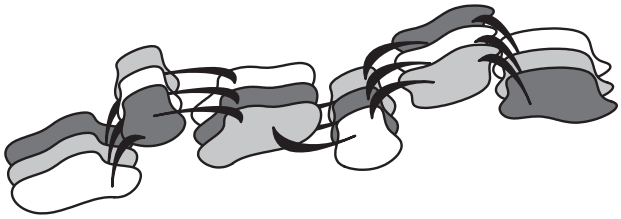
Chapter 10 Number Theory and Cryptography



Contents

Computers today are used for a multitude of sensitive applications. Customers utilize electronic commerce to make purchases and pay their bills. Businesses use the Internet to share sensitive company documents and interact with business partners. And universities use networks of computers to store personal information about students and their grades. Such sensitive information can be potentially damaging if it is altered, destroyed, or falls into the wrong hands. In this chapter, we discuss several powerful algorithmic techniques for protecting sensitive information, so as to achieve the following goals:

- ***Data integrity***: Information should not be altered without detection. For example, it is important to prevent the modification of purchase orders or other contractually binding documents transmitted electronically.
- ***Authentication***: Individuals and organizations that are accessing or communicating sensitive information must be correctly identified, that is, authenticated. For example, corporations offering telecommuting arrangements to their employees should set up an authentication procedure for accessing corporate databases through the Internet.
- ***Authorization***: Agents that are performing computations involving sensitive information must be authorized to perform those computations.
- ***Nonrepudiation***: In transactions that imply a contract, the parties that have agreed to that contract must not have the ability of backing out of their obligations without being detected.
- ***Confidentiality***: Sensitive information should be kept secret from individuals who are not authorized to see that information. That is, we must ensure that data is viewed by the sender and by the receiver, but not by unauthorized parties who can eavesdrop on the communication. For example, many email messages are meant to be confidential.

Many of the techniques we discuss in this chapter for achieving the above goals utilize number theory. Thus, we begin this chapter by discussing a number of important number theory concepts and algorithms. We describe the ancient, yet surprisingly efficient, Euclid's algorithm for computing greatest common divisors, as well as algorithms for computing modular exponents and inverses. In addition, because prime numbers play such a crucial role in cryptographic computations, we discuss efficient methods for testing if numbers are prime. We show how many of these number theory algorithms can be used in cryptographic algorithms that implement computer security services. We focus on encryption and digital signatures, including the popular RSA scheme. We also discuss, in this chapter, several protocols that can be built from these algorithms.

We conclude by discussing the fast Fourier transform (FFT), a general divide-and-conquer technique that can solve many problems with multiplication-like properties. We show how it use FFT to efficiently multiply polynomial and big integers. We also give a Java implementation of the FFT algorithm for multiplying big integers, and we empirically compare the performance of this algorithm to a standard multiplication method for big integers.

10.1 Fundamental Algorithms Involving Numbers

In this section we discuss several fundamental algorithms for performing important computations involving numbers. We describe efficient methods for computing exponents modulo n, for computing multiplicative inverses modulo n, and for testing if an integer n is prime. All of these computations have several important applications, including forming the critical algorithms in well-known cryptographic computations. But before we can present these algorithms, we must first present some basic facts from number theory. Throughout this discussion, we assume that all variables are integers. Also, proofs of some mathematical facts are left as exercises.

10.1.1 Some Facts from Elementary Number Theory

To get us started, we need some facts from elementary number theory, including some notation and definitions. Given positive integers a and b, we use the notation

$$a|b$$

to indicate that a ***divides*** b, that is, b is a multiple of a. If $a|b$, then we know that there is some integer k, such that $b = ak$. The following properties of divisibility follow immediately from this definition.

