendre symbol*** $\left(\frac{a}{p}\right)$, for $a \in Z_p^*$, to be 1 if a is a quadratic residue modulo p and -1 otherwise. Prove that if $a \in Z_p^*$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p .$$

Give an efficient algorithm for determining whether or not a given number a is a quadratic residue modulo p. Analyze the efficiency of your algorithm.

c. Prove that if p is a prime of the form $4k + 3$ and a is a quadratic residue in $\mathbf{Z}_p^*$, then $a^{k+1} \bmod p$ is a square root of a, modulo p. How much time is required to find the square root of a quadratic residue a modulo p?
d. Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p, that is, a member of $\mathbf{Z}_p^*$ that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

Chapter notes

Niven and Zuckerman [231] provide an excellent introduction to elementary number theory. Knuth [183] contains a good discussion of algorithms for finding the greatest common divisor, as well as other basic number-theoretic algorithms. Bach [28] and Riesel [258] provide more recent surveys of computational number theory. Dixon [78] gives an overview of factorization and primality testing. The conference proceedings edited by Pomerance [245] contains several excellent survey articles. More recently, Bach and Shallit [29] have provided an exceptional overview of the basics of computational number theory.

Knuth [183] discusses the origin of Euclid's algorithm. It appears in Book 7, Propositions 1 and 2, of the Greek mathematician Euclid's *Elements*, which was written around 300 B.C. Euclid's description may have been derived from an algorithm due to Eudoxus around 375 B.C. Euclid's algorithm may hold the honor of being the oldest nontrivial algorithm; it is rivaled only by an algorithm for multiplication which was known to the ancient Egyptians. Shallit [274] chronicles the history of the analysis of Euclid's algorithm.

Knuth attributes a special case of the Chinese remainder theorem (Theorem 31.27) to the Chinese mathematician Sun-Ts u, who lived sometime between 200 B.C. and A.D. 200-the date is quite uncertain. The same special case was given by the Greek mathematician Nichomachus around A.D. 100. It was generalized by Chhin Chiu-Shao in 1247. The Chinese remainder theorem was finally stated and proved in its full generality by L. Euler in 1734.

The randomized primality-testing algorithm presented here is due to Miller [221] and Rabin [254]; it is the fastest randomized primality-testing algorithm known, to within constant factors. The proof of Theorem 31.39 is a slight adaptation of one suggested by Bach [27]. A proof of a stronger result for MILLER-RABIN was given by Monier [224, 225]. Randomization appears to be necessary to obtain a polynomial-time primality-testing algorithm. The fastest deterministic primality-testing algorithm known is the Cohen-Lenstra version [65] of the primality test by Adleman, Pomerance, and Rumely [3]. When testing a number n of length $\lceil \lg(n + 1) \rceil$ for primality, it runs in $(\lg n)^{O(\lg \lg \lg n)}$ time, which is just slightly superpolynomial.

The problem of finding large "random" primes is nicely discussed in an article by Beauchemin, Brassard, Crépeau, Goutier, and Pomerance [33].

The concept of a public-key cryptosystem is due to Diffie and Hellman [74]. The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman [259]. Since then, the field of cryptography has blossomed. Our understanding of the RSA cryptosystem has

deepened, and modern implementations use significant refinements of the basic techniques presented here. In addition, many new techniques have been developed for proving cryptosystems to be secure. For example, Goldwasser and Micali [123] show that randomization can be an effective tool in the design of secure public-key encryption schemes. For signature schemes, Goldwasser, Micali, and Rivest [124] present a digital-signature scheme for which every conceivable type of forgery is provably as difficult as factoring. Menezes et al. [220] provide an overview of applied cryptography.

The rho heuristic for integer factorization was invented by Pollard [242]. The version presented here is a variant proposed by Brent [48].

The best algorithms for factoring large numbers have a running time that grows roughly exponentially with the cube root of the length of the number n to be factored. The general number-field seive factoring algorithm, as developed by Buhler et al. [51] as an extension of the ideas in the number-field sieve factoring algorithm by Pollard [243] and Lenstra et al. [201] and refined by Coppersmith [69] and others, is perhaps the most efficient such algorithm in general for large inputs. Although it is difficult to give a rigorous analysis of this algorithm, under reasonable assumptions we can derive a running-time estimate of $L(1/3, n)^{1.902+o(1)}$, where .

The elliptic-curve method due to Lenstra [202] may be more effective for some inputs than the number field sieve method, since, like Pollard's rho method, it can find a small prime factor p quite quickly. With this method, the time to find p is estimated to be $L(1/2, p)^{\sqrt{2}+o(1)}$.