Chapter 14

Efficient Implementation

Contents in Brief

14.1 Introduction

Many public-key encryption and digital signature schemes, and some hash functions (see §9.4.3), require computations in $\mathbb{Z}_m$, the integers modulo m (m is a large positive integer which may or may not be a prime). For example, the RSA, Rabin, and ElGamal schemes require efficient methods for performing multiplication and exponentiation in $\mathbb{Z}_m$. Although $\mathbb{Z}_m$ is prominent in many aspects of modern applied cryptography, other algebraic structures are also important. These include, but are not limited to, polynomial rings, finite fields, and finite cyclic groups. For example, the group formed by the points on an elliptic curve over a finite field has considerable appeal for various cryptographic applications. The efficiency of a particular cryptographic scheme based on any one of these algebraic structures will depend on a number of factors, such as parameter size, time-memory tradeoffs, processing power available, software and/or hardware optimization, and mathematical algorithms.

This chapter is concerned primarily with mathematical algorithms for efficiently carrying out computations in the underlying algebraic structure. Since many of the most widely implemented techniques rely on $\mathbb{Z}_m$, emphasis is placed on efficient algorithms for performing the basic arithmetic operations in this structure (addition, subtraction, multiplication, division, and exponentiation).

In some cases, several algorithms will be presented which perform the same operation. For example, a number of techniques for doing modular multiplication and exponentiation are discussed in §14.3 and §14.6, respectively. Efficiency can be measured in numerous ways; thus, it is difficult to definitively state which algorithm is the best. An algorithm may be efficient in the time it takes to perform a certain algebraic operation, but quite inefficient in the amount of storage it requires. One algorithm may require more code space than another. Depending on the environment in which computations are to be performed, one algorithm may be preferable over another. For example, current chipcard technology provides very limited storage for both precomputed values and program code. For such applications, an algorithm which is less efficient in time but very efficient in memory requirements may be preferred.

The algorithms described in this chapter are those which, for the most part, have received considerable attention in the literature. Although some attempt is made to point out their relative merits, no detailed comparisons are given.

Chapter outline

§14.2 deals with the basic arithmetic operations of addition, subtraction, multiplication, squaring, and division for multiple-precision integers. §14.3 describes the basic arithmetic operations of addition, subtraction, and multiplication in $\mathbb{Z}_m$. Techniques described for performing modular reduction for an arbitrary modulus m are the classical method (§14.3.1), Montgomery's method (§14.3.2), and Barrett's method (§14.3.3). §14.3.4 describes a reduction procedure ideally suited to moduli of a special form. Greatest common divisor (gcd) algorithms are the topic of §14.4, including the binary gcd algorithm (§14.4.1) and Lehmer's gcd algorithm (§14.4.2). Efficient algorithms for performing extended gcd computations are given in §14.4.3. Modular inverses are also considered in §14.4.3. Garner's algorithm for implementing the Chinese remainder theorem can be found in §14.5. §14.6 is a treatment of several of the most practical exponentiation algorithms. §14.6.1 deals with exponentiation in general, without consideration of any special conditions. §14.6.2 looks at exponentiation when the base is variable and the exponent is fixed. §14.6.3 considers algorithms which take advantage of a fixed-base element and variable exponent. Techniques involving representing the exponent in non-binary form are given in §14.7; recoding the exponent may allow significant performance enhancements. §14.8 contains further notes and references.

14.2 Multiple-precision integer arithmetic

This section deals with the basic operations performed on multiple-precision integers: addition, subtraction, multiplication, squaring, and division. The algorithms presented in this section are commonly referred to as the *classical methods*.

14.2.1 Radix representation

Positive integers can be represented in various ways, the most common being *base* 10. For example, $a = 123$ base 10 means $a = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$. For machine computations, *base* 2 (*binary representation*) is preferable. If $a = 1111011$ base 2, then $a = 2^6 + 2^5 + 2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0$.

14.1 Fact If $b \geq 2$ is an integer, then any positive integer a can be expressed uniquely as $a = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$, where a_i is an integer with $0 \leq a_i < b$ for $0 \leq i \leq n$, and $a_n \neq 0$.

14.2 Definition The representation of a positive integer a as a sum of multiples of powers of b, as given in Fact 14.1, is called the *base* b or *radix* b representation of a.

14.3 Note (*notation and terminology*)

(i) The base b representation of a positive integer a given in Fact 14.1 is usually written as $a = (a_n a_{n-1} \cdots a_1 a_0)_b$. The integers a_i, $0 \le i \le n$, are called *digits*. a_n is called the *most significant digit* or *high-order digit*; a_0 the *least significant digit* or *low-order digit*. If $b = 10$, the standard notation is $a = a_n a_{n-1} \cdots a_1 a_0$.

(ii) It is sometimes convenient to pad high-order digits of a base b representation with 0's; such a padded number will also be referred to as the base b representation.

(iii) If $(a_n a_{n-1} \cdots a_1 a_0)_b$ is the base b representation of a and $a_n \ne 0$, then the *precision* or *length* of a is $n+1$. If $n = 0$, then a is called a *single-precision integer*; otherwise, a is a *multiple-precision integer*. $a = 0$ is also a single-precision integer.

The division algorithm for integers (see Definition 2.82) provides an efficient method for determining the base b representation of a non-negative integer, for a given base b. This provides the basis for Algorithm 14.4.

14.4 Algorithm Radix b representation

INPUT: integers a and b, $a \ge 0$, $b \ge 2$.
OUTPUT: the base b representation $a = (a_n \cdots a_1 a_0)_b$, where $n \ge 0$ and $a_n \ne 0$ if $n \ge 1$.

1. $i \leftarrow 0$, $x \leftarrow a$, $q \leftarrow \lfloor \frac{x}{b} \rfloor$, $a_i \leftarrow x - qb$. ($\lfloor \cdot \rfloor$ is the floor function; see page 49.)
2. While $q > 0$, do the following:
   2.1 $i \leftarrow i+1$, $x \leftarrow q$, $q \leftarrow \lfloor \frac{x}{b} \rfloor$, $a_i \leftarrow x - qb$.
3. Return($(a_i a_{i-1} \cdots a_1 a_0)$).

14.5 Fact If $(a_n a_{n-1} \cdots a_1 a_0)_b$ is the base b representation of a and k is a positive integer, then $(u_l u_{l-1} \cdots u_1 u_0)_{b^k}$ is the base b^k representation of a, where $l = \lceil (n+1)/k \rceil - 1$, $u_i = \sum_{j=0}^{k-1} a_{ik+j} b^j$ for $0 \le i \le l-1$, and $u_l = \sum_{j=0}^{n-lk} a_{lk+j} b^j$.

14.6 Example (*radix b representation*) The base 2 representation of $a = 123$ is $(1111011)_2$. The base 4 representation of a is easily obtained from its base 2 representation by grouping digits in pairs from the right: $a = ((1)_2 (11)_2 (10)_2 (11)_2)_4 = (1323)_4$. □

Representing negative numbers

Negative integers can be represented in several ways. Two commonly used methods are:

1. *signed-magnitude representation*
2. *complement representation*.

These methods are described below. The algorithms provided in this chapter all assume a signed-magnitude representation for integers, with the sign digit being implicit.

(i) Signed-magnitude representation

The *sign* of an integer (i.e., either positive or negative) and its *magnitude* (i.e., absolute value) are represented separately in a *signed-magnitude representation*. Typically, a positive integer is assigned a sign digit 0, while a negative integer is assigned a sign digit $b-1$. For n-digit radix b representations, only $2b^{n-1}$ sequences out of the b^n possible sequences are utilized: precisely $b^{n-1}-1$ positive integers and $b^{n-1}-1$ negative integers can be represented, and 0 has two representations. Table 14.1 illustrates the binary signed-magnitude representation of the integers in the range $[7, -7]$.

Signed-magnitude representation has the drawback that when certain operations (such as addition and subtraction) are performed, the sign digit must be checked to determine the appropriate manner to perform the computation. Conditional branching of this type can be costly when many operations are performed.

(ii) Complement representation

Addition and subtraction using *complement representation* do not require the checking of the sign digit. Non-negative integers in the range $[0, b^{n-1} - 1]$ are represented by base b sequences of length n with the high-order digit being 0. Suppose x is a positive integer in this range represented by the sequence $(x_n x_{n-1} \cdots x_1 x_0)_b$ where $x_n = 0$. Then $-x$ is represented by the sequence $\overline{x} = (\overline{x}_n \overline{x}_{n-1} \cdots \overline{x}_1 \overline{x}_0) + 1$ where $\overline{x}_i = b - 1 - x_i$ and $+$ is the standard addition with carry. Table 14.1 illustrates the binary complement representation of the integers in the range $[-7, 7]$. In the binary case, complement representation is referred to as *two's complement representation*.

Sequence	Signed-magnitude	Two's complement
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

Sequence	Signed-magnitude	Two's complement
1111	-7	-1
1110	-6	-2
1101	-5	-3
1100	-4	-4
1011	-3	-5
1010	-2	-6
1001	-1	-7
1000	-0	-8

Table 14.1: *Signed-magnitude and two's complement representations of integers in* $[-7, 7]$.

14.2.2 Addition and subtraction

Addition and subtraction are performed on two integers having the same number of base b digits. To add or subtract two integers of different lengths, the smaller of the two integers is first padded with 0's on the left (i.e., in the high-order positions).

14.7 Algorithm Multiple-precision addition

INPUT: positive integers x and y, each having $n + 1$ base b digits.
OUTPUT: the sum $x + y = (w_{n+1} w_n \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$ (c is the *carry* digit).
2. For i from 0 to n do the following:
   2.1 $w_i \leftarrow (x_i + y_i + c) \bmod b$.
   2.2 If $(x_i + y_i + c) < b$ then $c \leftarrow 0$; otherwise $c \leftarrow 1$.
3. $w_{n+1} \leftarrow c$.
4. Return($(w_{n+1} w_n \cdots w_1 w_0)$).

14.8 Note (*computational efficiency*) The base b should be chosen so that $(x_i + y_i + c) \bmod b$ can be computed by the hardware on the computing device. Some processors have instruction sets which provide an add-with-carry to facilitate multiple-precision addition.

14.9 Algorithm Multiple-precision subtraction

INPUT: positive integers x and y, each having $n+1$ base b digits, with $x \geq y$.
OUTPUT: the difference $x - y = (w_n w_{n-1} \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$.
2. For i from 0 to n do the following:
   2.1 $w_i \leftarrow (x_i - y_i + c) \bmod b$.
   2.2 If $(x_i - y_i + c) \geq 0$ then $c \leftarrow 0$; otherwise $c \leftarrow -1$.
3. Return($(w_n w_{n-1} \cdots w_1 w_0)$).

14.10 Note (*eliminating the requirement $x \geq y$*) If the relative magnitudes of the integers x and y are unknown, then Algorithm 14.9 can be modified as follows. On termination of the algorithm, if $c = -1$, then repeat Algorithm 14.9 with $x = (00 \cdots 00)_b$ and $y = (w_n w_{n-1} \cdots w_1 w_0)_b$. Conditional checking on the relative magnitudes of x and y can also be avoided by using a complement representation (§14.2.1(ii)).

14.11 Example (*modified subtraction*) Let $x = 3996879$ and $y = 4637923$ in base 10, so that $x < y$. Table 14.2 shows the steps of the modified subtraction algorithm (cf. Note 14.10). □

First execution of Algorithm 14.9							
i	6	5	4	3	2	1	0
x_i	3	9	9	6	8	7	9
y_i	4	6	3	7	9	2	3
w_i	9	3	5	8	9	5	6
c	−1	0	0	−1	−1	0	0

Second execution of Algorithm 14.9							
i	6	5	4	3	2	1	0
x_i	0	0	0	0	0	0	0
y_i	9	3	5	8	9	5	6
w_i	0	6	4	1	0	4	4
c	−1	−1	−1	−1	−1	−1	−1

Table 14.2: *Modified subtraction (see Example 14.11).*

14.2.3 Multiplication

Let x and y be integers expressed in radix b representation: $x = (x_n x_{n-1} \cdots x_1 x_0)_b$ and $y = (y_t y_{t-1} \cdots y_1 y_0)_b$. The product $x \cdot y$ will have at most $(n+t+2)$ base b digits. Algorithm 14.12 is a reorganization of the standard pencil-and-paper method taught in grade school. A *single-precision* multiplication means the multiplication of two base b digits. If x_j and y_i are two base b digits, then $x_j \cdot y_i$ can be written as $x_j \cdot y_i = (uv)_b$, where u and v are base b digits, and u may be 0.

14.12 Algorithm Multiple-precision multiplication

INPUT: positive integers x and y having $n+1$ and $t+1$ base b digits, respectively.
OUTPUT: the product $x \cdot y = (w_{n+t+1} \cdots w_1 w_0)_b$ in radix b representation.

1. For i from 0 to $(n+t+1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to t do the following:
   2.1 $c \leftarrow 0$.
   2.2 For j from 0 to n do the following:
   Compute $(uv)_b = w_{i+j} + x_j \cdot y_i + c$, and set $w_{i+j} \leftarrow v$, $c \leftarrow u$.
   2.3 $w_{i+n+1} \leftarrow u$.
3. Return($(w_{n+t+1} \cdots w_1 w_0)$).

14.13 Example (*multiple-precision multiplication*) Take $x = x_3x_2x_1x_0 = 9274$ and $y = y_2y_1y_0 = 847$ (base 10 representations), so that $n = 3$ and $t = 2$. Table 14.3 shows the steps performed by Algorithm 14.12 to compute $x \cdot y = 7855078$. □

i	j	c	$w_{i+j} + x_jy_i + c$	u	v	w_6	w_5	w_4	w_3	w_2	w_1	w_0
0	0	0	$0 + 28 + 0$	2	8	0	0	0	0	0	0	8
	1	2	$0 + 49 + 2$	5	1	0	0	0	0	0	1	8
	2	5	$0 + 14 + 5$	1	9	0	0	0	0	9	1	8
	3	1	$0 + 63 + 1$	6	4	0	0	6	4	9	1	8
1	0	0	$1 + 16 + 0$	1	7	0	0	6	4	9	7	8
	1	1	$9 + 28 + 1$	3	8	0	0	6	4	8	7	8
	2	3	$4 + 8 + 3$	1	5	0	0	6	5	8	7	8
	3	1	$6 + 36 + 1$	4	3	0	4	3	5	8	7	8
2	0	0	$8 + 32 + 0$	4	0	0	4	3	5	0	7	8
	1	4	$5 + 56 + 4$	6	5	0	4	3	5	0	7	8
	2	6	$3 + 16 + 6$	2	5	0	4	5	5	0	7	8
	3	2	$4 + 72 + 2$	7	8	7	8	5	5	0	7	8

Table 14.3: *Multiple-precision multiplication (see Example 14.13).*

14.14 Remark (*pencil-and-paper method*) The pencil-and-paper method for multiplying $x = 9274$ and $y = 847$ would appear as

$$\begin{array}{ccccccccl}
 & & & 9 & 2 & 7 & 4 & & \\
 & & & \times & 8 & 4 & 7 & & \\
\cline{3-7}
 & & 6 & 4 & 9 & 1 & 8 & & \text{(row 1)} \\
 & 3 & 7 & 0 & 9 & 6 & & & \text{(row 2)} \\
7 & 4 & 1 & 9 & 2 & & & & \text{(row 3)} \\
\hline
7 & 8 & 5 & 5 & 0 & 7 & 8 & &
\end{array}$$

The shaded entries in Table 14.3 correspond to row 1, row 1 + row 2, and row 1 + row 2 + row 3, respectively.

14.15 Note (*computational efficiency of Algorithm 14.12*)

(i) The computationally intensive portion of Algorithm 14.12 is step 2.2. Computing $w_{i+j} + x_j \cdot y_i + c$ is called the *inner-product operation*. Since w_{i+j}, x_j, y_i and c are all base b digits, the result of an inner-product operation is at most $(b-1) + (b-1)^2 + (b-1) = b^2 - 1$ and, hence, can be represented by two base b digits.

(ii) Algorithm 14.12 requires $(n+1)(t+1)$ single-precision multiplications.

(iii) It is assumed in Algorithm 14.12 that single-precision multiplications are part of the instruction set on a processor. The quality of the implementation of this instruction is crucial to an efficient implementation of Algorithm 14.12.

14.2.4 Squaring

In the preceding algorithms, $(uv)_b$ has both u and v as single-precision integers. This notation is abused in this subsection by permitting u to be a double-precision integer, such that $0 \le u \le 2(b-1)$. The value v will always be single-precision.

14.16 Algorithm Multiple-precision squaring

INPUT: positive integer $x = (x_{t-1}x_{t-2} \cdots x_1x_0)_b$.
OUTPUT: $x \cdot x = x^2$ in radix b representation.

1. For i from 0 to $(2t-1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to $(t-1)$ do the following:
   2.1 $(uv)_b \leftarrow w_{2i} + x_i \cdot x_i,\ w_{2i} \leftarrow v,\ c \leftarrow u$.
   2.2 For j from $(i+1)$ to $(t-1)$ do the following:
   $(uv)_b \leftarrow w_{i+j} + 2x_j \cdot x_i + c,\ w_{i+j} \leftarrow v,\ c \leftarrow u$.
   2.3 $w_{i+t} \leftarrow u$.
3. Return($(w_{2t-1}w_{2t-2} \ldots w_1w_0)_b$).

14.17 Note (*computational efficiency of Algorithm 14.16*)

(i) (*overflow*) In step 2.2, u can be larger than a single-precision integer. Since w_{i+j} is always set to v, $w_{i+j} \leq b-1$. If $c \leq 2(b-1)$, then $w_{i+j} + 2x_jx_i + c \leq (b-1) + 2(b-1)^2 + 2(b-1) = (b-1)(2b+1)$, implying $0 \leq u \leq 2(b-1)$. This value of u may exceed single-precision, and must be accommodated.

(ii) (*number of operations*) The computationally intensive part of the algorithm is step 2. The number of single-precision multiplications is about $(t^2+t)/2$, discounting the multiplication by 2. This is approximately one half of the single-precision multiplications required by Algorithm 14.12 (cf. Note 14.15(ii)).

14.18 Note (*squaring vs. multiplication in general*) Squaring a positive integer x (i.e., computing x^2) can at best be no more than twice as fast as multiplying distinct integers x and y. To see this, consider the identity $xy = ((x+y)^2 - (x-y)^2)/4$. Hence, $x \cdot y$ can be computed with two squarings (i.e., $(x+y)^2$ and $(x-y)^2$). Of course, a speed-up by a factor of 2 can be significant in many applications.

14.19 Example (*squaring*) Table 14.4 shows the steps performed by Algorithm 14.16 in squaring $x = 989$. Here, $t = 3$ and $b = 10$. □

i	j	$w_{2i}+x_i^2$	$w_{i+j}+2x_jx_i+c$	u	v	w_5	w_4	w_3	w_2	w_1	w_0
0	–	$0+81$	–	8	1	0	0	0	0	0	1
	1	–	$0+2\cdot8\cdot9+8$	15	2	0	0	0	0	2	1
	2	–	$0+2\cdot9\cdot9+15$	17	7	0	0	0	7	2	1
				17	7	0	0	17	7	2	1
1	–	$7+64$	–	7	1	0	0	17	1	2	1
	2	–	$17+2\cdot9\cdot8+7$	16	8	0	0	8	1	2	1
				16	8	0	16	8	1	2	1
2	–	$16+81$	–	9	7	0	7	8	1	2	1
				9	7	9	7	8	1	2	1

Table 14.4: *Multiple-precision squaring (see Example 14.19).*

14.2.5 Division

Division is the most complicated and costly of the basic multiple-precision operations. Algorithm 14.20 computes the quotient q and remainder r in radix b representation when x is divided by y.

14.20 Algorithm Multiple-precision division

INPUT: positive integers $x = (x_n \cdots x_1 x_0)_b$, $y = (y_t \cdots y_1 y_0)_b$ with $n \geq t \geq 1$, $y_t \neq 0$.
OUTPUT: the quotient $q = (q_{n-t} \cdots q_1 q_0)_b$ and remainder $r = (r_t \cdots r_1 r_0)_b$ such that $x = qy + r$, $0 \leq r < y$.

1. For j from 0 to $(n-t)$ do: $q_j \leftarrow 0$.
2. While $(x \geq yb^{n-t})$ do the following: $q_{n-t} \leftarrow q_{n-t} + 1$, $x \leftarrow x - yb^{n-t}$.
3. For i from n down to $(t+1)$ do the following:
   - 3.1 If $x_i = y_t$ then set $q_{i-t-1} \leftarrow b - 1$; otherwise set $q_{i-t-1} \leftarrow \lfloor (x_i b + x_{i-1})/y_t \rfloor$.
   - 3.2 While $(q_{i-t-1}(y_t b + y_{t-1}) > x_i b^2 + x_{i-1} b + x_{i-2})$ do: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
   - 3.3 $x \leftarrow x - q_{i-t-1} y b^{i-t-1}$.
   - 3.4 If $x < 0$ then set $x \leftarrow x + yb^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
4. $r \leftarrow x$.
5. Return(q,r).

14.21 Example (*multiple-precision division*) Let $x = 721948327$, $y = 84461$, so that $n = 8$ and $t = 4$. Table 14.5 illustrates the steps in Algorithm 14.20. The last row gives the quotient $q = 8547$ and the remainder $r = 60160$. □

i	q_4	q_3	q_2	q_1	q_0	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
–	0	0	0	0	0	7	2	1	9	4	8	3	2	7
8	0	9	0	0	0	7	2	1	9	4	8	3	2	7
		8	0	0	0		4	6	2	6	0	3	2	7
7		8	5	0	0			4	0	2	9	8	2	7
6		8	5	5	0			4	0	2	9	8	2	7
		8	5	4	0				6	5	1	3	8	7
5		8	5	4	8				6	5	1	3	8	7
		8	5	4	7					6	0	1	6	0

Table 14.5: *Multiple-precision division (see Example 14.21).*

14.22 Note (*comments on Algorithm 14.20*)

(i) Step 2 of Algorithm 14.20 is performed at most once if $y_t \geq \lfloor \frac{b}{2} \rfloor$ and b is even.
(ii) The condition $n \geq t \geq 1$ can be replaced by $n \geq t \geq 0$, provided one takes $x_j = y_j = 0$ whenever a subscript $j < 0$ in encountered in the algorithm.

14.23 Note (*normalization*) The estimate for the quotient digit q_{i-t-1} in step 3.1 of Algorithm 14.20 is never less than the true value of the quotient digit. Furthermore, if $y_t \geq \lfloor \frac{b}{2} \rfloor$, then step 3.2 is repeated no more than twice. If step 3.1 is modified so that $q_{i-t-1} \leftarrow \lfloor (x_i b^2 + x_{i-1} b + x_{i-2})/(y_t b + y_{t-1}) \rfloor$, then the estimate is almost always correct and step 3.2 is never repeated more than once. One can always guarantee that $y_t \geq \lfloor \frac{b}{2} \rfloor$ by replacing the integers x, y by λx, λy for some suitable choice of λ. The quotient of λx divided by λy is the same as that of x by y; the remainder is λ times the remainder of x divided by y. If the base b is a power of 2 (as in many applications), then the choice of λ should be a power of 2; multiplication by λ is achieved by simply left-shifting the binary representations of x and y. Multiplying by a suitable choice of λ to ensure that $y_t \geq \lfloor \frac{b}{2} \rfloor$ is called *normalization*. Example 14.24 illustrates the procedure.

14.24 Example (*normalized division*) Take $x = 73418$ and $y = 267$. Normalize x and y by multiplying each by $\lambda = 3$: $x' = 3x = 220254$ and $y' = 3y = 801$. Table 14.6 shows the steps of Algorithm 14.20 as applied to x' and y'. When x' is divided by y', the quotient is 274, and the remainder is 780. When x is divided by y, the quotient is also 274 and the remainder is $780/3 = 260$. □

i	q_3	q_2	q_1	q_0	x_5	x_4	x_3	x_2	x_1	x_0
–	0	0	0	0	2	2	0	2	5	4
5	0	2	0	0		6	0	0	5	4
4		2	7	0			3	9	8	4
3		2	7	4				7	8	0

Table 14.6: *Multiple-precision division after normalization (see Example 14.24).*

14.25 Note (*computational efficiency of Algorithm 14.20 with normalization*)

(i) *(multiplication count)* Assuming that normalization extends the number of digits in x by 1, each iteration of step 3 requires $1 + (t + 2) = t + 3$ single-precision multiplications. Hence, Algorithm 14.20 with normalization requires about $(n - t)(t + 3)$ single-precision multiplications.

(ii) *(division count)* Since step 3.1 of Algorithm 14.20 is executed $n - t$ times, at most $n - t$ single-precision divisions are required when normalization is used.

14.3 Multiple-precision modular arithmetic

§14.2 provided methods for carrying out the basic operations (addition, subtraction, multiplication, squaring, and division) with multiple-precision integers. This section deals with these operations in $\mathbb{Z}_m$, the integers modulo m, where m is a multiple-precision positive integer. (See §2.4.3 for definitions of $\mathbb{Z}_m$ and related operations.)

Let $m = (m_n m_{n-1} \cdots m_1 m_0)_b$ be a positive integer in radix b representation. Let $x = (x_n x_{n-1} \cdots x_1 x_0)_b$ and $y = (y_n y_{n-1} \cdots y_1 y_0)_b$ be non-negative integers in base b representation such that $x < m$ and $y < m$. Methods described in this section are for computing $x + y \bmod m$ (*modular addition*), $x - y \bmod m$ (*modular subtraction*), and $x \cdot y \bmod m$ (*modular multiplication*). Computing $x^{-1} \bmod m$ (*modular inversion*) is addressed in §14.4.3.

14.26 Definition If z is any integer, then $z \bmod m$ (the integer remainder in the range $[0, m-1]$ after z is divided by m) is called the *modular reduction* of z with respect to modulus m.

Modular addition and subtraction

As is the case for ordinary multiple-precision operations, addition and subtraction are the simplest to compute of the modular operations.

14.27 Fact Let x and y be non-negative integers with $x, y < m$. Then:
(i) $x + y < 2m$;
(ii) if $x \geq y$, then $0 \leq x - y < m$; and
(iii) if $x < y$, then $0 \leq x + m - y < m$.

If $x, y \in \mathbb{Z}_m$, then modular addition can be performed by using Algorithm 14.7 to add x and y as multiple-precision integers, with the additional step of subtracting m if (and only if) $x + y \geq m$. Modular subtraction is precisely Algorithm 14.9, provided $x \geq y$.

14.3.1 Classical modular multiplication

Modular multiplication is more involved than multiple-precision multiplication (§14.2.3), requiring both multiple-precision multiplication and some method for performing modular reduction (Definition 14.26). The most straightforward method for performing modular reduction is to compute the remainder on division by m, using a multiple-precision division algorithm such as Algorithm 14.20; this is commonly referred to as the *classical algorithm* for performing modular multiplication.

14.28 Algorithm Classical modular multiplication

INPUT: two positive integers x, y and a modulus m, all in radix b representation.
OUTPUT: $x \cdot y \bmod m$.
1. Compute $x \cdot y$ (using Algorithm 14.12).
2. Compute the remainder r when $x \cdot y$ is divided by m (using Algorithm 14.20).
3. Return(r).

14.3.2 Montgomery reduction

Montgomery reduction is a technique which allows efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step.

Let m be a positive integer, and let R and T be integers such that $R > m$, $\gcd(m, R) = 1$, and $0 \leq T < mR$. A method is described for computing $TR^{-1} \bmod m$ without using the classical method of Algorithm 14.28. $TR^{-1} \bmod m$ is called a *Montgomery reduction* of T modulo m *with respect to* R. With a suitable choice of R, a Montgomery reduction can be efficiently computed.

Suppose x and y are integers such that $0 \leq x, y < m$. Let $\widetilde{x} = xR \bmod m$ and $\widetilde{y} = yR \bmod m$. The Montgomery reduction of $\widetilde{x}\widetilde{y}$ is $\widetilde{x}\widetilde{y}R^{-1} \bmod m = xyR \bmod m$. This observation is used in Algorithm 14.94 to provide an efficient method for modular exponentiation.

To briefly illustrate, consider computing $x^5 \bmod m$ for some integer x, $1 \leq x < m$. First compute $\widetilde{x} = xR \bmod m$. Then compute the Montgomery reduction of $\widetilde{x}\widetilde{x}$, which is $A = \widetilde{x}^2R^{-1} \bmod m$. The Montgomery reduction of A^2 is $A^2R^{-1} \bmod m = \widetilde{x}^4R^{-3} \bmod m$. Finally, the Montgomery reduction of $(A^2R^{-1} \bmod m)\widetilde{x}$ is $(A^2R^{-1})\widetilde{x}R^{-1} \bmod m = \widetilde{x}^5R^{-4} \bmod m = x^5R \bmod m$. Multiplying this value by $R^{-1} \bmod m$ and reducing modulo m gives $x^5 \bmod m$. Provided that Montgomery reductions are more efficient to compute than classical modular reductions, this method may be more efficient than computing $x^5 \bmod m$ by repeated application of Algorithm 14.28.

If m is represented as a base b integer of length n, then a typical choice for R is b^n. The condition $R > m$ is clearly satisfied, but $\gcd(R, m) = 1$ will hold only if $\gcd(b, m) = 1$. Thus, this choice of R is not possible for all moduli. For those moduli of practical interest (such as RSA moduli), m will be odd; then b can be a power of 2 and $R = b^n$ will suffice.

Fact 14.29 is basic to the Montgomery reduction method. Note 14.30 then implies that $R = b^n$ is sufficient (but not necessary) for efficient implementation.

14.29 Fact (*Montgomery reduction*) Given integers m and R where $\gcd(m, R) = 1$, let $m' = -m^{-1} \bmod R$, and let T be any integer such that $0 \le T < mR$. If $U = Tm' \bmod R$, then $(T + Um)/R$ is an integer and $(T + Um)/R \equiv TR^{-1} \pmod{m}$.

Justification. $T + Um \equiv T \pmod{m}$ and, hence, $(T + Um)R^{-1} \equiv TR^{-1} \pmod{m}$. To see that $(T+Um)R^{-1}$ is an integer, observe that $U = Tm' + kR$ and $m'm = -1 + lR$ for some integers k and l. It follows that $(T + Um)/R = (T + (Tm' + kR)m)/R = (T + T(-1 + lR) + kRm)/R = lT + km$.

14.30 Note (*implications of Fact 14.29*)

(i) $(T + Um)/R$ is an estimate for $TR^{-1} \bmod m$. Since $T < mR$ and $U < R$, then $(T+Um)/R < (mR+mR)/R = 2m$. Thus either $(T+Um)/R = TR^{-1} \bmod m$ or $(T+Um)/R = (TR^{-1} \bmod m)+m$ (i.e., the estimate is within m of the residue). Example 14.31 illustrates that both possibilities can occur.

(ii) If all integers are represented in radix b and $R = b^n$, then $TR^{-1} \bmod m$ can be computed with two multiple-precision multiplications (i.e., $U = T \cdot m'$ and $U \cdot m$) and simple right-shifts of $T + Um$ in order to divide by R.

14.31 Example (*Montgomery reduction*) Let $m = 187, R = 190$. Then $R^{-1} \bmod m = 125$, $m^{-1} \bmod R = 63$, and $m' = 127$. If $T = 563$, then $U = Tm' \bmod R = 61$ and $(T + Um)/R = 63 = TR^{-1} \bmod m$. If $T = 1125$ then $U = Tm' \bmod R = 185$ and $(T + Um)/R = 188 = (TR^{-1} \bmod m) + m$. □

Algorithm 14.32 computes the Montgomery reduction of $T = (t_{2n-1} \cdots t_1 t_0)_b$ when $R = b^n$ and $m = (m_{n-1} \cdots m_1 m_0)_b$. The algorithm makes implicit use of Fact 14.29 by computing quantities which have similar properties to $U = Tm' \bmod R$ and $T + Um$, although the latter two expressions are not computed explicitly.

14.32 Algorithm Montgomery reduction

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$ with $\gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$, and $T = (t_{2n-1} \cdots t_1 t_0)_b < mR$.
OUTPUT: $TR^{-1} \bmod m$.

1. $A \leftarrow T$. (Notation: $A = (a_{2n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n - 1)$ do the following:
   2.1 $u_i \leftarrow a_i m' \bmod b$.
   2.2 $A \leftarrow A + u_i m b^i$.
3. $A \leftarrow A/b^n$.
4. If $A \ge m$ then $A \leftarrow A - m$.
5. Return(A).

14.33 Note (*comments on Montgomery reduction*)

(i) Algorithm 14.32 does not require $m' = -m^{-1} \bmod R$, as Fact 14.29 does, but rather $m' = -m^{-1} \bmod b$. This is due to the choice of $R = b^n$.

(ii) At step 2.1 of the algorithm with $i = l$, A has the property that $a_j = 0, 0 \le j \le l-1$. Step 2.2 does not modify these values, but does replace a_l by 0. It follows that in step 3, A is divisible by b^n.

(iii) Going into step 3, the value of A equals T plus some multiple of m (see step 2.2); here $A = (T + km)/b^n$ is an integer (see (ii) above) and $A \equiv TR^{-1} \pmod{m}$. It remains to show that A is less than $2m$, so that at step 4, a subtraction (rather than a division) will suffice. Going into step 3, $A = T + \sum_{i=0}^{n-1} u_i b^i m$. But $\sum_{i=0}^{n-1} u_i b^i m < b^n m = Rm$ and $T < Rm$; hence, $A < 2Rm$. Going into step 4 (after division of A by R), $A < 2m$ as required.

14.34 Note (*computational efficiency of Montgomery reduction*) Step 2.1 and step 2.2 of Algorithm 14.32 require a total of $n + 1$ single-precision multiplications. Since these steps are executed n times, the total number of single-precision multiplications is $n(n + 1)$. Algorithm 14.32 does not require any single-precision divisions.

14.35 Example (*Montgomery reduction*) Let $m = 72639$, $b = 10$, $R = 10^5$, and $T = 7118368$. Here $n = 5$, $m' = -m^{-1} \bmod 10 = 1$, $T \bmod m = 72385$, and $TR^{-1} \bmod m = 39796$. Table 14.7 displays the iterations of step 2 in Algorithm 14.32. □

i	$u_i = a_i m' \bmod 10$	$u_i m b^i$	A
–	–	–	7118368
0	8	581112	7699480
1	8	5811120	13510600
2	6	43583400	57094000
3	4	290556000	347650000
4	5	3631950000	3979600000

Table 14.7: *Montgomery reduction algorithm (see Example 14.35).*

Montgomery multiplication

Algorithm 14.36 combines Montgomery reduction (Algorithm 14.32) and multiple-precision multiplication (Algorithm 14.12) to compute the Montgomery reduction of the product of two integers.

14.36 Algorithm Montgomery multiplication

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \le x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.
OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n - 1)$ do the following:
   2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
   2.2 $A \leftarrow (A + x_i y + u_i m)/b$.
3. If $A \ge m$ then $A \leftarrow A - m$.
4. Return(A).

14.37 Note (*partial justification of Algorithm 14.36*) Suppose at the i^{th} iteration of step 2 that $0 \le A < 2m-1$. Step 2.2 replaces A with $(A+x_iy+u_im)/b$; but $(A+x_iy+u_im)/b \le (2m-2+(b-1)(m-1)+(b-1)m)/b = 2m-1-(1/b)$. Hence, $A < 2m-1$, justifying step 3.

14.38 Note (*computational efficiency of Algorithm 14.36*) Since $A+x_iy+u_im$ is a multiple of b, only a right-shift is required to perform a division by b in step 2.2. Step 2.1 requires two single-precision multiplications and step 2.2 requires $2n$. Since step 2 is executed n times, the total number of single-precision multiplications is $n(2+2n) = 2n(n+1)$.

14.39 Note (*computing xy mod m with Montgomery multiplication*) Suppose x, y, and m are n-digit base b integers with $0 \le x, y < m$. Neglecting the cost of the precomputation in the input, Algorithm 14.36 computes $xyR^{-1} \bmod m$ with $2n(n+1)$ single-precision multiplications. Neglecting the cost to compute $R^2 \bmod m$ and applying Algorithm 14.36 to $xyR^{-1} \bmod m$ and $R^2 \bmod m$, $xy \bmod m$ is computed in $4n(n+1)$ single-precision operations. Using classical modular multiplication (Algorithm 14.28) would require $2n(n+1)$ single-precision operations and no precomputation. Hence, the classical algorithm is superior for doing a single modular multiplication; however, Montgomery multiplication is very effective for performing modular exponentiation (Algorithm 14.94).

14.40 Remark (*Montgomery reduction vs. Montgomery multiplication*) Algorithm 14.36 (Montgomery multiplication) takes as input two n-digit numbers and then proceeds to interleave the multiplication and reduction steps. Because of this, Algorithm 14.36 is not able to take advantage of the special case where the input integers are equal (i.e., squaring). On the other hand, Algorithm 14.32 (Montgomery reduction) assumes as input the product of two integers, each of which has at most n digits. Since Algorithm 14.32 is independent of multiple-precision multiplication, a faster squaring algorithm such as Algorithm 14.16 may be used prior to the reduction step.

14.41 Example (*Montgomery multiplication*) In Algorithm 14.36, let $m = 72639$, $R = 10^5$, $x = 5792$, $y = 1229$. Here $n = 5$, $m' = -m^{-1} \bmod 10 = 1$, and $xyR^{-1} \bmod m = 39796$. Notice that m and R are the same values as in Example 14.35, as is $xy = 7118368$. Table 14.8 displays the steps in Algorithm 14.36. □

i	x_i	x_iy_0	u_i	x_iy	u_im	A
0	2	18	8	2458	581112	58357
1	9	81	8	11061	581112	65053
2	7	63	6	8603	435834	50949
3	5	45	4	6145	290556	34765
4	0	0	5	0	363195	39796

Table 14.8: *Montgomery multiplication (see Example 14.41).*

14.3.3 Barrett reduction

Barrett reduction (Algorithm 14.42) computes $r = x \bmod m$ given x and m. The algorithm requires the precomputation of the quantity $\mu = \lfloor b^{2k}/m \rfloor$; it is advantageous if many reductions are performed with a single modulus. For example, each RSA encryption for one entity requires reduction modulo that entity's public key modulus. The precomputation takes a fixed amount of work, which is negligible in comparison to modular exponentiation cost. Typically, the radix b is chosen to be close to the word-size of the processor. Hence, assume $b > 3$ in Algorithm 14.42 (see Note 14.44 (ii)).

14.42 Algorithm Barrett modular reduction

INPUT: positive integers $x = (x_{2k-1} \cdots x_1 x_0)_b$, $m = (m_{k-1} \cdots m_1 m_0)_b$ (with $m_{k-1} \neq 0$), and $\mu = \lfloor b^{2k}/m \rfloor$.
OUTPUT: $r = x \bmod m$.

1. $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$.
2. $r_1 \leftarrow x \bmod b^{k+1}$, $r_2 \leftarrow q_3 \cdot m \bmod b^{k+1}$, $r \leftarrow r_1 - r_2$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \geq m$ do: $r \leftarrow r - m$.
5. Return(r).

14.43 Fact By the division algorithm (Definition 2.82), there exist integers Q and R such that $x = Qm + R$ and $0 \leq R < m$. In step 1 of Algorithm 14.42, the following inequality is satisfied: $Q - 2 \leq q_3 \leq Q$.

14.44 Note (*partial justification of correctness of Barrett reduction*)

(i) Algorithm 14.42 is based on the observation that $\lfloor x/m \rfloor$ can be written as $Q = \lfloor (x/b^{k-1})(b^{2k}/m)(1/b^{k+1}) \rfloor$. Moreover, Q can be approximated by the quantity $q_3 = \left\lfloor \lfloor x/b^{k-1} \rfloor \mu / b^{k+1} \right\rfloor$. Fact 14.43 guarantees that q_3 is never larger than the true quotient Q, and is at most 2 smaller.

(ii) In step 2, observe that $-b^{k+1} < r_1 - r_2 < b^{k+1}$, $r_1 - r_2 \equiv (Q - q_3)m + R \pmod{b^{k+1}}$, and $0 \leq (Q - q_3)m + R < 3m < b^{k+1}$ since $m < b^k$ and $3 < b$. If $r_1 - r_2 \geq 0$, then $r_1 - r_2 = (Q - q_3)m + R$. If $r_1 - r_2 < 0$, then $r_1 - r_2 + b^{k+1} = (Q - q_3)m + R$. In either case, step 4 is repeated at most twice since $0 \leq r < 3m$.

14.45 Note (*computational efficiency of Barrett reduction*)

(i) All divisions performed in Algorithm 14.42 are simple right-shifts of the base b representation.

(ii) q_2 is only used to compute q_3. Since the $k + 1$ least significant digits of q_2 are not needed to determine q_3, only a partial multiple-precision multiplication (i.e., $q_1 \cdot \mu$) is necessary. The only influence of the $k + 1$ least significant digits on the higher order digits is the carry from position $k + 1$ to position $k + 2$. Provided the base b is sufficiently large with respect to k, this carry can be accurately computed by only calculating the digits at positions k and $k{+}1$. [1] Hence, the $k{-}1$ least significant digits of q_2 need not be computed. Since μ and q_1 have at most $k{+}1$ digits, determining q_3 requires at most $(k + 1)^2 - \binom{k}{2} = (k^2 + 5k + 2)/2$ single-precision multiplications.

(iii) In step 2 of Algorithm 14.42, r_2 can also be computed by a partial multiple-precision multiplication which evaluates only the least significant $k + 1$ digits of $q_3 \cdot m$. This can be done in at most $\binom{k+1}{2} + k$ single-precision multiplications.

14.46 Example (*Barrett reduction*) Let $b = 4$, $k = 3$, $x = (313221)_b$, and $m = (233)_b$ (i.e., $x = 3561$ and $m = 47$). Then $\mu = \lfloor 4^6/m \rfloor = 87 = (1113)_b$, $q_1 = \lfloor (313221)_b/4^2 \rfloor = (3132)_b$, $q_2 = (3132)_b \cdot (1113)_b = (10231302)_b$, $q_3 = (1023)_b$, $r_1 = (3221)_b$, $r_2 = (1023)_b \cdot (233)_b \bmod b^4 = (3011)_b$, and $r = r_1 - r_2 = (210)_b$. Thus $x \bmod m = 36$. □

[1] If $b > k$, then the carry computed by simply considering the digits at position $k - 1$ (and ignoring the carry from position $k - 2$) will be in error by at most 1.

14.3.4 Reduction methods for moduli of special form

When the modulus has a special (customized) form, reduction techniques can be employed to allow more efficient computation. Suppose that the modulus m is a t-digit base b positive integer of the form $m = b^t - c$, where c is an l-digit base b positive integer (for some $l < t$). Algorithm 14.47 computes $x \bmod m$ for any positive integer x by using only shifts, additions, and single-precision multiplications of base b numbers.

14.47 Algorithm Reduction modulo $m = b^t - c$

INPUT: a base b, positive integer x, and a modulus $m = b^t - c$, where c is an l-digit base b integer for some $l < t$.
OUTPUT: $r = x \bmod m$.

1. $q_0 \leftarrow \lfloor x/b^t \rfloor,\ r_0 \leftarrow x - q_0 b^t,\ r \leftarrow r_0,\ i \leftarrow 0$.
2. While $q_i > 0$ do the following:
   - 2.1 $q_{i+1} \leftarrow \lfloor q_i c/b^t \rfloor,\ r_{i+1} \leftarrow q_i c - q_{i+1} b^t$.
   - 2.2 $i \leftarrow i + 1,\ r \leftarrow r + r_i$.
3. While $r \geq m$ do: $r \leftarrow r - m$.
4. Return(r).

14.48 Example (*reduction modulo $b^t - c$*) Let $b = 4$, $m = 935 = (32213)_4$, and $x = 31085 = (13211231)_4$. Since $m = 4^5 - (1121)_4$, take $c = (1121)_4$. Here $t = 5$ and $l = 4$. Table 14.9 displays the quotients and remainders produced by Algorithm 14.47. At the beginning of step 3, $r = (102031)_4$. Since $r > m$, step 3 computes $r - m = (3212)_4$. □

i	$q_{i-1}c$	q_i	r_i	r
0	–	$(132)_4$	$(11231)_4$	$(11231)_4$
1	$(221232)_4$	$(2)_4$	$(21232)_4$	$(33123)_4$
2	$(2302)_4$	$(0)_4$	$(2302)_4$	$(102031)_4$

Table 14.9: *Reduction modulo $m = b^t - c$ (see Example 14.48).*

14.49 Fact (*termination*) For some integer $s \geq 0$, $q_s = 0$; hence, Algorithm 14.47 terminates.

Justification. $q_i c = q_{i+1} b^t + r_{i+1}$, $i \geq 0$. Since $c < b^t$, $q_i = (q_{i+1} b^t/c) + (r_{i+1}/c) > q_{i+1}$. Since the q_i's are non-negative integers which strictly decrease as i increases, there is some integer $s \geq 0$ such that $q_s = 0$.

14.50 Fact (*correctness*) Algorithm 14.47 terminates with the correct residue modulo m.

Justification. Suppose that s is the smallest index i for which $q_i = 0$ (i.e., $q_s = 0$). Now, $x = q_0 b^t + r_0$ and $q_i c = q_{i+1} b^t + r_{i+1}$, $0 \leq i \leq s - 1$. Adding these equations gives $x + \left(\sum_{i=0}^{s-1} q_i\right) c = \left(\sum_{i=0}^{s-1} q_i\right) b^t + \sum_{i=0}^{s} r_i$. Since $b^t \equiv c \pmod{m}$, it follows that $x \equiv \sum_{i=0}^{s} r_i \pmod{m}$. Hence, repeated subtraction of m from $r = \sum_{i=0}^{s} r_i$ gives the correct residue.

14.51 Note (*computational efficiency of reduction modulo $b^t - c$*)

(i) Suppose that x has $2t$ base b digits. If $l \leq t/2$, then Algorithm 14.47 executes step 2 at most $s = 3$ times, requiring 2 multiplications by c. In general, if l is approximately $(s-2)t/(s-1)$, then Algorithm 14.47 executes step 2 about s times. Thus, Algorithm 14.47 requires about sl single-precision multiplications.

(ii) If c has few non-zero digits, then multiplication by c will be relatively inexpensive. If c is large but has few non-zero digits, the number of iterations of Algorithm 14.47 will be greater, but each iteration requires a very simple multiplication.

14.52 Note (*modifications*) Algorithm 14.47 can be modified if $m = b^t + c$ for some positive integer $c < b^t$: in step 2.2, replace $r \leftarrow r + r_i$ with $r \leftarrow r + (-1)^i r_i$.

14.53 Remark (*using moduli of a special form*) Selecting RSA moduli of the form $b^t \pm c$ for small values of c limits the choices of primes p and q. Care must also be exercised when selecting moduli of a special form, so that factoring is not made substantially easier; this is because numbers of this form are more susceptible to factoring by the special number field sieve (see §3.2.7). A similar statement can be made regarding the selection of primes of a special form for cryptographic schemes based on the discrete logarithm problem.

14.4 Greatest common divisor algorithms

Many situations in cryptography require the computation of the greatest common divisor (gcd) of two positive integers (see Definition 2.86). Algorithm 2.104 describes the classical Euclidean algorithm for this computation. For multiple-precision integers, Algorithm 2.104 requires a multiple-precision division at step 1.1 which is a relatively expensive operation. This section describes three methods for computing the gcd which are more efficient than the classical approach using multiple-precision numbers. The first is non-Euclidean and is referred to as the *binary gcd algorithm* (§14.4.1). Although it requires more steps than the classical algorithm, the binary gcd algorithm eliminates the computationally expensive division and replaces it with elementary shifts and additions. Lehmer's gcd algorithm (§14.4.2) is a variant of the classical algorithm more suited to multiple-precision computations. A binary version of the extended Euclidean algorithm is given in §14.4.3.

14.4.1 Binary gcd algorithm

14.54 Algorithm Binary gcd algorithm

INPUT: two positive integers x and y with $x \geq y$.
OUTPUT: $\gcd(x, y)$.

1. $g \leftarrow 1$.
2. While both x and y are even do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
3. While $x \neq 0$ do the following:
   - 3.1 While x is even do: $x \leftarrow x/2$.
   - 3.2 While y is even do: $y \leftarrow y/2$.
   - 3.3 $t \leftarrow |x - y|/2$.
   - 3.4 If $x \geq y$ then $x \leftarrow t$; otherwise, $y \leftarrow t$.
4. Return$(g \cdot y)$.

14.55 Example (*binary gcd algorithm*) The following table displays the steps performed by Algorithm 14.54 for computing $\gcd(1764, 868) = 28$. □

x	1764	441	112	7	7	7	7	7	0
y	868	217	217	217	105	49	21	7	7
g	1	4	4	4	4	4	4	4	4

14.56 Note (*computational efficiency of Algorithm 14.54*)

(i) If x and y are in radix 2 representation, then the divisions by 2 are simply right-shifts.
(ii) Step 3.3 for multiple-precision integers can be computed using Algorithm 14.9.

14.4.2 Lehmer's gcd algorithm

Algorithm 14.57 is a variant of the classical Euclidean algorithm (Algorithm 2.104) and is suited to computations involving multiple-precision integers. It replaces many of the multiple-precision divisions by simpler single-precision operations.

Let x and y be positive integers in radix b representation, with $x \geq y$. Without loss of generality, assume that x and y have the same number of base b digits throughout Algorithm 14.57; this may necessitate padding the high-order digits of y with 0's.

14.57 Algorithm Lehmer's gcd algorithm

INPUT: two positive integers x and y in radix b representation, with $x \geq y$.
OUTPUT: $\gcd(x, y)$.

1. While $y \geq b$ do the following:
   1.1 Set $\widetilde{x}$, $\widetilde{y}$ to be the high-order digit of x, y, respectively ($\widetilde{y}$ could be 0).
   1.2 $A \leftarrow 1,\ B \leftarrow 0,\ C \leftarrow 0,\ D \leftarrow 1$.
   1.3 While $(\widetilde{y} + C) \neq 0$ and $(\widetilde{y} + D) \neq 0$ do the following:
   $q \leftarrow \lfloor (\widetilde{x} + A)/(\widetilde{y} + C) \rfloor,\ q' \leftarrow \lfloor (\widetilde{x} + B)/(\widetilde{y} + D) \rfloor$.
   If $q \neq q'$ then go to step 1.4.
   $t \leftarrow A - qC,\ A \leftarrow C,\ C \leftarrow t,\ t \leftarrow B - qD,\ B \leftarrow D,\ D \leftarrow t$.
   $t \leftarrow \widetilde{x} - q\widetilde{y},\ \widetilde{x} \leftarrow \widetilde{y},\ \widetilde{y} \leftarrow t$.
   1.4 If $B = 0$, then $T \leftarrow x \bmod y,\ x \leftarrow y,\ y \leftarrow T$;
   otherwise, $T \leftarrow Ax + By,\ u \leftarrow Cx + Dy,\ x \leftarrow T,\ y \leftarrow u$.
2. Compute $v = \gcd(x, y)$ using Algorithm 2.104.
3. Return(v).

14.58 Note (*implementation notes for Algorithm 14.57*)

(i) T is a multiple-precision variable. A, B, C, D, and t are signed single-precision variables; hence, one bit of each of these variables must be reserved for the sign.
(ii) The first operation of step 1.3 may result in overflow since $0 \leq \widetilde{x} + A, \widetilde{y} + D \leq b$. This possibility needs to be accommodated. One solution is to reserve two bits more than the number of bits in a digit for each of $\widetilde{x}$ and $\widetilde{y}$ to accommodate both the sign and the possible overflow.
(iii) The multiple-precision additions of step 1.4 are actually subtractions, since $AB \leq 0$ and $CD \leq 0$.

14.59 Note (*computational efficiency of Algorithm 14.57*)

(i) Step 1.3 attempts to simulate multiple-precision divisions by much simpler single-precision operations. In each iteration of step 1.3, all computations are single precision. The number of iterations of step 1.3 depends on b.

(ii) The modular reduction in step 1.4 is a multiple-precision operation. The other operations are multiple-precision, but require only linear time since the multipliers are single precision.

14.60 Example (*Lehmer's gcd algorithm*) Let $b = 10^3$, $x = 768\,454\,923$, and $y = 542\,167\,814$. Since $b = 10^3$, the high-order digits of x and y are $\widetilde{x} = 768$ and $\widetilde{y} = 542$, respectively. Table 14.10 displays the values of the variables at various stages of Algorithm 14.57. The single-precision computations (Step 1.3) when $q = q'$ are shown in Table 14.11. Hence $\gcd(x, y) = 1$. □

14.4.3 Binary extended gcd algorithm

Given integers x and y, Algorithm 2.107 computes integers a and b such that $ax + by = v$, where $v = \gcd(x, y)$. It has the drawback of requiring relatively costly multiple-precision divisions when x and y are multiple-precision integers. Algorithm 14.61 eliminates this requirement at the expense of more iterations.

14.61 Algorithm Binary extended gcd algorithm

INPUT: two positive integers x and y.
OUTPUT: integers a, b, and v such that $ax + by = v$, where $v = \gcd(x, y)$.

1. $g \leftarrow 1$.
2. While x and y are both even, do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
3. $u \leftarrow x$, $v \leftarrow y$, $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.
4. While u is even do the following:
   - 4.1 $u \leftarrow u/2$.
   - 4.2 If $A \equiv B \equiv 0 \pmod 2$ then $A \leftarrow A/2$, $B \leftarrow B/2$; otherwise, $A \leftarrow (A + y)/2$, $B \leftarrow (B - x)/2$.
5. While v is even do the following:
   - 5.1 $v \leftarrow v/2$.
   - 5.2 If $C \equiv D \equiv 0 \pmod 2$ then $C \leftarrow C/2$, $D \leftarrow D/2$; otherwise, $C \leftarrow (C + y)/2$, $D \leftarrow (D - x)/2$.
6. If $u \geq v$ then $u \leftarrow u - v$, $A \leftarrow A - C$, $B \leftarrow B - D$;
   otherwise, $v \leftarrow v - u$, $C \leftarrow C - A$, $D \leftarrow D - B$.
7. If $u = 0$, then $a \leftarrow C$, $b \leftarrow D$, and return$(a, b, g \cdot v)$; otherwise, go to step 4.

14.62 Example (*binary extended gcd algorithm*) Let $x = 693$ and $y = 609$. Table 14.12 displays the steps in Algorithm 14.61 for computing integers a, b, v such that $693a + 609b = v$, where $v = \gcd(693, 609)$. The algorithm returns $v = 21$, $a = -181$, and $b = 206$. □

x	y	q	q'	precision	reference
768 454 923	542 167 814	1	1	single	Table 14.11(i)
89 593 596	47 099 917	1	1	single	Table 14.11(ii)
42 493 679	4 606 238	10	8	multiple	
4 606 238	1 037 537	5	2	multiple	
1 037 537	456 090	–	–	multiple	
456 090	125 357	3	3	single	Table 14.11(iii)
34 681	10 657	3	3	single	Table 14.11(iv)
10 657	2 710	5	3	multiple	
2 710	2 527	1	0	multiple	
2 527	183				Algorithm 2.104
183	148				Algorithm 2.104
148	35				Algorithm 2.104
35	8				Algorithm 2.104
8	3				Algorithm 2.104
3	2				Algorithm 2.104
2	1				Algorithm 2.104
1	0				Algorithm 2.104

Table 14.10: *Lehmer's gcd algorithm (see Example 14.60).*

	$\widetilde{x}$	$\widetilde{y}$	A	B	C	D	q	q'
(i)	768	542	1	0	0	1	1	1
	542	226	0	1	1	-1	2	2
	226	90	1	-1	-2	3	2	2
	90	46	-2	3	5	-7	1	2
(ii)	89	47	1	0	0	1	1	1
	47	42	0	1	1	-1	1	1
	42	5	1	-1	-1	2	10	5
(iii)	456	125	1	0	0	1	3	3
	125	81	0	1	1	-3	1	1
	81	44	1	-3	-1	4	1	1
	44	37	-1	4	2	-7	1	1
	37	7	2	-7	-3	11	9	1
(iv)	34	10	1	0	0	1	3	3
	10	4	0	1	1	-3	2	11

Table 14.11: *Single-precision computations (see Example 14.60 and Table 14.10).*

u	v	A	B	C	D
693	609	1	0	0	1
84	609	1	-1	0	1
42	609	305	-347	0	1
21	609	457	-520	0	1
21	588	457	-520	-457	521
21	294	457	-520	76	-86
21	147	457	-520	38	-43
21	126	457	-520	-419	477
21	63	457	-520	95	-108
21	42	457	-520	-362	412
21	21	457	-520	-181	206
0	21	638	-726	-181	206

Table 14.12: *The binary extended gcd algorithm with $x = 693$, $y = 609$ (see Example 14.62).*

14.63 Note (*computational efficiency of Algorithm 14.61*)

(i) The only multiple-precision operations needed for Algorithm 14.61 are addition and subtraction. Division by 2 is simply a right-shift of the binary representation.

(ii) The number of bits needed to represent either u or v decreases by (at least) 1, after at most two iterations of steps 4 – 7; thus, the algorithm takes at most $2(\lfloor \lg x \rfloor + \lfloor \lg y \rfloor + 2)$ such iterations.

14.64 Note (*multiplicative inverses*) Given positive integers m and a, it is often necessary to find an integer $z \in \mathbb{Z}_m$ such that $az \equiv 1 \pmod{m}$, if such an integer exists. z is called the multiplicative inverse of a modulo m (see Definition 2.115). For example, constructing the private key for RSA requires the computation of an integer d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$ (see Algorithm 8.1). Algorithm 14.61 provides a computationally efficient method for determining z given a and m, by setting $x = m$ and $y = a$. If $\gcd(x, y) = 1$, then, at termination, $z = D$ if $D > 0$, or $z = m + D$ if $D < 0$; if $\gcd(x, y) \neq 1$, then a is not invertible modulo m. Notice that if m is odd, it is not necessary to compute the values of A and C. It would appear that step 4 of Algorithm 14.61 requires both A and B in order to decide which case in step 4.2 is executed. But if m is odd and B is even, then A must be even; hence, the decision can be made using the parities of B and m.

Example 14.65 illustrates Algorithm 14.61 for computing a multiplicative inverse.

14.65 Example (*multiplicative inverse*) Let $m = 383$ and $a = 271$. Table 14.13 illustrates the steps of Algorithm 14.61 for computing $271^{-1} \bmod 383 = 106$. Notice that values for the variables A and C need not be computed. □

14.5 Chinese remainder theorem for integers

Fact 2.120 introduced the Chinese remainder theorem (CRT) and Fact 2.121 outlined an algorithm for solving the associated system of linear congruences. Although the method described there is the one found in most textbooks on elementary number theory, it is not the

iteration:	1	2	3	4	5	6	7	8	9	10
u	383	112	56	28	14	7	7	7	7	7
v	271	271	271	271	271	271	264	132	66	33
B	0	-1	-192	-96	-48	-24	-24	-24	-24	-24
D	1	1	1	1	1	1	25	-179	-281	-332
iteration:	11	12	13	14	15	16	17	18	19	
u	7	7	7	7	4	2	1	1	1	
v	26	13	6	3	3	3	3	2	1	
B	-24	-24	-24	-24	41	-171	-277	-277	-277	
D	-308	-154	-130	-65	-65	-65	-65	212	106	

Table 14.13: *Inverse computation using the binary extended gcd algorithm (see Example 14.65).*

method of choice for large integers. Garner's algorithm (Algorithm 14.71) has some computational advantages. §14.5.1 describes an alternate (non-radix) representation for non-negative integers, called a *modular representation*, that allows some computational advantages compared to standard radix representations. Algorithm 14.71 provides a technique for converting numbers from modular to base b representation.

14.5.1 Residue number systems

In previous sections, non-negative integers have been represented in radix b notation. An alternate means is to use a mixed-radix representation.

14.66 Fact Let B be a fixed positive integer. Let $m_1, m_2, \ldots, m_t$ be positive integers such that $\gcd(m_i, m_j) = 1$ for all $i \neq j$, and $M = \prod_{i=1}^{t} m_i \geq B$. Then each integer x, $0 \leq x < B$, can be uniquely represented by the sequence of integers $v(x) = (v_1, v_2, \ldots, v_t)$, where $v_i = x \bmod m_i$, $1 \leq i \leq t$.

14.67 Definition Referring to Fact 14.66, $v(x)$ is called the *modular representation* or *mixed-radix representation* of x for the moduli $m_1, m_2, \ldots, m_t$. The set of modular representations for all integers x in the range $0 \leq x < B$ is called a *residue number system*.

If $v(x) = (v_1, v_2, \ldots, v_t)$ and $v(y) = (u_1, u_2, \ldots, u_t)$, define $v(x)+v(y) = (w_1, w_2, \ldots, w_t)$ where $w_i = v_i + u_i \bmod m_i$, and $v(x) \cdot v(y) = (z_1, z_2, \ldots, z_t)$ where $z_i = v_i \cdot u_i \bmod m_i$.

14.68 Fact If $0 \leq x, y < M$, then $v((x+y) \bmod M) = v(x) + v(y)$ and $v((x \cdot y) \bmod M) = v(x) \cdot v(y)$.

14.69 Example (*modular representation*) Let $M = 30 = 2 \times 3 \times 5$; here, $t = 3$, $m_1 = 2$, $m_1 = 3$, and $m_3 = 5$. Table 14.14 displays each residue modulo 30 along with its associated modular representation. As an example of Fact 14.68, note that $21 + 27 \equiv 18 \pmod{30}$ and $(101) + (102) = (003)$. Also $22 \cdot 17 \equiv 14 \pmod{30}$ and $(012) \cdot (122) = (024)$. □

14.70 Note (*computational efficiency of modular representation for RSA decryption*) Suppose that $n = pq$, where p and q are distinct primes. Fact 14.68 implies that $x^d \bmod n$ can be computed in a modular representation as $v^d(x)$; that is, if $v(x) = (v_1, v_2)$ with respect to moduli $m_1 = p$, $m_2 = q$, then $v^d(x) = (v_1^d \bmod p, v_2^d \bmod q)$. In general, computing

x	$v(x)$	x	$v(x)$	x	$v(x)$	x	$v(x)$	x	$v(x)$
0	(000)	6	(001)	12	(002)	18	(003)	24	(004)
1	(111)	7	(112)	13	(113)	19	(114)	25	(110)
2	(022)	8	(023)	14	(024)	20	(020)	26	(021)
3	(103)	9	(104)	15	(100)	21	(101)	27	(102)
4	(014)	10	(010)	16	(011)	22	(012)	28	(013)
5	(120)	11	(121)	17	(122)	23	(123)	29	(124)

Table 14.14: *Modular representations (see Example 14.69).*

$v_1^d \bmod p$ and $v_2^d \bmod q$ is faster than computing $x^d \bmod n$. For RSA, if p and q are part of the private key, modular representation can be used to improve the performance of both decryption and signature generation (see Note 14.75).

Converting an integer x from a base b representation to a modular representation is easily done by applying a modular reduction algorithm to compute $v_i = x \bmod m_i$, $1 \le i \le t$. Modular representations of integers in $\mathbb{Z}_M$ may facilitate some computational efficiencies, provided conversion from a standard radix to modular representation and back are relatively efficient operations. Algorithm 14.71 describes one way of converting from modular representation back to a standard radix representation.

14.5.2 Garner's algorithm

Garner's algorithm is an efficient method for determining x, $0 \le x < M$, given $v(x) = (v_1, v_2, \ldots, v_t)$, the residues of x modulo the pairwise co-prime moduli $m_1, m_2, \ldots, m_t$.

14.71 Algorithm Garner's algorithm for CRT

INPUT: a positive integer $M = \prod_{i=1}^{t} m_i > 1$, with $\gcd(m_i, m_j) = 1$ for all $i \ne j$, and a modular representation $v(x) = (v_1, v_2, \ldots, v_t)$ of x for the m_i.
OUTPUT: the integer x in radix b representation.

1. For i from 2 to t do the following:
   1.1 $C_i \leftarrow 1$.
   1.2 For j from 1 to $(i-1)$ do the following:
   $u \leftarrow m_j^{-1} \bmod m_i$ (use Algorithm 14.61).
   $C_i \leftarrow u \cdot C_i \bmod m_i$.
2. $u \leftarrow v_1$, $x \leftarrow u$.
3. For i from 2 to t do the following: $u \leftarrow (v_i - x)C_i \bmod m_i$, $x \leftarrow x + u \cdot \prod_{j=1}^{i-1} m_j$.
4. Return(x).

14.72 Fact x returned by Algorithm 14.71 satisfies $0 \le x < M$, $x \equiv v_i \pmod{m_i}$, $1 \le i \le t$.

14.73 Example (*Garner's algorithm*) Let $m_1 = 5$, $m_2 = 7$, $m_3 = 11$, $m_4 = 13$, $M = \prod_{i=1}^{4} m_i = 5005$, and $v(x) = (2, 1, 3, 8)$. The constants C_i computed are $C_2 = 3$, $C_3 = 6$, and $C_4 = 5$. The values of (i, u, x) computed in step 3 of Algorithm 14.71 are $(1, 2, 2)$, $(2, 4, 22)$, $(3, 7, 267)$, and $(4, 5, 2192)$. Hence, the modular representation $v(x) = (2, 1, 3, 8)$ corresponds to the integer $x = 2192$. □

14.74 Note (*computational efficiency of Algorithm 14.71*)

(i) If Garner's algorithm is used repeatedly with the same modulus M and the same factors of M, then step 1 can be considered as a precomputation, requiring the storage of $t-1$ numbers.

(ii) The classical algorithm for the CRT (Algorithm 2.121) typically requires a modular reduction with modulus M, whereas Algorithm 14.71 does not. Suppose M is a kt-bit integer and each m_i is a k-bit integer. A modular reduction by M takes $O((kt)^2)$ bit operations, whereas a modular reduction by m_i takes $O(k^2)$ bit operations. Since Algorithm 14.71 only does modular reduction with m_i, $2 \le i \le t$, it takes $O(tk^2)$ bit operations in total for the reduction phase, and is thus more efficient.

14.75 Note (*RSA decryption and signature generation*)

(i) (*special case of two moduli*) Algorithm 14.71 is particularly efficient for RSA moduli $n = pq$, where $m_1 = p$ and $m_2 = q$ are distinct primes. Step 1 computes a single value $C_2 = p^{-1} \bmod q$. Step 3 is executed once: $u = (v_2 - v_1)C_2 \bmod q$ and $x = v_1 + up$.

(ii) (*RSA exponentiation*) Suppose p and q are t-bit primes, and let $n = pq$. Let d be a $2t$-bit RSA private key. RSA decryption and signature generation compute $x^d \bmod n$ for some $x \in \mathbb{Z}_n$. Suppose that modular multiplication and squaring require k^2 bit operations for k-bit inputs, and that exponentiation with a k-bit exponent requires about $\frac{3}{2}k$ multiplications and squarings (see Note 14.78). Then computing $x^d \bmod n$ requires about $\frac{3}{2}(2t)^3 = 12t^3$ bit operations. A more efficient approach is to compute $x^{d_p} \bmod p$ and $x^{d_q} \bmod q$ (where $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$), and then use Garner's algorithm to construct $x^d \bmod pq$. Although this procedure takes two exponentiations, each is considerably more efficient because the moduli are smaller. Assuming that the cost of Algorithm 14.71 is negligible with respect to the exponentiations, computing $x^d \bmod n$ is about $\frac{3}{2}(2t)^3/2(\frac{3}{2}t^3) = 4$ times faster.

14.6 Exponentiation

One of the most important arithmetic operations for public-key cryptography is exponentiation. The RSA scheme (§8.2) requires exponentiation in $\mathbb{Z}_m$ for some positive integer m, whereas Diffie-Hellman key agreement (§12.6.1) and the ElGamal encryption scheme (§8.4) use exponentiation in $\mathbb{Z}_p$ for some large prime p. As pointed out in §8.4.2, ElGamal encryption can be generalized to any finite cyclic group. This section discusses methods for computing the *exponential* g^e, where the *base* g is an element of a finite group G (§2.5.1) and the *exponent* e is a non-negative integer. A reader uncomfortable with the setting of a general group may consider G to be $\mathbb{Z}_m^*$; that is, read g^e as $g^e \bmod m$.

An efficient method for multiplying two elements in the group G is essential to performing efficient exponentiation. The most naive way to compute g^e is to do $e-1$ multiplications in the group G. For cryptographic applications, the order of the group G typically exceeds 2^{160} elements, and may exceed 2^{1024}. Most choices of e are large enough that it would be infeasible to compute g^e using $e-1$ successive multiplications by g.

There are two ways to reduce the time required to do exponentiation. One way is to decrease the time to multiply two elements in the group; the other is to reduce the number of multiplications used to compute g^e. Ideally, one would do both.

This section considers three types of exponentiation algorithms.

1. *basic techniques for exponentiation.* Arbitrary choices of the base g and exponent e are allowed.
2. *fixed-exponent exponentiation algorithms.* The exponent e is fixed and arbitrary choices of the base g are allowed. RSA encryption and decryption schemes benefit from such algorithms.
3. *fixed-base exponentiation algorithms.* The base g is fixed and arbitrary choices of the exponent e are allowed. ElGamal encryption and signatures schemes and Diffie-Hellman key agreement protocols benefit from such algorithms.

14.6.1 Techniques for general exponentiation

This section includes general-purpose exponentiation algorithms referred to as *repeated square-and-multiply* algorithms.

(i) Basic binary and k-ary exponentiation

Algorithm 14.76 is simply Algorithm 2.143 restated in terms of an arbitrary finite abelian group G with identity element 1.

14.76 Algorithm Right-to-left binary exponentiation

INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: g^e.

1. $A \leftarrow 1$, $S \leftarrow g$.
2. While $e \neq 0$ do the following:
   2.1 If e is odd then $A \leftarrow A \cdot S$.
   2.2 $e \leftarrow \lfloor e/2 \rfloor$.
   2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
3. Return(A).

14.77 Example (*right-to-left binary exponentiation*) The following table displays the values of A, e, and S during each iteration of Algorithm 14.76 for computing g^{283}. □

A	1	g	g^3	g^3	g^{11}	g^{27}	g^{27}	g^{27}	g^{27}	g^{283}
e	283	141	70	35	17	8	4	2	1	0
S	g	g^2	g^4	g^8	g^{16}	g^{32}	g^{64}	g^{128}	g^{256}	−

14.78 Note (*computational efficiency of Algorithm 14.76*) Let $t+1$ be the bitlength of the binary representation of e, and let $\mathrm{wt}(e)$ be the number of 1's in this representation. Algorithm 14.76 performs t squarings and $\mathrm{wt}(e)-1$ multiplications. If e is randomly selected in the range $0 \leq e < |G| = n$, then about $\lfloor \lg n \rfloor$ squarings and $\frac{1}{2}(\lfloor \lg n \rfloor + 1)$ multiplications can be expected. (The assignment $1 \cdot x$ is not counted as a multiplication, nor is the operation $1 \cdot 1$ counted as a squaring.) If squaring is approximately as costly as an arbitrary multiplication (cf. Note 14.18), then the expected amount of work is roughly $\frac{3}{2}\lfloor \lg n \rfloor$ multiplications.

Algorithm 14.76 computes $A \cdot S$ whenever e is odd. For some choices of g, $A \cdot g$ can be computed more efficiently than $A \cdot S$ for arbitrary S. Algorithm 14.79 is a left-to-right binary exponentiation which replaces the operation $A \cdot S$ (for arbitrary S) by the operation $A \cdot g$ (for fixed g).

14.79 Algorithm Left-to-right binary exponentiation

INPUT: $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
OUTPUT: g^e.

1. $A \leftarrow 1$.
2. For i from t down to 0 do the following:
   - 2.1 $A \leftarrow A \cdot A$.
   - 2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return(A).

14.80 Example (*left-to-right binary exponentiation*) The following table displays the values of A during each iteration of Algorithm 14.79 for computing g^{283}. Note that $t = 8$ and $283 = (100011011)_2$. □

i	8	7	6	5	4	3	2	1	0
e_i	1	0	0	0	1	1	0	1	1
A	g	g^2	g^4	g^8	g^{17}	g^{35}	g^{70}	g^{141}	g^{283}

14.81 Note (*computational efficiency of Algorithm 14.79*) Let $t + 1$ be the bitlength of the binary representation of e, and let $\mathrm{wt}(e)$ be the number of 1's in this representation. Algorithm 14.79 performs $t + 1$ squarings and $\mathrm{wt}(e) - 1$ multiplications by g. The number of squarings and multiplications is the same as in Algorithm 14.76 but, in this algorithm, multiplication is always with the fixed value g. If g has a special structure, this multiplication may be substantially easier than multiplying two arbitrary elements. For example, a frequent operation in ElGamal public-key schemes is the computation of $g^k \bmod p$, where g is a generator of $\mathbb{Z}_p^*$ and p is a large prime number. The multiple-precision computation $A \cdot g$ can be done in linear time if g is chosen so that it can be represented by a single-precision integer (e.g., $g = 2$). If the radix b is sufficiently large, there is a high probability that such a generator exists.

Algorithm 14.82, sometimes referred to as the *window method* for exponentiation, is a generalization of Algorithm 14.79 which processes more than one bit of the exponent per iteration.

14.82 Algorithm Left-to-right k-ary exponentiation

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: g^e.

1. *Precomputation.*
   - 1.1 $g_0 \leftarrow 1$.
   - 1.2 For i from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For i from t down to 0 do the following:
   - 3.1 $A \leftarrow A^{2^k}$.
   - 3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return(A).

In Algorithm 14.83, Algorithm 14.82 is modified slightly to reduce the amount of precomputation. The following notation is used: for each i, $0 \le i \le t$, if $e_i \ne 0$, then write $e_i = 2^{h_i} u_i$ where u_i is odd; if $e_i = 0$, then let $h_i = 0$ and $u_i = 0$.

14.83 Algorithm Modified left-to-right k-ary exponentiation

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \ge 1$.
OUTPUT: g^e.

1. *Precomputation.*
   - 1.1 $g_0 \leftarrow 1$, $g_1 \leftarrow g$, $g_2 \leftarrow g^2$.
   - 1.2 For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$.
2. $A \leftarrow 1$.
3. For i from t down to 0 do: $A \leftarrow (A^{2^{k-h_i}} \cdot g_{u_i})^{2^{h_i}}$.
4. Return(A).

14.84 Remark (*right-to-left k-ary exponentiation*) Algorithm 14.82 is a generalization of Algorithm 14.79. In a similar manner, Algorithm 14.76 can be generalized to the k-ary case. However, the optimization given in Algorithm 14.83 is not possible for the generalized right-to-left k-ary exponentiation method.

(ii) Sliding-window exponentiation

Algorithm 14.85 also reduces the amount of precomputation compared to Algorithm 14.82 and, moreover, reduces the average number of multiplications performed (excluding squarings). k is called the *window size*.

14.85 Algorithm Sliding-window exponentiation

INPUT: g, $e = (e_t e_{t-1} \cdots e_1 e_0)_2$ with $e_t = 1$, and an integer $k \ge 1$.
OUTPUT: g^e.

1. *Precomputation.*
   - 1.1 $g_1 \leftarrow g$, $g_2 \leftarrow g^2$.
   - 1.2 For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$.
2. $A \leftarrow 1$, $i \leftarrow t$.
3. While $i \ge 0$ do the following:
   - 3.1 If $e_i = 0$ then do: $A \leftarrow A^2$, $i \leftarrow i - 1$.
   - 3.2 Otherwise ($e_i \ne 0$), find the longest bitstring $e_i e_{i-1} \cdots e_l$ such that $i - l + 1 \le k$ and $e_l = 1$, and do the following:

     $$A \leftarrow A^{2^{i-l+1}} \cdot g_{(e_i e_{i-1} \ldots e_l)_2}, \quad i \leftarrow l - 1.$$
4. Return(A).

14.86 Example (*sliding-window exponentiation*) Take $e = 11749 = (10110111100101)_2$ and $k = 3$. Table 14.15 illustrates the steps of Algorithm 14.85. Notice that the sliding-window method for this exponent requires three multiplications, corresponding to $i = 7$, 4, and 0. Algorithm 14.79 would have required four multiplications for the same values of k and e. □

i	A	Longest bitstring
13	1	101
10	g^5	101
7	$(g^5)^8 g^5 = g^{45}$	111
4	$(g^{45})^8 g^7 = g^{367}$	–
3	$(g^{367})^2 = g^{734}$	–
2	$(g^{734})^2 = g^{1468}$	101
0	$(g^{1468})^8 g^5 = g^{11749}$	–

Table 14.15: *Sliding-window exponentiation with $k = 3$ and exponent $e = (10110111100101)_2$.*

14.87 Note (*comparison of exponentiation algorithms*) Let $t + 1$ be the bitlength of e, and let $l + 1$ be the number of k-bit words formed from e; that is, $l = \lceil (t+1)/k \rceil - 1 = \lfloor t/k \rfloor$. Table 14.16 summarizes the number of squarings and multiplications required by Algorithms 14.76, 14.79, 14.82, and 14.83. Analysis of the number of squarings and multiplications for Algorithm 14.85 is more difficult, although it is the recommended method.

(i) (*squarings for Algorithm 14.82*) The number of squarings for Algorithm 14.82 is lk. Observe that $lk = \lfloor t/k \rfloor k = t - (t \bmod k)$. It follows that $t - (k-1) \le lk \le t$ and that Algorithm 14.82 can save up to $k - 1$ squarings over Algorithms 14.76 and 14.79. An optimal value for k in Algorithm 14.82 will depend on t.

(ii) (*squarings for Algorithm 14.83*) The number of squarings for Algorithm 14.83 is $lk + h_l$ where $0 \le h_l \le t \bmod k$. Since $t-(k-1) \le lk \le lk+h_l \le lk+(t \bmod k) = t$ or $t-(k-1) \le lk+h_l \le t$, the number of squarings for this algorithm has the same bounds as Algorithm 14.82.

	Precomputation			Multiplications	
Algorithm	sq	mult	squarings	worst case	average case
14.76	0	0	t	t	$t/2$
14.79	0	0	t	t	$t/2$
14.82	1	$2^k - 3$	$t-(k-1) \le lk \le t$	$l-1$	$l(2^k-1)/2^k$
14.83	1	$2^{k-1} - 1$	$t-(k-1) \le lk + h_l \le t$	$l-1$	$l(2^k-1)/2^k$

Table 14.16: *Number of squarings (sq) and multiplications (mult) for exponentiation algorithms.*

(iii) Simultaneous multiple exponentiation

There are a number of situations which require computation of the product of several exponentials with distinct bases and distinct exponents (for example, verification of ElGamal signatures; see Note 14.91). Rather than computing each exponential separately, Algorithm 14.88 presents a method to do them simultaneously.

Let $e_0, e_1, \ldots, e_{k-1}$ be positive integers each of bitlength t; some of the high-order bits of some of the exponents might be 0, but there is at least one e_i whose high-order bit is 1. Form a $k \times t$ array EA (called the *exponent array*) whose rows are the binary representations of the exponents e_i, $0 \le i \le k-1$. Let I_j be the non-negative integer whose binary representation is the jth column, $1 \le j \le t$, of EA, where low-order bits are at the top of the column.

14.88 Algorithm Simultaneous multiple exponentiation

INPUT: group elements $g_0, g_1, \ldots, g_{k-1}$ and non-negative t-bit integers $e_0, e_1, \ldots e_{k-1}$.
OUTPUT: $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$.

1. *Precomputation*. For i from 0 to $(2^k - 1)$: $G_i \leftarrow \prod_{j=0}^{k-1} g_j^{i_j}$ where $i = (i_{k-1} \cdots i_0)_2$.
2. $A \leftarrow 1$.
3. For i from 1 to t do the following: $A \leftarrow A \cdot A$, $A \leftarrow A \cdot G_{I_i}$.
4. Return(A).

14.89 Example (*simultaneous multiple exponentiation*) In this example, $g_0^{30} g_1^{10} g_2^{24}$ is computed using Algorithm 14.88. Let $e_0 = 30 = (11110)_2$, $e_1 = 10 = (01010)_2$, and $e_2 = 24 = (11000)_2$. The 3×5 array EA is:

1	1	1	1	0
0	1	0	1	0
1	1	0	0	0

The next table displays precomputed values from step 1 of Algorithm 14.88.

i	0	1	2	3	4	5	6	7
G_i	1	g_0	g_1	$g_0 g_1$	g_2	$g_0 g_2$	$g_1 g_2$	$g_0 g_1 g_2$

Finally, the value of A at the end of each iteration of step 3 is shown in the following table. Here, $I_1 = 5$, $I_2 = 7$, $I_3 = 1$, $I_4 = 3$, and $I_5 = 0$.

i	1	2	3	4	5
A	$g_0 g_2$	$g_0^3 g_1 g_2^3$	$g_0^7 g_1^2 g_2^6$	$g_0^{15} g_1^5 g_2^{12}$	$g_0^{30} g_1^{10} g_2^{24}$

□

14.90 Note (*computational efficiency of Algorithm 14.88*)

(i) Algorithm 14.88 computes $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ (where each e_i is represented by t bits) by performing $t - 1$ squarings and at most $(2^k - 2) + t - 1$ multiplications. The multiplication is trivial for any column consisting of all 0's.

(ii) Not all of the G_i, $0 \le i \le 2^k - 1$, need to be precomputed, but only for those i whose binary representation is a column of EA.

14.91 Note (*ElGamal signature verification*) The signature verification equation for the ElGamal signature scheme (Algorithm 11.64) is $\alpha^{h(m)}(\alpha^{-a})^r \equiv r^s \pmod p$ where p is a large prime, α a generator of $\mathbb{Z}_p^*$, α^a is the public key, and (r, s) is a signature for message m. It would appear that three exponentiations and one multiplication are required to verify the equation. If $t = \lceil \lg p \rceil$ and Algorithm 11.64 is applied, the number of squarings is $3(t - 1)$ and the number of multiplications is, on average, $3t/2$. Hence, one would expect to perform about $(9t - 4)/2$ multiplications and squarings modulo p. Algorithm 14.88 can reduce the number of computations substantially if the verification equation is rewritten as $\alpha^{h(m)}(\alpha^{-a})^r r^{-s} \equiv 1 \pmod p$. Taking $g_0 = \alpha$, $g_1 = \alpha^{-a}$, $g_2 = r$, and $e_0 = h(m) \bmod (p-1)$, $e_1 = r \bmod (p-1)$, $e_2 = -s \bmod (p-1)$ in Algorithm 14.88, the expected number of multiplications and squarings is $(t-1) + (6 + (7t/8)) = (15t + 40)/8$. (For random exponents, one would expect that, on average, $\frac{7}{8}$ of the columns of EA will be non-zero and necessitate a non-trivial multiplication.) This is only about 25% more costly than a single exponentiation computed by Algorithm 14.79.

(iv) Additive notation

Algorithms 14.76 and 14.79 have been described in the setting of a multiplicative group. Algorithm 14.92 uses the methodology of Algorithm 14.79 to perform efficient multiplication in an additive group G. (For example, the group formed by the points on an elliptic curve over a finite field uses additive notation.) Multiplication in an additive group corresponds to exponentiation in a multiplicative group.

14.92 Algorithm Left-to-right binary multiplication in an additive group

---

INPUT: $g \in G$, where G is an additive group, and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
OUTPUT: $e \cdot g$.

1. $A \leftarrow 0$.
2. For i from t down to 0 do the following:
   - 2.1 $A \leftarrow A + A$.
   - 2.2 If $e_i = 1$ then $A \leftarrow A + g$.
3. Return(A).

14.93 Note (*the additive group* $\mathbb{Z}_m$)

(i) If G is the additive group $\mathbb{Z}_m$, then Algorithm 14.92 provides a method for doing modular multiplication. For example, if $a, b \in \mathbb{Z}_m$, then $a \cdot b \bmod m$ can be computed using Algorithm 14.92 by taking $g = a$ and $e = b$, provided b is written in binary.

(ii) If $a, b \in \mathbb{Z}_m$, then $a < m$ and $b < m$. The accumulator A in Algorithm 14.92 never contains an integer as large as $2m$; hence, modular reduction of the value in the accumulator can be performed by a simple subtraction when $A \geq m$; thus no divisions are required.

(iii) Algorithms 14.82 and 14.83 can also be used for modular multiplication. In the case of the additive group $\mathbb{Z}_m$, the time required to do modular multiplication can be improved at the expense of precomputing a table of residues modulo m. For a left-to-right k-ary exponentiation scheme, the table will contain $2^k - 1$ residues modulo m.

(v) Montgomery exponentiation

The introductory remarks to §14.3.2 outline an application of the Montgomery reduction method for exponentiation. Algorithm 14.94 below combines Algorithm 14.79 and Algorithm 14.36 to give a *Montgomery exponentiation algorithm* for computing $x^e \bmod m$. Note the definition of m' requires that $\gcd(m, R) = 1$. For integers u and v where $0 \leq u, v < m$, define Mont(u, v) to be $uvR^{-1} \bmod m$ as computed by Algorithm 14.36.

14.94 Algorithm Montgomery exponentiation

INPUT: $m = (m_{l-1} \cdots m_0)_b$, $R = b^l$, $m' = -m^{-1} \bmod b$, $e = (e_t \cdots e_0)_2$ with $e_t = 1$, and an integer x, $1 \le x < m$.
OUTPUT: $x^e \bmod m$.

1. $\widetilde{x} \leftarrow$ Mont$(x, R^2 \bmod m)$, $A \leftarrow R \bmod m$. ($R \bmod m$ and $R^2 \bmod m$ may be provided as inputs.)
2. For i from t down to 0 do the following:
   2.1 $A \leftarrow$ Mont(A, A).
   2.2 If $e_i = 1$ then $A \leftarrow$ Mont$(A, \widetilde{x})$.
3. $A \leftarrow$ Mont$(A, 1)$.
4. Return(A).

14.95 Example (*Montgomery exponentiation*) Let x, m, and R be integers suitable as inputs to Algorithm 14.94. Let $e = 11 = (1011)_2$; here, $t = 3$. The following table displays the values of $A \bmod m$ at the end of each iteration of step 2, and after step 3. □

i	3	2	1	0	Step 3
$A \bmod m$	$\widetilde{x}$	$\widetilde{x}^2 R^{-1}$	$\widetilde{x}^5 R^{-4}$	$\widetilde{x}^{11} R^{-10}$	Mont$(A, 1) = \widetilde{x}^{11} R^{-11} = x^{11}$

14.96 Note (*computational efficiency of Montgomery exponentiation*)

(i) Table 14.17 displays the average number of single-precision multiplications required for each step of Algorithm 14.94. The expected number of single-precision multiplications to compute $x^e \bmod m$ by Algorithm 14.94 is $3l(l+1)(t+1)$.

(ii) Each iteration of step 2 in Algorithm 14.94 applies Algorithm 14.36 at a cost of $2l(l+1)$ single-precision multiplications but no single-precision divisions. A similar algorithm for modular exponentiation based on classical modular multiplication (Algorithm 14.28) would similarly use $2l(l+1)$ single-precision multiplications per iteration but also l single-precision divisions.

(iii) Any of the other exponentiation algorithms discussed in §14.6.1 can be combined with Montgomery reduction to give other Montgomery exponentiation algorithms.

Step	1	2	3
Number of Montgomery multiplications	1	$\frac{3}{2}t$	1
Number of single-precision multiplications	$2l(l+1)$	$3tl(l+1)$	$l(l+1)$

Table 14.17: *Average number of single-precision multiplications per step of Algorithm 14.94.*

14.6.2 Fixed-exponent exponentiation algorithms

There are numerous situations in which a number of exponentiations by a fixed exponent must be performed. Examples include RSA encryption and decryption, and ElGamal decryption. This subsection describes selected algorithms which improve the repeated square-and-multiply algorithms of §14.6.1 by reducing the number of multiplications.

(i) Addition chains

The purpose of an addition chain is to minimize the number of multiplications required for an exponentiation.

14.97 Definition An *addition chain* V of length s for a positive integer e is a sequence $u_0, u_1, \ldots, u_s$ of positive integers, and an associated sequence $w_1, \ldots, w_s$ of pairs $w_i = (i_1, i_2)$, $0 \leq i_1, i_2 < i$, having the following properties:

(i) $u_0 = 1$ and $u_s = e$; and
(ii) for each u_i, $1 \leq i \leq s$, $u_i = u_{i_1} + u_{i_2}$.

14.98 Algorithm Addition chain exponentiation

INPUT: a group element g, an addition chain $V = (u_0, u_1, \ldots, u_s)$ of length s for a positive integer e, and the associated sequence $w_1, \ldots, w_s$, where $w_i = (i_1, i_2)$.
OUTPUT: g^e.

1. $g_0 \leftarrow g$.
2. For i from 1 to s do: $g_i \leftarrow g_{i_1} \cdot g_{i_2}$.
3. Return(g_s).

14.99 Example (*addition chain exponentiation*) An addition chain of length 5 for $e = 15$ is $u_0 = 1$, $u_1 = 2$, $u_2 = 3$, $u_3 = 6$, $u_4 = 12$, $u_5 = 15$. The following table displays the values of w_i and g_i during each iteration of Algorithm 14.98 for computing g^{15}. □

i	0	1	2	3	4	5
w_i	–	$(0,0)$	$(0,1)$	$(2,2)$	$(3,3)$	$(2,4)$
g_i	g	g^2	g^3	g^6	g^{12}	g^{15}

14.100 Remark (*addition chains and binary representations*) Given the binary representation of an exponent e, it is a relatively simple task to construct an addition chain directly from this representation. Chains constructed in this way generally do not provide the shortest addition chain possible for the given exponent. The methods for exponentiation described in §14.6.1 could be phrased in terms of addition chains, but this is typically not done.

14.101 Note (*computational efficiency of addition chain exponentiation*) Given an addition chain of length s for the positive integer e, Algorithm 14.98 computes g^e for any $g \in G$, $g \neq 1$, using exactly s multiplications.

14.102 Fact If l is the length of a shortest addition chain for a positive integer e, then $l \geq (\lg e + \lg \mathrm{wt}(e) - 2.13)$, where $\mathrm{wt}(e)$ is the number of 1's in the binary representation of e. An upper bound of $(\lfloor \lg e \rfloor + \mathrm{wt}(e) - 1)$ is obtained by constructing an addition chain for e from its binary representation. Determining a shortest addition chain for e is known to be an **NP**-hard problem.

(ii) Vector-addition chains

Algorithms 14.88 and 14.104 are useful for computing $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $g_0, g_1, \ldots, g_{k-1}$ are arbitrary elements in a group G and $e_0, e_1, \ldots, e_{k-1}$ are fixed positive integers. These algorithms can also be used to advantage when the exponents are not necessarily fixed values (see Note 14.91). Algorithm 14.104 makes use of vector-addition chains.

14.103 Definition Let s and k be positive integers and let v_i denote a k-dimensional vector of non-negative integers. An ordered set $V = \{v_i : -k+1 \le i \le s\}$ is called a *vector-addition chain* of length s and dimension k if V satisfies the following:

(i) Each v_i, $-k+1 \le i \le 0$, has a 0 in each coordinate position, except for coordinate position $i+k-1$, which is a 1. (Coordinate positions are labeled 0 through $k-1$.)
(ii) For each v_i, $1 \le i \le s$, there exists an associated pair of integers $w_i = (i_1, i_2)$ such that $-k+1 \le i_1, i_2 < i$ and $v_i = v_{i_1} + v_{i_2}$ ($i_1 = i_2$ is allowed).

Example 14.105 illustrates a sample vector-addition chain. Let $V = \{v_i : -k+1 \le i \le s\}$ be a vector-addition chain of length s and dimension k with associated sequence $w_1, \ldots, w_s$. Algorithm 14.104 computes $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $v_s = (e_0, e_1, \ldots, e_{k-1})$.

14.104 Algorithm Vector-addition chain exponentiation

INPUT: group elements $g_0, g_1, \ldots, g_{k-1}$ and a vector-addition chain V of length s and dimension k with associated sequence $w_1, \ldots, w_s$, where $w_i = (i_1, i_2)$.
OUTPUT: $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $v_s = (e_0, e_1, \ldots, e_{k-1})$.

1. For i from $(-k+1)$ to 0 do: $a_i \leftarrow g_{i+k-1}$.
2. For i from 1 to s do: $a_i \leftarrow a_{i_1} \cdot a_{i_2}$.
3. Return(a_s).

14.105 Example (*vector-addition chain exponentiation*) A vector-addition chain V of length $s = 9$ and dimension $k = 3$ is displayed in the following table.

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
1	0	0	1	2	2	3	5	6	12	15	30
0	1	0	0	0	1	1	2	2	4	5	10
0	0	1	1	2	2	2	4	5	10	12	24

The following table displays the values of w_i and a_i during each iteration of step 2 in Algorithm 14.104 for computing $g_0^{30} g_1^{10} g_2^{24}$. Nine multiplications are required. □

i	1	2	3	4	5	6	7	8	9
w_i	$(-2,0)$	$(1,1)$	$(-1,2)$	$(-2,3)$	$(3,4)$	$(1,5)$	$(6,6)$	$(4,7)$	$(8,8)$
a_i	$g_0 g_2$	$g_0^2 g_2^2$	$g_0^2 g_1 g_2^2$	$g_0^3 g_1 g_2^2$	$g_0^5 g_1^2 g_2^4$	$g_0^6 g_1^2 g_2^5$	$g_0^{12} g_1^4 g_2^{10}$	$g_0^{15} g_1^5 g_2^{12}$	$g_0^{30} g_1^{10} g_2^{24}$

14.106 Note (*computational efficiency of vector-addition chain exponentiation*)

(i) (*multiplications*) Algorithm 14.104 performs exactly s multiplications for a vector-addition chain of length s. To compute $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ using Algorithm 14.104, one would like to find a vector-addition chain of length s and dimension k with $v_s = (e_0, e_1, \ldots, e_{k-1})$, where s is as small as possible (see Fact 14.107).

(ii) (*storage*) Algorithm 14.104 requires intermediate storage for the elements a_i, $-k + 1 \le i < t$, at the t^{th} iteration of step 2. If not all of these are required for succeeding iterations, then they need not be stored. Algorithm 14.88 provides a special case of Algorithm 14.104 where the intermediate storage is no larger than $2^k - 1$ vectors of dimension k.

14.107 Fact The minimum value of s in Note 14.106(i) satisfies the following bound, where $M = \max\{e_i : 0 \le i \le k-1\}$ and c is a constant:

$$s \le k - 1 + \lg M + ck \cdot \lg M / \lg\lg(M+2).$$

14.108 Example (*vector-addition chains from binary representations*) The vector-addition chain implicit in Algorithm 14.88 is not necessarily of minimum length. The vector-addition chain associated with Example 14.89 is displayed in Table 14.18. This chain is longer than the one used in Example 14.105. The advantage of Algorithm 14.88 is that the vector-addition chain does not have to be explicitly provided to the algorithm. In view of this, Algorithm 14.88 can be applied more generally to situations where the exponents are not necessarily fixed. □

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	0	0	1	1	1	2	3	6	7	14	15	30
0	1	0	1	1	0	0	1	2	2	4	5	10
0	0	1	0	1	1	2	3	6	6	12	12	24

Table 14.18: *Binary vector-addition chain exponentiation (see Example 14.108).*

14.6.3 Fixed-base exponentiation algorithms

Three methods are presented for exponentiation when the base g is fixed and the exponent e varies. With a fixed base, precomputation can be done once and used for many exponentiations. For example, Diffie-Hellman key agreement (Protocol 12.47) requires the computation of α^x, where α is a fixed element in $\mathbb{Z}_p^*$.

For each of the algorithms described in this section, $\{b_0, b_1, \ldots, b_t\}$ is a set of integers for some $t \ge 0$, such that any exponent $e \ge 1$ (suitably bounded) can be written as $e = \sum_{i=0}^{t} e_i b_i$, where $0 \le e_i < h$ for some fixed positive integer h. For example, if e is any $(t+1)$-digit base b integer with $b \ge 2$, then $b_i = b^i$ and $h = b$ are possible choices.

Algorithms 14.109 and 14.113 are two fixed-base exponentiation methods. Both require precomputation of the exponentials $g^{b_0}, g^{b_1}, \ldots, g^{b_t}$, e.g., using one of the algorithms from §14.6.1. The precomputation needed for Algorithm 14.117 is more involved and is explicitly described in Algorithm 14.116.

(i) Fixed-base windowing method

Algorithm 14.109 takes as input the precomputed exponentials $g_i = g^{b_i}$, $0 \le i \le t$, and positive integers h and $e = \sum_{i=0}^{t} e_i b_i$ where $0 \le e_i < h$, $0 \le i \le t$. The basis for the algorithm is the observation that $g^e = \prod_{i=0}^{t} g_i^{e_i} = \prod_{j=1}^{h-1} (\prod_{e_i = j} g_i)^j$.

14.109 Algorithm Fixed-base windowing method for exponentiation

INPUT: $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$, $e = \sum_{i=0}^{t} e_i b_i$, and h.
OUTPUT: g^e.

1. $A \leftarrow 1$, $B \leftarrow 1$.
2. For j from $(h-1)$ down to 1 do the following:
   2.1 For each i for which $e_i = j$ do: $B \leftarrow B \cdot g^{b_i}$.
   2.2 $A \leftarrow A \cdot B$.
3. Return(A).

14.110 Example (*fixed-base windowing exponentiation*) Precompute the group elements g^1, g^4, g^{16}, g^{64}, g^{256}. To compute g^e for $e = 862 = (31132)_4$, take $t = 4$, $h = 4$, and $b_i = 4^i$ for $0 \le i \le 4$, in Algorithm 14.109. The following table displays the values of A and B at the end of each iteration of step 2. □

j	$-$	3	2	1
B	1	$g^4 g^{256} = g^{260}$	$g^{260} g = g^{261}$	$g^{261} g^{16} g^{64} = g^{341}$
A	1	g^{260}	$g^{260} g^{261} = g^{521}$	$g^{521} g^{341} = g^{862}$

14.111 Note (*computational efficiency of fixed-base windowing exponentiation*)

(i) (*number of multiplications*) Suppose $t + h \ge 2$. Only multiplications where both operands are distinct from 1 are counted. Step 2.2 is executed $h-1$ times, but at least one of these multiplications involves an operand with value 1 (A is initialized to 1). Since B is also initially 1, at most t multiplications are done in step 2.1. Thus, Algorithm 14.109 computes g^e with at most $t + h - 2$ multiplications (cf. Note 14.112).

(ii) (*storage*) Storage is required for the $t + 1$ group elements g_i, $0 \le i \le t$.

14.112 Note (*a particular case*) The most obvious application of Algorithm 14.109 is the case where the exponent e is represented in radix b. If $e = \sum_{i=0}^{t} e_i b^i$, then $g_i = g^{b^i}$, $0 \le i \le t$, are precomputed. If e is randomly selected from $\{0, 1, \dots, m-1\}$, then $t+1 \le \lceil \log_b m \rceil$ and, on average, $\frac{1}{b}$ of the base b digits in e will be 0. In this case, the expected number of multiplications is $\frac{b-1}{b} \lceil \log_b m \rceil + b - 3$. If m is a 512-bit integer and $b = 32$, then 128.8 multiplications are needed on average, 132 in the worst case; 103 values must be stored.

(ii) Fixed-base Euclidean method

Let $\{x_0, x_1, \dots, x_t\}$ be a set of integers with $t \ge 2$. Define M to be an integer in the interval $[0, t]$ such that $x_M \ge x_i$ for all $0 \le i \le t$. Define N to be an integer in the interval $[0, t]$, $N \ne M$, such that $e_N \ge e_i$ for all $0 \le i \le t$, $i \ne M$.

14.113 Algorithm Fixed-base Euclidean method for exponentiation

INPUT: $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$ and $e = \sum_{i=0}^{t} e_i b_i$.
OUTPUT: g^e.

1. For i from 0 to t do the following: $g_i \leftarrow g^{b_i}$, $x_i \leftarrow e_i$.
2. Determine the indices M and N for $\{x_0, x_1, \dots, x_t\}$.
3. While $x_N \ne 0$ do the following:
   3.1 $q \leftarrow \lfloor x_M / x_N \rfloor$, $g_N \leftarrow (g_M)^q \cdot g_N$, $x_M \leftarrow x_M \bmod x_N$.

3.2 Determine the indices M and N for $\{x_0, x_1, \ldots, x_t\}$.

4. Return($g_M^{x_M}$).

14.114 Example (*fixed-base Euclidean method*) This example repeats the computation of g^e, $e = 862$ done in Example 14.110, but now uses Algorithm 14.113. Take $b_0 = 1$, $b_1 = 16$, $b_2 = 256$. Then $e = (3, 5, 14)_{16}$. Precompute g^1, g^{16}, g^{256}. Table 14.19 illustrates the steps performed by Algorithm 14.113. Notice that for this example, Algorithm 14.113 does 8 multiplications, whereas Algorithm 14.109 needs only 6 to do the same computation. Storage requirements for Algorithm 14.113 are, however, smaller. The vector-addition chain (Definition 14.103) corresponding to this example is displayed in the following table. □

x_0	x_1	x_2	M	N	q	g_0	g_1	g_2
14	5	3	0	1	2	g	g^{18}	g^{256}
4	5	3	1	0	1	g^{19}	g^{18}	g^{256}
4	1	3	0	2	1	g^{19}	g^{18}	g^{275}
1	1	3	2	1	3	g^{19}	g^{843}	g^{275}
1	1	0	0	1	1	g^{19}	g^{862}	g^{275}
0	1	0	1	0	–	g^{19}	g^{862}	g^{275}

Table 14.19: *Fixed-base Euclidean method to compute g^{862} (see Example 14.114).*

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	0	0	2	2	3	3	6	9	11	14
0	1	0	0	1	1	1	2	3	4	5
0	0	1	0	0	0	1	2	3	3	3

14.115 Note (*fixed-base Euclidean vs. fixed-base windowing methods*)

(i) In most cases, the quotient q computed in step 3.1 of Algorithm 14.113 is 1. For a given base b, the computational requirements of this algorithm are not significantly greater than those of Algorithm 14.109.

(ii) Since the division algorithm is logarithmic in the size of the inputs, Algorithm 14.113 can take advantage of a larger value of h than Algorithm 14.109. This results in less storage for precomputed values.

(iii) Fixed-base comb method

Algorithm 14.117 computes g^e where $e = (e_t e_{t-1} \cdots e_1 e_0)_2$, $t \geq 1$. Select an integer h, $1 \leq h \leq t+1$ and compute $a = \lceil (t+1)/h \rceil$. Select an integer v, $1 \leq v \leq a$, and compute $b = \lceil a/v \rceil$. Clearly, $ah \geq t+1$. Let $X = R_{h-1}||R_{h-2}|| \cdots ||R_0$ be a bitstring formed from e by padding (if necessary) e on the left with 0's, so that X has bitlength ah and each R_i, $0 \leq i \leq h-1$, is a bitstring of length a. Form an $h \times a$ array EA (called the *exponent array*) where row i of EA is the bitstring R_i, $0 \leq i \leq h-1$. Algorithm 14.116 is the precomputation required for Algorithm 14.117.

14.116 Algorithm Precomputation for Algorithm 14.117

INPUT: group element g and parameters h, v, a, and b (defined above).
OUTPUT: $\{G[j][i] : 1 \le i < 2^h, 0 \le j < v\}$.

1. For i from 0 to $(h-1)$ do: $g_i \leftarrow g^{2^{ia}}$.
2. For i from 1 to $(2^h - 1)$ (where $i = (i_{h-1} \cdots i_0)_2$), do the following:
   2.1 $G[0][i] \leftarrow \prod_{j=0}^{h-1} g_j^{i_j}$.
   2.2 For j from 1 to $(v-1)$ do: $G[j][i] \leftarrow (G[0][i])^{2^{jb}}$.
3. Return($\{G[j][i] : 1 \le i < 2^h, 0 \le j < v\}$).

Let $I_{j,k}$, $0 \le k < b$, $0 \le j < v$, be the integer whose binary representation is column $(jb+k)$ of EA, where column 0 is on the right and the least significant bits of a column are at the top.

14.117 Algorithm Fixed-base comb method for exponentiation

INPUT: g, e and $\{G[j][i] : 1 \le i < 2^h, 0 \le j < v\}$ (precomputed in Algorithm 14.116).
OUTPUT: g^e.

1. $A \leftarrow 1$.
2. For k from $(b-1)$ down to 0 do the following:
   2.1 $A \leftarrow A \cdot A$.
   2.2 For j from $(v-1)$ down to 0 do: $A \leftarrow G[j][I_{j,k}] \cdot A$.
3. Return(A).

14.118 Example (*fixed-base comb method for exponentiation*) Let $t = 9$ and $h = 3$; then $a = \lceil 10/3 \rceil = 4$. Let $v = 2$; then $b = \lceil a/v \rceil = 2$. Suppose the exponent input to Algorithm 14.117 is $e = (e_9 e_8 \cdots e_1 e_0)_2$. Form the bitstring $X = x_{11} x_{10} \cdots x_1 x_0$ where $x_i = e_i$, $0 \le i \le 9$, and $x_{11} = x_{10} = 0$. The following table displays the exponent array EA.

$I_{1,1}$	$I_{1,0}$	$I_{0,1}$	$I_{0,0}$
x_3	x_2	x_1	x_0
x_7	x_6	x_5	x_4
x_{11}	x_{10}	x_9	x_8

The precomputed values from Algorithm 14.116 are displayed below. Recall that $g_i = g^{2^{ia}}$, $0 \le i < 3$.

i	1	2	3	4	5	6	7
$G[0][i]$	g_0	g_1	$g_1 g_0$	g_2	$g_2 g_0$	$g_2 g_1$	$g_2 g_1 g_0$
$G[1][i]$	g_0^4	g_1^4	$g_1^4 g_0^4$	g_2^4	$g_2^4 g_0^4$	$g_2^4 g_1^4$	$g_2^4 g_1^4 g_0^4$

Finally, the following table displays the steps in Algorithm 14.117 for EA.

		$A = g_0^{l_0} g_1^{l_1} g_2^{l_2}$		
k	j	l_0	l_1	l_2
1	–	0	0	0
1	1	$4x_3$	$4x_7$	$4x_{11}$
1	0	$4x_3 + x_1$	$4x_7 + x_5$	$4x_{11} + x_9$
0	–	$8x_3 + 2x_1$	$8x_7 + 2x_5$	$8x_{11} + 2x_9$
0	1	$8x_3 + 2x_1 + 4x_2$	$8x_7 + 2x_5 + 4x_6$	$8x_{11} + 2x_9 + 4x_{10}$
0	0	$8x_3 + 2x_1 + 4x_2 + x_0$	$8x_7 + 2x_5 + 4x_6 + x_4$	$8x_{11} + 2x_9 + 4x_{10} + x_8$

The last row of the table corresponds to $g^{\sum_{i=0}^{11} x_i 2^i} = g^e$. □

14.119 Note (*computational efficiency of fixed-base comb method*)

(i) (*number of multiplications*) Algorithm 14.117 requires at most one multiplication for each column of EA. The right-most column of EA requires a multiplication with the initial value 1 of the accumulator A. The algorithm also requires a squaring of the accumulator A for each k, $0 \leq k < b$, except for $k = b - 1$ when A has value 1. Discounting multiplications by 1, the total number of non-trivial multiplications (including squarings) is, at most, $a + b - 2$.

(ii) (*storage*) Algorithm 14.117 requires storage for the $v(2^h - 1)$ precomputed group elements (Algorithm 14.116). If squaring is a relatively simple operation compared to multiplication in the group, then some space-saving can be achieved by storing only $2^h - 1$ group elements (i.e., only those elements computed in step 2.1 of Algorithm 14.116).

(iii) (*trade-offs*) Since h and v are independent of the number of bits in the exponent, selection of these parameters can be made based on the amount of storage available vs. the amount of time (determined by multiplication) to do the computation.

14.7 Exponent recoding

Another approach to reducing the number of multiplications in the basic repeated square-and-multiply algorithms (§14.6.1) is to replace the binary representation of the exponent e with a representation which has fewer non-zero terms. Since the binary representation is unique (Fact 14.1), finding a representation with fewer non-zero components necessitates the use of digits besides 0 and 1. Transforming an exponent from one representation to another is called *exponent recoding*. Many techniques for exponent recoding have been proposed in the literature. This section describes two possibilities: signed-digit representation (§14.7.1) and string-replacement representation (§14.7.2).

14.7.1 Signed-digit representation

14.120 Definition If $e = \sum_{i=0}^{t} d_i 2^i$ where $d_i \in \{0, 1, -1\}$, $0 \leq i \leq t$, then $(d_t \cdots d_1 d_0)_{SD}$ is called a *signed-digit representation with radix* 2 for the integer e.

Unlike the binary representation, the signed-digit representation of an integer is not unique. The binary representation is an example of a signed-digit representation. Let e be a positive integer whose binary representation is $(e_{t+1} e_t e_{t-1} \cdots e_1 e_0)_2$, with $e_{t+1} = e_t = 0$. Algorithm 14.121 constructs a signed-digit representation for e having at most $t + 1$ digits and the smallest possible number of non-zero terms.

14.121 Algorithm Signed-digit exponent recoding

INPUT: a positive integer $e = (e_{t+1}e_te_{t-1}\cdots e_1e_0)_2$ with $e_{t+1} = e_t = 0$.
OUTPUT: a signed-digit representation $(d_t \cdots d_1d_0)_{SD}$ for e. (See Definition 14.120.)

1. $c_0 \leftarrow 0$.
2. For i from 0 to t do the following:
   2.1 $c_{i+1} \leftarrow \lfloor (e_i + e_{i+1} + c_i)/2 \rfloor$, $d_i \leftarrow e_i + c_i - 2c_{i+1}$.
3. Return($(d_t \cdots d_1d_0)_{SD}$).

14.122 Example (*signed-digit exponent recoding*) Table 14.20 lists all possible inputs to the i^{th} iteration of step 2, and the corresponding outputs. If $e = (1101110111)_2$, then Algorithm 14.121 produces the signed-digit representation $e = (100\overline{1}000\overline{1}00\overline{1})_{SD}$ where $\overline{1} = -1$. Note that $e = 2^9 + 2^8 + 2^6 + 2^5 + 2^4 + 2^2 + 2 + 1 = 2^{10} - 2^7 - 2^3 - 1$. □

inputs	e_i	0	0	0	0	1	1	1	1
	c_i	0	0	1	1	0	0	1	1
	e_{i+1}	0	1	0	1	0	1	0	1
outputs	c_{i+1}	0	0	0	1	0	1	1	1
	d_i	0	0	1	-1	1	-1	0	0

Table 14.20: *Signed-digit exponent recoding (see Example 14.122).*

14.123 Definition A signed-digit representation of an integer e is said to be *sparse* if no two non-zero entries are adjacent in the representation.

14.124 Fact (*sparse signed-digit representation*)

(i) Every integer e has a unique sparse signed-digit representation.
(ii) A sparse signed-digit representation for e has the smallest number of non-zero entries among all signed-digit representations for e.
(iii) The signed-digit representation produced by Algorithm 14.121 is sparse.

14.125 Note (*computational efficiency of signed-digit exponent recoding*)

(i) Signed-digit exponent recoding as per Algorithm 14.121 is very efficient, and can be done by table look-up (using Table 14.20).
(ii) When e is given in a signed-digit representation, computing g^e requires both g and g^{-1}. If g is a fixed base, then g^{-1} can be precomputed. For a variable base g, unless g^{-1} can be computed very quickly, recoding an exponent to signed-digit representation may not be worthwhile.

14.7.2 String-replacement representation

14.126 Definition Let $k \geq 1$ be a positive integer. A non-negative integer e is said to have a *k-ary string-replacement representation* $(f_{t-1}f_{t-2}\cdots f_1f_0)_{SR(k)}$, denoted $SR(k)$, if $e = \sum_{i=0}^{t-1} f_i 2^i$ and $f_i \in \{2^j - 1 : 0 \leq j \leq k\}$ for $0 \leq i \leq t-1$.

14.127 Example (*non-uniqueness of string-replacement representations*) A string-replacement representation for a non-negative integer is generally not unique. The binary representation is a 1-ary string-replacement representation. If $k = 3$ and $e = 987 = (1111011011)_2$, then some other string-replacements of e are $(303003003)_{SR(3)}$, $(1007003003)_{SR(3)}$, and $(71003003)_{SR(3)}$. □

14.128 Algorithm k-ary string-replacement representation

INPUT: $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$ and positive integer $k \geq 2$.
OUTPUT: $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.

1. For i from k down to 2 do the following: starting with the most significant digit of $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$, replace each consecutive string of i ones with a string of length i consisting of $i - 1$ zeros in the high-order string positions and the integer $2^i - 1$ in the low-order position.
2. Return($(f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$).

14.129 Example (*k-ary string-replacement*) Suppose $e = (110111110011101)_2$ and $k = 3$. The $SR(3)$ representations of e at the end of each of the two iterations of Algorithm 14.128 are $(110007110000701)_{SR(3)}$ and $(030007030000701)_{SR(3)}$. □

14.130 Algorithm Exponentiation using an $SR(k)$ representation

INPUT: an integer $k \geq 2$, an element $g \in G$, and $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.
OUTPUT: g^e.

1. *Precomputation*. Set $g_1 \leftarrow g$. For i from 2 to k do: $g_{2^i-1} \leftarrow (g_{2^{i-1}-1})^2 \cdot g$.
2. $A \leftarrow 1$.
3. For i from $(t-1)$ down to 0 do the following:
   3.1 $A \leftarrow A \cdot A$.
   3.2 If $f_i \neq 0$ then $A \leftarrow A \cdot g_{f_i}$.
4. Return(A).

14.131 Example (*$SR(k)$ vs. left-to-right binary exponentiation*) Let $e = 987 = (1111011011)_2$ and consider the 3-ary string-replacement representation $(0071003003)_{SR(3)}$. Computing g^e using Algorithm 14.79 requires 9 squarings and 7 multiplications. Algorithm 14.130 requires 2 squarings and 2 multiplications for computing g^3 and g^7, and then 7 squarings and 3 multiplications for the main part of the algorithm. In total, the $SR(3)$ for e computes g^e with 9 squarings and 5 multiplications. □

14.132 Note (*computational efficiency of Algorithm 14.130*) The precomputation requires $k - 1$ squarings and $k - 1$ multiplications. Algorithm 14.128 is not guaranteed to produce an $SR(k)$ representation with a minimum number of non-zero entries, but in practice it seems to give representations which are close to minimal. Heuristic arguments indicate that a randomly selected t-bit exponent will be encoded with a suitably chosen value of k to an $SR(k)$ representation having about $t/4$ non-zero entries; hence, one expects to perform $t-1$ squarings in step 3, and about $t/4$ multiplications.

14.8 Notes and further references

§14.1

This chapter deals almost exclusively with methods to perform operations in the integers and the integers modulo some positive integer. When p is a prime number, $\mathbb{Z}_p$ is called a finite field (Fact 2.184). There are other finite fields which have significance in cryptography. Of particular importance are those of characteristic two, $\mathbb{F}_{2^m}$. Perhaps the most useful property of these structures is that squaring is a *linear operator* (i.e., if $\alpha, \beta \in \mathbb{F}_{2^m}$, then $(\alpha+\beta)^2 = \alpha^2+\beta^2$). This property leads to efficient methods for exponentiation and for inversion. Characteristic two finite fields have been used extensively in connection with error-correcting codes; for example, see Berlekamp [118] and Lin and Costello [769]. For error-correcting codes, m is typically quite small (e.g., $1 \leq m \leq 16$); for cryptographic applications, m is usually much larger (e.g., $m \geq 100$).

The majority of the algorithms presented in this chapter are best suited to software implementations. There is a vast literature on methods to perform modular multiplication and other operations in hardware. The basis for most hardware implementations for modular multiplication is efficient methods for integer addition. In particular, *carry-save adders* and *delayed-carry adders* are at the heart of the best methods to perform modular multiplication. The concept of a delayed-carry adder was proposed by Norris and Simmons [933] to produce a hardware modular multiplier which computes the product of two t-bit operands modulo a t-bit modulus in $2t$ clock cycles. Brickell [199] improved the idea to produce a modular multiplier requiring only $t+7$ clock cycles. Enhancements of Brickell's method were given by Walter [1230]. Koç [699] gives a comprehensive survey of hardware methods for modular multiplication.

§14.2

For a treatment of radix representations including *mixed-radix representations*, see Knuth [692]. Knuth describes efficient methods for performing radix conversions. Representing and manipulating negative numbers is an important topic; for an introduction, consult the book by Koren [706].

The techniques described in §14.2 are commonly referred to as the *classical algorithms* for multiple-precision addition, subtraction, multiplication, and division. These algorithms are the most useful for integers of the size used for cryptographic purposes. For much larger integers (on the order of thousands of decimal digits), more efficient methods exist. Although not of current practical interest, some of these may become more useful as security requirements force practitioners to increase parameter sizes. The Karatsuba-Ofman method, described next, is practical in some situations.

The classical algorithm for multiplication (Algorithm 14.12) takes $O(n^2)$ bit operations for multiplying two n-bit integers. A recursive algorithm due to Karatsuba and Ofman [661] reduces the complexity of multiplying two n-bit integers to $O(n^{1.58})$. This *divide-and-conquer* method is based on the following simple observation. Suppose that x and y are n-bit integers and $n = 2t$. Then $x = 2^t x_1 + x_0$ and $y = 2^t y_1 + y_0$, where x_1, y_1 are the t high-order bits of x and y, respectively, and x_0, y_0 are the t low-order bits. Furthermore, $x \cdot y = u_2 2^{2t} + u_1 2^t + u_0$ where $u_0 = x_0 \cdot y_0$, $u_2 = x_1 \cdot y_1$ and $u_1 = (x_0+x_1)\cdot(y_0+y_1) - u_0 - u_2$. It follows that $x \cdot y$ can be computed by performing three multiplications of t-bit integers (as opposed to one multiplication with $2t$-bit integers) along with two additions and two subtractions. For large values of t, the cost of the additions and subtractions is insignificant relative to the cost of the multiplications. With appropriate modifications, u_0, u_1 and

$(x_0 + x_1) \cdot (y_0 + y_1)$ can each be computed similarly. This procedure is continued on the intermediate values until the size of the integers reaches the word size of the computing device, and multiplication can be efficiently accomplished. Due to the recursive nature of the algorithm, a number of intermediate results must be stored which can add significant overhead, and detract from the algorithm's efficiency for relatively small integers. Combining the Karatsuba-Ofman method with classical multiplication may have some practical significance. For a more detailed treatment of the Karatsuba-Ofman algorithm, see Knuth [692], Koç [698], and Geddes, Czapor, and Labahn [445].

Another commonly used method for multiple-precision integer multiplication is the *discrete Fourier transform* (DFT). Although mathematically elegant and asymptotically better than the classical algorithm, it does not appear to be superior for the size of integers of practical importance to cryptography. Lipson [770] provides a well-motivated and easily readable treatment of this method.

The identity given in Note 14.18 was known to Karatsuba and Ofman [661].

§14.3

There is an extensive literature on methods for multiple-precision modular arithmetic. A detailed treatment of methods for performing modular multiplication can be found in Knuth [692]. Koç [698] and Bosselaers, Govaerts, and Vandewalle [176] provide comprehensive but brief descriptions of the classical method for modular multiplication.

Montgomery reduction (Algorithm 14.32) is due to Montgomery [893], and is one of the most widely used methods in practice for performing modular exponentiation (Algorithm 14.94). Dussé and Kaliski [361] discuss variants of Montgomery's method. Montgomery reduction is a generalization of a much older technique due to Hensel (see Shand and Vuillemin [1119] and Bosselaers, Govaerts, and Vandewalle [176]). Hensel's observation is the following. If m is an odd positive integer less than 2^k (k a positive integer) and T is some integer such that $2^k \le T < 2^{2k}$, then $R_0 = (T + q_0 m)/2$, where $q_0 = T \bmod 2$ is an integer and $R_0 \equiv T2^{-1} \bmod m$. More generally, $R_i = (R_{i-1} + q_i m)/2$, where $q_i = R_{i-1} \bmod 2$ is an integer and $R_i \equiv N2^{-i+1} \bmod m$. Since $T < 2^{2k}$, it follows that $R_{k-1} < 2m$.

Barrett reduction (Algorithm 14.42) is due to Barrett [75]. Bosselaers, Govaerts, and Vandewalle [176] provide a clear and concise description of the algorithm along with motivation and justification for various choices of parameters and steps, and compare three alternative methods: classical (§14.3.1), Montgomery reduction (§14.3.2), and Barrett reduction (§14.3.3). This comparison indicates that there is not a significant difference in performance between the three methods, provided the precomputation necessary for Montgomery and Barrett reduction is ignored. Montgomery exponentiation is shown to be somewhat better than the other two methods. The conclusions are based on both theoretical analysis and machine implementation for various sized moduli. Koç, Acar, and Kaliski [700] provide a more detailed comparison of various Montgomery multiplication algorithms; see also Naccache, M'Raïhi, and Raphaeli [915]. Naccache and M'silti [917] provide proofs for the correctness of Barrett reduction along with a possible optimization.

Mohan and Adiga [890] describe a special case of Algorithm 14.47 where $b = 2$.

Hong, Oh, and Yoon [561] proposed new methods for modular multiplication and modular squaring. They report improvements of 50% and 30%, respectively, on execution times over Montgomery's method for multiplication and squaring. Their approach to modular multiplication interleaves multiplication and modular reduction and uses precomputed tables such that one operand is always single-precision. Squaring uses recursion and precomputed tables and, unlike Montgomery's method, also integrates the multiplication and reduction steps.

§14.4

The binary gcd algorithm (Algorithm 14.54) is due to Stein [1170]. An analysis of the algorithm is given by Knuth [692]. Harris [542] proposed an algorithm for computing gcd's which combines the classical Euclidean algorithm (Algorithm 2.104) and binary operations; the method is called the *binary Euclidean algorithm*.

Lehmer's gcd algorithm (Algorithm 14.57), due to Lehmer [743], determines the gcd of two positive multiple-precision integers using mostly single-precision operations. This has the advantage of using the hardware divide in the machine and only periodically resorting to an algorithm such as Algorithm 14.20 for a multiple-precision divide. Knuth [692] gives a comprehensive description of the algorithm along with motivation of its correctness. Cohen [263] provides a similar discussion, but without motivation. Lehmer's gcd algorithm is readily adapted to the extended Euclidean algorithm (Algorithm 2.107).

According to Sorenson [1164], the binary gcd algorithm is the most efficient method for computing the greatest common divisor. Jebelean [633] suggests that Lehmer's gcd algorithm is more efficient. Sorenson [1164] also describes a k-ary version of the binary gcd algorithm, and proves a worst-case running time of $O(n^2/\lg n)$ bit operations for computing the gcd of two n-bit integers.

The binary extended gcd algorithm was first described by Knuth [692], who attributes it to Penk. Algorithm 14.61 is due to Bach and Shallit [70], who also give a comprehensive and clear analysis of several gcd and extended gcd algorithms. Norton [934] described a version of the binary extended gcd algorithm which is somewhat more complicated than Algorithm 14.61. Gordon [516] proposed a method for computing modular inverses, derived from the classical extended Euclidean algorithm (Algorithm 2.107) with multiple-precision division replaced by an approximation to the quotient by an appropriate power of 2; no analysis of the expected running time is given, but observed results on moduli of specific sizes are described.

The *Montgomery inverse* of $a \bmod m$ is defined to be $a^{-1}2^t \bmod m$ where t is the bitlength of m. Kaliski [653] extended ideas of Guyot [534] on the right-shift binary extended Euclidean algorithm, and presented an algorithm for computing the Montgomery inverse.

§14.5

Let m_i, $1 \le i \le t$, be a set of pairwise relatively prime positive integers which define a residue number system (RNS). If $n = \prod_{i=1}^{t} m_i$ then this RNS provides an effective method for computing the product of integers modulo n where the integers and the product are represented in the RNS. If n is a positive integer where the m_i do not necessarily divide n, then a method for performing arithmetic modulo n entirely within the RNS is not obvious. Couveignes [284] and Montgomery and Silverman [895] propose an interesting method for accomplishing this. Further research in the area is required to determine if this approach is competitive with or better than the modular multiplication methods described in §14.3.

Algorithm 14.71 is due to Garner [443]. A detailed discussion of this algorithm and variants of it are given by Knuth [692]; see also Cohen [263]. Algorithm 2.121 for applying the Chinese remainder theorem is due to Gauss; see Bach and Shallit [70]. Gauss's algorithm is a special case of the following result due to Nagasaka, Shiue, and Ho [918]. The solution to the system of linear congruences $x \equiv a_i \pmod{m_i}$, $1 \le i \le t$, for pairwise relative prime moduli m_i, is equivalent to the solution to the single linear congruence $(\sum_{i=1}^{t} b_i M_i)x \equiv \sum_{i=1}^{t} a_i b_i M_i \pmod{M}$ where $M = \prod_{i=1}^{t} m_i$, $M_i = M/m_i$

for $1 \le i \le t$, for any choice of integers b_i where gcd $(b_i, M_i) = 1$. Notice that if $\sum_{i=1}^{t} b_i M_i \equiv 1 \pmod{M}$, then $b_i \equiv M_i^{-1} \pmod{m_i}$, giving the special case discussed in Algorithm 2.121. Quisquater and Couvreur [1016] were the first to apply the Chinese remainder theorem to RSA decryption and signature generation.

§14.6

Knuth [692] and Bach and Shallit [70] describe the right-to-left binary exponentiation method (Algorithm 14.76). Cohen [263] provides a more comprehensive treatment of the right-to-left and left-to-right (Algorithm 14.79) binary methods along with their generalizations to the k-ary method. Koç [698] discusses these algorithms in the context of the RSA public-key cryptosystem. Algorithm 14.92 is the basis for Blakley's modular multiplication algorithm (see Blakley [149] and Koç [698]). The generalization of Blakley's method to process more than one bit per iteration (Note 14.93(iii)) is due to Quisquater and Couvreur [1016]. Quisquater and Couvreur describe an algorithm for modular exponentiation which makes use of the generalization and precomputed tables to accelerate multiplication in $\mathbb{Z}_m$.

For a comprehensive and detailed discussion of addition chains, see Knuth [692], where various methods for constructing addition chains (such as the *power tree* and *factor* methods) are described. Computing the shortest addition chain for a positive integer was shown to be an **NP**-hard problem by Downey, Leong, and Sethi [360]. The lower bound on the length of a shortest addition chain (Fact 14.102) was proven by Schönhage [1101].

An *addition sequence* for positive integers $a_1 < a_2 < \cdots < a_k$ is an addition chain for a_k in which $a_1, a_2, \ldots, a_{k-1}$ appear. Yao [1257] proved that there exists an addition sequence for $a_1 < a_2 < \cdots < a_k$ of length less than $\lg a_k + ck \cdot \lg a_k / \lg\lg(a_k + 2)$ for some constant c. Olivos [955] established a 1-1 correspondence between addition sequences of length l for $a_1 < a_2 < \cdots < a_k$ and vector-addition chains of length $l + k - 1$ where $v_{l+k-1} = (a_1, a_2, \ldots, a_k)$. These results are the basis for the inequality given in Fact 14.107. Bos and Coster [173] described a heuristic method for computing vector-addition chains. The special case of Algorithm 14.104 (Algorithm 14.88) is attributed by ElGamal [368] to Shamir.

The fixed-base windowing method (Algorithm 14.109) for exponentiation is due to Brickell et al. [204], who describe a number of variants of the basic algorithm. For b a positive integer, let S be a set of integers with the property that any integer can be expressed in base b using only coefficients from S. S is called a *basic digit set* for the base b. Brickell et al. show how basic digit sets can be used to reduce the amount of work in Algorithm 14.109 without large increases in storage requirements. De Rooij [316] proposed the fixed-base Euclidean method (Algorithm 14.113) for exponentiation; compares this algorithm to Algorithm 14.109; and provides a table of values for numbers of practical importance. The fixed-base comb method (Algorithm 14.117) for exponentiation is due to Lim and Lee [767]. For a given exponent size, they discuss various possibilities for the choice of parameters h and v, along with a comparison of their method to fixed-base windowing.

§14.7

The signed-digit exponent recoding algorithm (Algorithm 14.121) is due to Reitwiesner [1031]. A simpler description of the algorithm was given by Hwang [566]. Booth [171] described another algorithm for producing a signed-digit representation, but not necessarily one with the minimum possible non-zero components. It was originally given in terms of the additive group of integers where exponentiation is referred to as multiplication. In this case, $-g$ is easily computed from g. The additive abelian group formed from the points on an elliptic curve over a finite field is another example where signed-digit representation is very useful (see Morain and Olivos [904]). Zhang [1267] described a modified signed-digit representation which requires on average $t/3$ multiplications for a square-and-multiply algorithm for t-bit exponents. A slightly more general version of Algorithm 14.121, given by Jedwab and Mitchell [634], does not require as input a binary representation of the exponent e but simply a signed-digit representation. For binary inputs, the algorithms of Reitwiesner and Jedwab-Mitchell are the same. Fact 14.124 is due to Jedwab and Mitchell [634].

String-replacement representations were introduced by Gollmann, Han, and Mitchell [497], who describe Algorithms 14.128 and 14.130. They also provide an analysis of the expected number of non-zero entries in an $SR(k)$ representation for a randomly selected t-bit exponent (see Note 14.132), as well as a complexity analysis of Algorithm 14.130 for various values of t and k. Lam and Hui [735] proposed an alternate string-replacement algorithm. The idea is to precompute all odd powers $g, g^3, g^5, \ldots, g^{2^k-1}$ for some fixed positive integer k. Given a t-bit exponent e, start at the most significant bit, and look for the longest bitstring of bitlength at most k whose last digit is a 1 (i.e., this substring represents an odd positive integer between 1 and $2^k - 1$). Applying a left-to-right square-and-multiply exponentiation algorithm based on this scanning process results in an algorithm which requires, at most, $\lceil t/k \rceil$ multiplications. Lam and Hui proved that as t increases, the average number of multiplications approaches $\lceil t/(k+1) \rceil$.