Theorem 10.1: *Let a, b, and c be arbitrary integers. Then*

- *If $a|b$ and $b|c$, then $a|c$.*
- *If $a|b$ and $a|c$, then $a|(ib + jc)$, for all integers i and j.*
- *If $a|b$ and $b|a$, then $a = b$ or $a = -b$.*

Proof: See Exercise R-10.1. ■

An integer p is said to be a ***prime*** if $p \geq 2$ and its only divisors are the trivial divisors 1 and p. Thus, in the case that p is prime, $d|p$ implies $d = 1$ or $d = p$. An integer greater than 2 that is not prime is said to be ***composite***. So, for example, 5, 11, 101, and 98 711 are prime, whereas 25 and 10 403 ($= 101 \cdot 103$) are composite. We also have the following:

Theorem 10.2 (Fundamental Theorem of Arithmetic): *Let $n > 1$ be an integer. Then there is a unique set of prime numbers $\{p_1, \ldots, p_k\}$ and positive integer exponents $\{e_1, \ldots, e_k\}$, such that*

$$n = p_1^{e_1} \cdots p_k^{e_k}.$$

The product $p_1^{e_1} \cdots p_k^{e_k}$ is known as the ***prime decomposition*** of n in this case. Theorem 10.2 and the notion of unique prime decomposition is the basis of several cryptographic schemes.

The Greatest Common Divisor (GCD)

The ***greatest common divisor*** of positive integers a and b, denoted $\gcd(a,b)$, is the largest integer that divides both a and b. Alternatively, we could say that $\gcd(a,b)$ is the number c, such that if $d|a$ and $d|b$, then $d|c$. If $\gcd(a,b) = 1$, we say that a and b are ***relatively prime***. We extend the notion of greatest common divisor to a pair of arbitrary integers by the following two rules:

- $\gcd(a,0) = \gcd(0,a) = a$.
- $\gcd(a,b) = \gcd(|a|,|b|)$, which takes care of negative values.

Thus, $\gcd(12,0) = 12$, $\gcd(10\,403,303) = 101$, and $\gcd(-12,78) = 6$.

The Modulo Operator

A few words about the ***modulo operator*** (mod) are in order. Recall that $a \bmod n$ is the remainder of a when divided by n. That is,

$$r = a \bmod n$$

means that

$$r = a - \lfloor a/n \rfloor n.$$

In other words, there is some integer q, such that

$$a = qn + r.$$

Note, in addition, that $a \bmod n$ is always an integer in the set $\{0,1,2,\ldots,n-1\}$, even when a is negative.

It is sometimes convenient to talk about ***congruence*** modulo n. If

$$a \bmod n = b \bmod n,$$

we say that a is ***congruent*** to b modulo n, which we call the ***modulus***, and we write

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

Therefore, if $a \equiv b \bmod n$, then $a - b = kn$ for some integer k.

Relating the Modulo Operator and the GCD

The following theorem gives an alternative characterization of the greatest common divisor. Its proof makes use of the modulo operator.

Theorem 10.3: *For any positive integers a and b, $\gcd(a,b)$ is the smallest positive integer d such that $d = ia + jb$ for some integers i and j. In other words, if d is the smallest positive integer linear combination of a and b, then $d = \gcd(a,b)$.*

Proof: Suppose d is the smallest integer such that $d = ia + jb$ for integers i and j. Note that, immediately from the definition of d, any common divisor of both a and b is also a divisor of d. Thus, $d \geq \gcd(a,b)$. To complete the proof, we need to show that $d \leq \gcd(a,b)$.

Let $h = \lfloor a/d \rfloor$. That is, h is the integer such that $a \bmod d = a - hd$. Then

$$\begin{aligned} a \bmod d &= a - hd \\ &= a - h(ia + jb) \\ &= (1 - hi)a + (-hj)b. \end{aligned}$$

In other words, $a \bmod d$ is also an integer linear combination of a and b. Moreover, by the definition of the modulo operator, $a \bmod d < d$. But d is the smallest positive integer linear combination of a and b. Thus, we must conclude that $a \bmod d = 0$, which implies that $d|a$. In addition, by a similar argument, we get that $d|b$. Thus, d is a divisor of both a and b, which implies $d \leq \gcd(a,b)$. ■

As we will show in Section 10.1.3, this theorem shows that the gcd function is useful for computing multiplicative modular inverses. In the next subsection, we show how to quickly compute the gcd function.

10.1.2 Euclid's GCD Algorithm

To compute the greatest common divisor of two numbers, we can use one of the oldest algorithms known, Euclid's algorithm. This algorithm is based on the following property of $\gcd(a,b)$:

Lemma 10.4: *Let a and b be two positive integers. For any integer r, we have*

$$\gcd(a,b) = \gcd(b, a - rb).$$

Proof: Let $d = \gcd(a,b)$ and $c = \gcd(b, a - rb)$. That is, d is the largest integer such that $d|a$ and $d|b$, and c is the largest integer such that $c|b$ and $c|(a - rb)$. We want to prove that $d = c$. By the definition of d, the number

$$(a - rb)/d = a/d - r(b/d)$$

is an integer. Thus, d divides both a and $a - rb$; hence, $d \leq c$.

By the definition of c, $k = b/c$ must be an integer, since $c|b$. Moreover, the number

$$(a - rb)/c = a/c - rk$$

must also be an integer, since $c|(a - rb)$. Thus, a/c must also be an integer, that is, $c|a$. Therefore, c divides both a and b; hence, $c \leq d$. We conclude then that $d = c$. ■

Lemma 10.4 leads us easily to an ancient algorithm, known as Euclid's algorithm, for computing the greatest common divisor (GCD) of two numbers, shown next in Algorithm 10.1.

Algorithm EuclidGCD(a,b):
  Input: Nonnegative integers a and b
  Output: gcd(a,b)
if $b = 0$ **then**
  return a
return EuclidGCD$(b, a \bmod b)$

Algorithm 10.1: Euclid's GCD algorithm.

An example of the execution of Euclid's algorithm is shown in Table 10.2.

	1	2	3	4	5	6	7
a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0

Table 10.2: Example of an execution of Euclid's algorithm to compute $\gcd(412,260) = 4$. The arguments a and b of each recursive invocation of method EuclidGCD$(412,260)$ are shown left-to-right, with the column headings showing the level of recursion in the EuclidGCD method.

Analyzing Euclid's Algorithm

The number of arithmetic operations performed by method EuclidGCD(a,b) is proportional to the number of recursive calls. So to bound the number of arithmetic operations performed by Euclid's algorithm we need only bound the number of recursive calls. First, we observe that after the first call, the first argument is always larger than the second one. For $i > 0$, let a_i be the first argument of the ith recursive call of method EuclidGCD. Clearly, the second argument of a recursive call is equal to a_{i+1}, the first argument of the next call. Also, we have

$$a_{i+2} = a_i \bmod a_{i+1},$$

which implies that the sequence of the a_i's is strictly decreasing. We will now show that the sequence decreases quickly. Specifically, we claim that

$$a_{i+2} < \frac{1}{2} a_i.$$

To prove the claim, we distinguish two cases:

Case 1: $a_{i+1} \leq \frac{1}{2} a_i$. Since the sequence of the a_i's is strictly decreasing, we have

$$a_{i+2} < a_{i+1} \leq \frac{1}{2} a_i.$$

Case 2: $a_{i+1} > \frac{1}{2} a_i$. In this case, since $a_{i+2} = a_i \bmod a_{i+1}$, we have

$$a_{i+2} = a_i \bmod a_{i+1} = a_i - a_{i+1} < \frac{1}{2} a_i.$$

Thus, the size of the first argument to the EuclidGCD method decreases by half with every other recursive call. We may therefore summarize the above analysis as follows.

Theorem 10.5: *Let a and b be two positive integers. Euclid's algorithm computes* $\gcd(a,b)$ *by executing* $O(\log \max(a,b))$ *arithmetic operations.*

We note that the complexity bound here is based on counting arithmetic operations. We can in fact improve the constant factor in the above bound by taking advantage of the fact that in modern times Euclid's algorithm should be implemented on a digital computer.

Binary Euclid's Algorithm

A variation of Euclid's algorithm, called the ***Binary Euclid's Algorithm***, takes into account the fact that integer division by 2 in a computer is faster than division by a general integer, since it can be accomplished by the ***right-shift*** native processor instruction. The binary Euclid's algorithm is shown in Algorithm 10.3. Like the original Euclid's algorithm, it computes the greatest common divisor of two integers, a and b, in $O(\log \max(a,b))$ arithmetic operations, though with a smaller constant factor. The justification of its correctness and the asymptotic analysis of its running time are explored in Exercise C-10.1.

Algorithm EuclidBinaryGCD(a,b):
  Input: Nonnegative integers a and b
  Output: $\gcd(a,b)$
if $a = 0$ **then**
  return b
else if $b = 0$ **then**
  return a
else if a is even and b is even **then**
  return $2 \cdot$ EuclidBinaryGCD$(a/2, b/2)$
else if a is even and b is odd **then**
  return EuclidBinaryGCD$(a/2, b)$
else if a is odd and b is even **then**
  return EuclidBinaryGCD$(a, b/2)$
else
  $\{a$ is odd and b is odd$\}$
  return EuclidBinaryGCD$(|a-b|/2, b)$

Algorithm 10.3: The Binary Euclid's Algorithm for computing the greatest common divisor of two nonnegative integers.

10.1.3 Modular Arithmetic

Let Z_n denote the set of nonnegative integers less than n:

$$Z_n = \{0, 1, \cdots, (n-1)\}.$$

The set Z_n is also called the set of ***residues*** modulo n, because if $b = a \bmod n$, b is sometimes called the ***residue*** of a modulo n. Modular arithmetic in Z_n, where operations on the elements of Z_n are performed $\bmod n$, exhibits properties similar to those of traditional arithmetic, such as the associativity, commutativity, distributivity of addition and multiplication, and the existence of identity elements 0 and 1 for addition and multiplication, respectively. Moreover, in any arithmetic expression, reducing each of its subexpressions modulo n produces the same result as computing the entire expression and then reducing that value modulo n. Also, every element x in Z_n has an ***additive inverse***, that is, for each $x \in Z_n$, there is a $y \in Z_n$ such that $x + y \bmod n = 0$. For example, the additive inverse of 5 modulo 11 is 6.

When it comes to multiplicative inverses, however, an important difference arises. Let x be an element of Z_n. A ***multiplicative inverse*** of x is an element $z^{-1} \in Z_n$ such that $xx^{-1} \equiv 1 \bmod n$. For example, the multiplicative inverse of 5 modulo 9 is 2, that is, $5^{-1} = 2$ in Z_9. As in standard arithmetic, 0 does not have a multiplicative inverse in Z_n. Interestingly, some nonzero elements also may not have a multiplicative inverse in Z_n. For example, 3 does not have a multiplicative inverse in Z_9. However, if n is prime, then every element $x \neq 0$ of Z_n has a multiplicative inverse in Z_n (1 is its own multiplicative inverse).

Theorem 10.6: *An element $x > 0$ of Z_n has a multiplicative inverse in Z_n if and only if $\gcd(x,n) = 1$ (that is, either $x = 1$ or x does not divide n).*

Proof: Suppose that $\gcd(x,n) = 1$. By Theorem 10.3, there are integers i and j such that $ix + jn = 1$. This implies $ix \bmod n = 1$, that is, $i \bmod n$ is the multiplicative inverse of x in Z_n, which proves the "if" part of the theorem.

To prove the "only if" part, suppose, for a contradiction, that $x > 1$ divides n, and there is an element y such that $xy \equiv 1 \bmod n$. We have $xy = kn + 1$, for some integer k. Thus, we have found integers $i = y$ and $j = -k$ such that $ix + jn = 1$. By Theorem 10.3, this implies that $\gcd(x,n) = 1$, a contradiction. ∎

If $\gcd(x,n) = 1$, we say x and n are ***relatively prime*** (1 is relatively prime to all other numbers). