23. Find a **div** m and a **mod** m when

a) $a = 228, m = 119$.

b) $a = 9009, m = 223$.

c) $a = -10101, m = 333$.

d) $a = -765432, m = 38271$.

24. Find the integer a such that

a) $a \equiv 43 \pmod{23}$ and $-22 \le a \le 0$.

b) $a \equiv 17 \pmod{29}$ and $-14 \le a \le 14$.

c) $a \equiv -11 \pmod{21}$ and $90 \le a \le 110$.

25. Find the integer a such that

a) $a \equiv -15 \pmod{27}$ and $-26 \le a \le 0$.

b) $a \equiv 24 \pmod{31}$ and $-15 \le a \le 15$.

c) $a \equiv 99 \pmod{41}$ and $100 \le a \le 140$.

26. List five integers that are congruent to 4 modulo 12.

27. List all integers between -100 and 100 that are congruent to -1 modulo 25.

28. Decide whether each of these integers is congruent to 3 modulo 7.

a) 37 **b)** 66

c) -17 **d)** -67

29. Decide whether each of these integers is congruent to 5 modulo 17.

a) 80 **b)** 103

c) -29 **d)** -122

30. Find each of these values.

a) $(177 \textbf{ mod } 31 + 270 \textbf{ mod } 31) \textbf{ mod } 31$

b) $(177 \textbf{ mod } 31 \cdot 270 \textbf{ mod } 31) \textbf{ mod } 31$

31. Find each of these values.

a) $(-133 \textbf{ mod } 23 + 261 \textbf{ mod } 23) \textbf{ mod } 23$

b) $(457 \textbf{ mod } 23 \cdot 182 \textbf{ mod } 23) \textbf{ mod } 23$

32. Find each of these values.

a) $(19^2 \textbf{ mod } 41) \textbf{ mod } 9$

b) $(32^3 \textbf{ mod } 13)^2 \textbf{ mod } 11$

c) $(7^3 \textbf{ mod } 23)^2 \textbf{ mod } 31$

d) $(21^2 \textbf{ mod } 15)^3 \textbf{ mod } 22$

33. Find each of these values.

a) $(99^2 \textbf{ mod } 32)^3 \textbf{ mod } 15$

b) $(3^4 \textbf{ mod } 17)^2 \textbf{ mod } 11$

c) $(19^3 \textbf{ mod } 23)^2 \textbf{ mod } 31$

d) $(89^3 \textbf{ mod } 79)^4 \textbf{ mod } 26$

34. Show that if $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, where a, b, c, d, and m are integers with $m \ge 2$, then $a - c \equiv b - d \pmod{m}$.

35. Show that if $n \mid m$, where n and m are integers greater than 1, and if $a \equiv b \pmod{m}$, where a and b are integers, then $a \equiv b \pmod{n}$.

36. Show that if a, b, c, and m are integers such that $m \ge 2$, $c > 0$, and $a \equiv b \pmod{m}$, then $ac \equiv bc \pmod{mc}$.

37. Find counterexamples to each of these statements about congruences.

a) If $ac \equiv bc \pmod{m}$, where a, b, c, and m are integers with $m \ge 2$, then $a \equiv b \pmod{m}$.

b) If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, where a, b, c, d, and m are integers with c and d positive and $m \ge 2$, then $a^c \equiv b^d \pmod{m}$.

38. Show that if n is an integer then $n^2 \equiv 0$ or $1 \pmod{4}$.

39. Use Exercise 38 to show that if m is a positive integer of the form $4k + 3$ for some nonnegative integer k, then m is not the sum of the squares of two integers.

40. Prove that if n is an odd positive integer, then $n^2 \equiv 1 \pmod{8}$.

41. Show that if a, b, k, and m are integers such that $k \ge 1$, $m \ge 2$, and $a \equiv b \pmod{m}$, then $a^k \equiv b^k \pmod{m}$.

42. Show that $\mathbf{Z}_m$ with addition modulo m, where $m \ge 2$ is an integer, satisfies the closure, associative, and commutative properties, 0 is an additive identity, and for every nonzero $a \in \mathbf{Z}_m$, $m - a$ is an inverse of a modulo m.

43. Show that $\mathbf{Z}_m$ with multiplication modulo m, where $m \ge 2$ is an integer, satisfies the closure, associative, and commutativity properties, and 1 is a multiplicative identity.

44. Show that the distributive property of multiplication over addition holds for $\mathbf{Z}_m$, where $m \ge 2$ is an integer.

45. Write out the addition and multiplication tables for $\mathbf{Z}_5$ (where by addition and multiplication we mean $+_5$ and $\cdot_5$).

46. Write out the addition and multiplication tables for $\mathbf{Z}_6$ (where by addition and multiplication we mean $+_6$ and $\cdot_6$).

47. Determine whether each of the functions $f(a) = a$ **div** d and $g(a) = a$ **mod** d, where d is a fixed positive integer, from the set of integers to the set of integers, is one-to-one, and determine whether each of these functions is onto.

4.2 Integer Representations and Algorithms

Introduction

Integers can be expressed using any integer greater than one as a base, as we will show in this section. Although we commonly use decimal (base 10), representations, binary (base 2), octal (base 8), and hexadecimal (base 16) representations are often used, especially in computer science. Given a base b and an integer n, we will show how to construct the base b representation of this integer. We will also explain how to quickly covert between binary and octal and between binary and hexadecimal notations.

As mentioned in Section 3.1, the term *algorithm* originally referred to procedures for performing arithmetic operations using the decimal representations of integers. These algorithms, adapted for use with binary representations, are the basis for computer arithmetic. They provide good illustrations of the concept of an algorithm and the complexity of algorithms. For these reasons, they will be discussed in this section.

We will also introduce an algorithm for finding a **div** d and a **mod** d where a and d are integers with $d > 1$. Finally, we will describe an efficient algorithm for modular exponentiation, which is a particularly important algorithm for cryptography, as we will see in Section 4.6.

Representations of Integers

In everyday life we use decimal notation to express integers. For example, 965 is used to denote $9 \cdot 10^2 + 6 \cdot 10 + 5$. However, it is often convenient to use bases other than 10. In particular, computers usually use binary notation (with 2 as the base) when carrying out arithmetic, and octal (base 8) or hexadecimal (base 16) notation when expressing characters, such as letters or digits. In fact, we can use any integer greater than 1 as the base when expressing integers. This is stated in Theorem 1.

THEOREM 1 Let b be an integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0,$$

where k is a nonnegative integer, $a_0, a_1, \ldots, a_k$ are nonnegative integers less than b, and $a_k \neq 0$.

A proof of this theorem can be constructed using mathematical induction, a proof method that is discussed in Section 5.1. It can also be found in [Ro10]. The representation of n given in Theorem 1 is called the **base *b* expansion of *n***. The base b expansion of n is denoted by $(a_k a_{k-1} \ldots a_1 a_0)_b$. For instance, $(245)_8$ represents $2 \cdot 8^2 + 4 \cdot 8 + 5 = 165$. Typically, the subscript 10 is omitted for base 10 expansions of integers because base 10, or **decimal expansions**, are commonly used to represent integers.

BINARY EXPANSIONS Choosing 2 as the base gives **binary expansions** of integers. In binary notation each digit is either a 0 or a 1. In other words, the binary expansion of an integer is just a bit string. Binary expansions (and related expansions that are variants of binary expansions) are used by computers to represent and do arithmetic with integers.

EXAMPLE 1 What is the decimal expansion of the integer that has $(1\ 0101\ 1111)_2$ as its binary expansion?

Solution: We have

$$\begin{aligned}(1\ 0101\ 1111)_2 &= 1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 \\ &\quad + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 351.\end{aligned}$$ ◀

OCTAL AND HEXADECIMAL EXPANSIONS Among the most important bases in computer science are base 2, base 8, and base 16. Base 8 expansions are called **octal** expansions and base 16 expansions are **hexadecimal** expansions.

EXAMPLE 2 What is the decimal expansion of the number with octal expansion $(7016)_8$?

Solution: Using the definition of a base b expansion with $b = 8$ tells us that

$$(7016)_8 = 7 \cdot 8^3 + 0 \cdot 8^2 + 1 \cdot 8 + 6 = 3598. \quad ◀$$

Sixteen different digits are required for hexadecimal expansions. Usually, the hexadecimal digits used are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F, where the letters A through F represent the digits corresponding to the numbers 10 through 15 (in decimal notation).

EXAMPLE 3 What is the decimal expansion of the number with hexadecimal expansion $(2AE0B)_{16}$?

Solution: Using the definition of a base b expansion with $b = 16$ tells us that

$$(2AE0B)_{16} = 2 \cdot 16^4 + 10 \cdot 16^3 + 14 \cdot 16^2 + 0 \cdot 16 + 11 = 175627. \quad ◀$$

Each hexadecimal digit can be represented using four bits. For instance, we see that $(1110\ \ 0101)_2 = (E5)_{16}$ because $(1110)_2 = (E)_{16}$ and $(0101)_2 = (5)_{16}$. **Bytes**, which are bit strings of length eight, can be represented by two hexadecimal digits.

BASE CONVERSION We will now describe an algorithm for constructing the base b expansion of an integer n. First, divide n by b to obtain a quotient and remainder, that is,

$$n = bq_0 + a_0, \qquad 0 \le a_0 < b.$$

The remainder, a_0, is the rightmost digit in the base b expansion of n. Next, divide q_0 by b to obtain

$$q_0 = bq_1 + a_1, \qquad 0 \le a_1 < b.$$

We see that a_1 is the second digit from the right in the base b expansion of n. Continue this process, successively dividing the quotients by b, obtaining additional base b digits as the remainders. This process terminates when we obtain a quotient equal to zero. It produces the base b digits of n from the right to the left.

EXAMPLE 4 Find the octal expansion of $(12345)_{10}$.

Extra Examples

Solution: First, divide 12345 by 8 to obtain

$$12345 = 8 \cdot 1543 + 1.$$

Successively dividing quotients by 8 gives

$$\begin{aligned} 1543 &= 8 \cdot 192 + 7, \\ 192 &= 8 \cdot 24 + 0, \\ 24 &= 8 \cdot 3 + 0, \\ 3 &= 8 \cdot 0 + 3. \end{aligned}$$

The successive remainders that we have found, 1, 7, 0, 0, and 3, are the digits from the right to the left of 12345 in base 8. Hence,

$$(12345)_{10} = (30071)_8. \quad ◀$$

EXAMPLE 5 Find the hexadecimal expansion of $(177130)_{10}$.

Solution: First divide 177130 by 16 to obtain

$$177130 = 16 \cdot 11070 + 10.$$

Successively dividing quotients by 16 gives

$$\begin{aligned} 11070 &= 16 \cdot 691 + 14, \\ 691 &= 16 \cdot 43 + 3, \\ 43 &= 16 \cdot 2 + 11, \\ 2 &= 16 \cdot 0 + 2. \end{aligned}$$

The successive remainders that we have found, 10, 14, 3, 11, 2, give us the digits from the right to the left of 177130 in the hexadecimal (base 16) expansion of $(177130)_{10}$. It follows that

$$(177130)_{10} = (2B3EA)_{16}.$$

(Recall that the integers 10, 11, and 14 correspond to the hexadecimal digits A, B, and E, respectively.) ◀

EXAMPLE 6 Find the binary expansion of $(241)_{10}$.

Solution: First divide 241 by 2 to obtain

$$241 = 2 \cdot 120 + 1.$$

Successively dividing quotients by 2 gives

$$\begin{aligned} 120 &= 2 \cdot 60 + 0, \\ 60 &= 2 \cdot 30 + 0, \\ 30 &= 2 \cdot 15 + 0, \\ 15 &= 2 \cdot 7 + 1, \\ 7 &= 2 \cdot 3 + 1, \\ 3 &= 2 \cdot 1 + 1, \\ 1 &= 2 \cdot 0 + 1. \end{aligned}$$

The successive remainders that we have found, 1, 0, 0, 0, 1, 1, 1, 1, are the digits from the right to the left in the binary (base 2) expansion of $(241)_{10}$. Hence,

$$(241)_{10} = (1111\ 0001)_2.$$ ◀

The pseudocode given in Algorithm 1 finds the base b expansion $(a_{k-1} \ldots a_1 a_0)_b$ of the integer n.

TABLE 1 **Hexadecimal, Octal, and Binary Representation of the Integers 0 through 15.**

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Octal	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

ALGORITHM 1 **Constructing Base b Expansions.**

```
procedure base b expansion($n, b$: positive integers with $b > 1$)
$q := n$
$k := 0$
while $q \neq 0$
      $a_k := q$ mod $b$
      $q := q$ div $b$
      $k := k + 1$
return $(a_{k-1}, \dots, a_1, a_0)$ {$(a_{k-1} \dots a_1 a_0)_b$ is the base $b$ expansion of $n$}
```

In Algorithm 1, q represents the quotient obtained by successive divisions by b, starting with $q = n$. The digits in the base b expansion are the remainders of these divisions and are given by q **mod** b. The algorithm terminates when a quotient $q = 0$ is reached.

Remark: Note that Algorithm 1 can be thought of as a greedy algorithm, as the base b digits are taken as large as possible in each step.

CONVERSION BETWEEN BINARY, OCTAL, AND HEXADECIMAL EXPANSIONS

Conversion between binary and octal and between binary and hexadecimal expansions is extremely easy because each octal digit corresponds to a block of three binary digits and each hexadecimal digit corresponds to a block of four binary digits, with these correspondences shown in Table 1 without initial 0s shown. (We leave it as Exercises 13–16 to show that this is the case.) This conversion is illustrated in Example 7.

EXAMPLE 7 Find the octal and hexadecimal expansions of $(11\ 1110\ 1011\ 1100)_2$ and the binary expansions of $(765)_8$ and $(\mathrm{A8D})_{16}$.

Solution: To convert $(11\ 1110\ 1011\ 1100)_2$ into octal notation we group the binary digits into blocks of three, adding initial zeros at the start of the leftmost block if necessary. These blocks, from left to right, are 011, 111, 010, 111, and 100, corresponding to 3, 7, 2, 7, and 4, respectively. Consequently, $(11\ 1110\ 1011\ 1100)_2 = (37274)_8$. To convert $(11\ 1110\ 1011\ 1100)_2$ into hexadecimal notation we group the binary digits into blocks of four, adding initial zeros at the start of the leftmost block if necessary. These blocks, from left to right, are 0011, 1110, 1011, and 1100, corresponding to the hexadecimal digits 3, E, B, and C, respectively. Consequently, $(11\ 1110\ 1011\ 1100)_2 = (\mathrm{3EBC})_{16}$.

To convert $(765)_8$ into binary notation, we replace each octal digit by a block of three binary digits. These blocks are 111, 110, and 101. Hence, $(765)_8 = (1\ 1111\ 0101)_2$. To convert $(\mathrm{A8D})_{16}$ into binary notation, we replace each hexadecimal digit by a block of four binary digits. These blocks are 1010, 1000, and 1101. Hence, $(\mathrm{A8D})_{16} = (1010\ 1000\ 1101)_2$. ◀

Algorithms for Integer Operations

The algorithms for performing operations with integers using their binary expansions are extremely important in computer arithmetic. We will describe algorithms for the addition and the multiplication of two integers expressed in binary notation. We will also analyze the computational complexity of these algorithms, in terms of the actual number of bit operations used. Throughout this discussion, suppose that the binary expansions of a and b are

$$a = (a_{n-1}a_{n-2} \ldots a_1a_0)_2, \; b = (b_{n-1}b_{n-2} \ldots b_1b_0)_2,$$

so that a and b each have n bits (putting bits equal to 0 at the beginning of one of these expansions if necessary).

We will measure the complexity of algorithms for integer arithmetic in terms of the number of bits in these numbers.

ADDITION ALGORITHM Consider the problem of adding two integers in binary notation. A procedure to perform addition can be based on the usual method for adding numbers with pencil and paper. This method proceeds by adding pairs of binary digits together with carries, when they occur, to compute the sum of two integers. This procedure will now be specified in detail.

To add a and b, first add their rightmost bits. This gives

$$a_0 + b_0 = c_0 \cdot 2 + s_0,$$

where s_0 is the rightmost bit in the binary expansion of $a + b$ and c_0 is the **carry**, which is either 0 or 1. Then add the next pair of bits and the carry,

$$a_1 + b_1 + c_0 = c_1 \cdot 2 + s_1,$$

where s_1 is the next bit (from the right) in the binary expansion of $a + b$, and c_1 is the carry. Continue this process, adding the corresponding bits in the two binary expansions and the carry, to determine the next bit from the right in the binary expansion of $a + b$. At the last stage, add a_{n-1}, b_{n-1}, and c_{n-2} to obtain $c_{n-1} \cdot 2 + s_{n-1}$. The leading bit of the sum is $s_n = c_{n-1}$. This procedure produces the binary expansion of the sum, namely, $a + b = (s_ns_{n-1}s_{n-2} \ldots s_1s_0)_2$.

EXAMPLE 8 Add $a = (1110)_2$ and $b = (1011)_2$.

Solution: Following the procedure specified in the algorithm, first note that

$$a_0 + b_0 = 0 + 1 = 0 \cdot 2 + 1,$$

so that $c_0 = 0$ and $s_0 = 1$. Then, because

$$a_1 + b_1 + c_0 = 1 + 1 + 0 = 1 \cdot 2 + 0,$$

it follows that $c_1 = 1$ and $s_1 = 0$. Continuing,

$$a_2 + b_2 + c_1 = 1 + 0 + 1 = 1 \cdot 2 + 0,$$

so that $c_2 = 1$ and $s_2 = 0$. Finally, because

$$a_3 + b_3 + c_2 = 1 + 1 + 1 = 1 \cdot 2 + 1,$$

follows that $c_3 = 1$ and $s_3 = 1$. This means that $s_4 = c_3 = 1$. Therefore, $s = a + b = (1\,1001)_2$. This addition is displayed in Figure 1, where carries are shown in blue.

$$\begin{array}{r} 1\ 1\ 1\ \ \ \\ 1\ 1\ 1\ 0 \\ +\ 1\ 0\ 1\ 1 \\ \hline 1\ 1\ 0\ 0\ 1 \end{array}$$

FIGURE 1
Adding $(1110)_2$ and $(1011)_2$.

The algorithm for addition can be described using pseudocode as follows.

ALGORITHM 2 Addition of Integers.

```
procedure add(a, b: positive integers)
{the binary expansions of $a$ and $b$ are $(a_{n-1}a_{n-2}\ldots a_1a_0)_2$
  and $(b_{n-1}b_{n-2}\ldots b_1b_0)_2$, respectively}
$c := 0$
for $j := 0$ to $n - 1$
    $d := \lfloor (a_j + b_j + c)/2 \rfloor$
    $s_j := a_j + b_j + c - 2d$
    $c := d$
$s_n := c$
return $(s_0, s_1, \ldots, s_n)$ {the binary expansion of the sum is $(s_n s_{n-1}\ldots s_0)_2$}
```

Next, the number of additions of bits used by Algorithm 2 will be analyzed.

EXAMPLE 9 How many additions of bits are required to use Algorithm 2 to add two integers with n bits (or less) in their binary representations?

Solution: Two integers are added by successively adding pairs of bits and, when it occurs, a carry. Adding each pair of bits and the carry requires two additions of bits. Thus, the total number of additions of bits used is less than twice the number of bits in the expansion. Hence, the number of additions of bits used by Algorithm 2 to add two n-bit integers is $O(n)$.

MULTIPLICATION ALGORITHM Next, consider the multiplication of two n-bit integers a and b. The conventional algorithm (used when multiplying with pencil and paper) works as follows. Using the distributive law, we see that

$$\begin{aligned} ab &= a(b_0 2^0 + b_1 2^1 + \cdots + b_{n-1}2^{n-1}) \\ &= a(b_0 2^0) + a(b_1 2^1) + \cdots + a(b_{n-1}2^{n-1}). \end{aligned}$$

We can compute ab using this equation. We first note that $ab_j = a$ if $b_j = 1$ and $ab_j = 0$ if $b_j = 0$. Each time we multiply a term by 2, we shift its binary expansion one place to the left and add a zero at the tail end of the expansion. Consequently, we can obtain $(ab_j)2^j$ by **shifting** the binary expansion of ab_j j places to the left, adding j zero bits at the tail end of this binary expansion. Finally, we obtain ab by adding the n integers $ab_j 2^j$, $j = 0, 1, 2, \ldots, n - 1$.

Algorithm 3 displays this procedure for multiplication.

ALGORITHM 3 Multiplication of Integers.

```
procedure multiply($a$, $b$: positive integers)
{the binary expansions of $a$ and $b$ are $(a_{n-1}a_{n-2}\dots a_1a_0)_2$
  and $(b_{n-1}b_{n-2}\dots b_1b_0)_2$, respectively}
for $j := 0$ to $n-1$
    if $b_j = 1$ then $c_j := a$ shifted $j$ places
    else $c_j := 0$
{$c_0, c_1, \dots, c_{n-1}$ are the partial products}
$p := 0$
for $j := 0$ to $n-1$
    $p := p + c_j$
return $p$ {$p$ is the value of $ab$}
```

Example 10 illustrates the use of this algorithm.

EXAMPLE 10 Find the product of $a = (110)_2$ and $b = (101)_2$.

Solution: First note that

$$ab_0 \cdot 2^0 = (110)_2 \cdot 1 \cdot 2^0 = (110)_2,$$

$$ab_1 \cdot 2^1 = (110)_2 \cdot 0 \cdot 2^1 = (0000)_2,$$

and

$$ab_2 \cdot 2^2 = (110)_2 \cdot 1 \cdot 2^2 = (11000)_2.$$

To find the product, add $(110)_2$, $(0000)_2$, and $(11000)_2$. Carrying out these additions (using Algorithm 2, including initial zero bits when necessary) shows that $ab = (1\ 1110)_2$. This multiplication is displayed in Figure 2. ◀

$$\begin{array}{r}
1\,1\,0 \\
\times\ 1\,0\,1 \\
\hline
1\,1\,0 \\
0\,0\,0 \\
1\,1\,0 \\
\hline
1\,1\,1\,1\,0
\end{array}$$

FIGURE 2 Multiplying $(110)_2$ and $(101)_2$.

Next, we determine the number of additions of bits and shifts of bits used by Algorithm 3 to multiply two integers.

EXAMPLE 11 How many additions of bits and shifts of bits are used to multiply a and b using Algorithm 3?

Solution: Algorithm 3 computes the products of a and b by adding the partial products $c_0, c_1, c_2, \dots,$ and c_{n-1}. When $b_j = 1$, we compute the partial product c_j by shifting the binary expansion of a by j bits. When $b_j = 0$, no shifts are required because $c_j = 0$. Hence, to find all n of the integers $ab_j 2^j$, $j = 0, 1, \dots, n-1$, requires at most

$$0 + 1 + 2 + \cdots + n - 1$$

shifts. Hence, by Example 5 in Section 3.2 the number of shifts required is $O(n^2)$.

To add the integers ab_j from $j = 0$ to $j = n - 1$ requires the addition of an n-bit integer, an $(n + 1)$-bit integer, ..., and a $(2n)$-bit integer. We know from Example 9 that each of these additions requires $O(n)$ additions of bits. Consequently, a total of $O(n^2)$ additions of bits are required for all n additions. ◀

Surprisingly, there are more efficient algorithms than the conventional algorithm for multiplying integers. One such algorithm, which uses $O(n^{1.585})$ bit operations to multiply n-bit numbers, will be described in Section 8.3.

ALGORITHM FOR div AND mod Given integers a and d, $d > 0$, we can find $q = a$ **div** d and $r = a$ **mod** d using Algorithm 4. In this brute-force algorithm, when a is positive we subtract d from a as many times as necessary until what is left is less than d. The number of times we perform this subtraction is the quotient and what is left over after all these subtractions is the remainder. Algorithm 4 also covers the case where a is negative. This algorithm finds the quotient q and remainder r when $|a|$ is divided by d. Then, when $a < 0$ and $r > 0$, it uses these to find the quotient $-(q + 1)$ and remainder $d - r$ when a is divided by d. We leave it to the reader (Exercise 59) to show that, assuming that $a > d$, this algorithm uses $O(q \log a)$ bit operations.

ALGORITHM 4 Computing div and mod.

```
procedure division algorithm($a$: integer, $d$: positive integer)
$q := 0$
$r := |a|$
while $r \geq d$
    $r := r - d$
    $q := q + 1$
if $a < 0$ and $r > 0$ then
    $r := d - r$
    $q := -(q + 1)$
return $(q, r)$ {$q = a$ div $d$ is the quotient, $r = a$ mod $d$ is the remainder}
```

There are more efficient algorithms than Algorithm 4 for determining the quotient $q = a$ **div** d and the remainder $r = a$ **mod** d when a positive integer a is divided by a positive integer d (see [Kn98] for details). These algorithms require $O(\log a \cdot \log d)$ bit operations. If both of the binary expansions of a and d contain n or fewer bits, then we can replace $\log a \cdot \log d$ by n^2. This means that we need $O(n^2)$ bit operations to find the quotient and remainder when a is divided by d.

Modular Exponentiation

In cryptography it is important to be able to find b^n **mod** m efficiently, where b, n, and m are large integers. It is impractical to first compute b^n and then find its remainder when divided by m because b^n will be a huge number. Instead, we can use an algorithm that employs the binary expansion of the exponent n.

Before we present this algorithm, we illustrate its basic idea. We will explain how to use the binary expansion of n, say $n = (a_{k-1} \ldots a_1 a_0)_2$, to compute b^n. First, note that

$$b^n = b^{a_{k-1}\cdot 2^{k-1}+\cdots+a_1\cdot 2+a_0} = b^{a_{k-1}\cdot 2^{k-1}} \cdots b^{a_1\cdot 2} \cdot b^{a_0}.$$

This shows that to compute b^n, we need only compute the values of b, b^2, $(b^2)^2 = b^4$, $(b^4)^2 = b^8, \ldots, b^{2^k}$. Once we have these values, we multiply the terms b^{2^j} in this list, where $a_j = 1$. (For efficiency, after multiplying by each term, we reduce the result modulo m.) This gives us b^n. For example, to compute 3^{11} we first note that $11 = (1011)_2$, so that $3^{11} = 3^8 3^2 3^1$. By successively squaring, we find that $3^2 = 9$, $3^4 = 9^2 = 81$, and $3^8 = (81)^2 = 6561$. Consequently, $3^{11} = 3^8 3^2 3^1 = 6561 \cdot 9 \cdot 3 = 177{,}147$.

The algorithm successively finds b **mod** m, b^2 **mod** m, b^4 **mod** $m, \ldots, b^{2^{k-1}}$ **mod** m and multiplies together those terms b^{2^j} **mod** m where $a_j = 1$, finding the remainder of the product when divided by m after each multiplication. Pseudocode for this algorithm is shown in Algorithm 5. Note that in Algorithm 5 we can use the most efficient algorithm available to compute values of the **mod** function, not necessarily Algorithm 4.

Be sure to reduce modulo m after each multiplication!

ALGORITHM 5 Modular Exponentiation.

```
procedure modular exponentiation(b: integer, $n = (a_{k-1}a_{k-2}\ldots a_1a_0)_2$,
        m: positive integers)
x := 1
power := b mod m
for i := 0 to k − 1
    if $a_i = 1$ then x := (x · power) mod m
    power := (power · power) mod m
return x {x equals $b^n$ mod m}
```

We illustrate how Algorithm 5 works in Example 12.

EXAMPLE 12 Use Algorithm 5 to find 3^{644} **mod** 645.

Solution: Algorithm 5 initially sets $x = 1$ and *power* $= 3$ **mod** $645 = 3$. In the computation of 3^{644} **mod** 645, this algorithm determines 3^{2^j} **mod** 645 for $j = 1, 2, \ldots, 9$ by successively squaring and reducing modulo 645. If $a_j = 1$ (where a_j is the bit in the jth position in the binary expansion of 644, which is $(1010000100)_2$), it multiplies the current value of x by 3^{2^j} **mod** 645 and reduces the result modulo 645. Here are the steps used:

$i = 0$: Because $a_0 = 0$, we have $x = 1$ and *power* $= 3^2$ **mod** $645 = 9$ **mod** $645 = 9$;
$i = 1$: Because $a_1 = 0$, we have $x = 1$ and *power* $= 9^2$ **mod** $645 = 81$ **mod** $645 = 81$;
$i = 2$: Because $a_2 = 1$, we have $x = 1 \cdot 81$ **mod** $645 = 81$ and *power* $= 81^2$ **mod** $645 = 6561$ **mod** $645 = 111$;
$i = 3$: Because $a_3 = 0$, we have $x = 81$ and *power* $= 111^2$ **mod** $645 = 12{,}321$ **mod** $645 = 66$;
$i = 4$: Because $a_4 = 0$, we have $x = 81$ and *power* $= 66^2$ **mod** $645 = 4356$ **mod** $645 = 486$;
$i = 5$: Because $a_5 = 0$, we have $x = 81$ and *power* $= 486^2$ **mod** $645 = 236{,}196$ **mod** $645 = 126$;
$i = 6$: Because $a_6 = 0$, we have $x = 81$ and *power* $= 126^2$ **mod** $645 = 15{,}876$ **mod** $645 = 396$;
$i = 7$: Because $a_7 = 1$, we find that $x = (81 \cdot 396)$ **mod** $645 = 471$ and *power* $= 396^2$ **mod** $645 = 156{,}816$ **mod** $645 = 81$;
$i = 8$: Because $a_8 = 0$, we have $x = 471$ and *power* $= 81^2$ **mod** $645 = 6561$ **mod** $645 = 111$;
$i = 9$: Because $a_9 = 1$, we find that $x = (471 \cdot 111)$ **mod** $645 = 36$.

This shows that following the steps of Algorithm 5 produces the result 3^{644} **mod** $645 = 36$. ◀

Algorithm 5 is quite efficient; it uses $O((\log m)^2 \log n)$ bit operations to find b^n **mod** m (see Exercise 58).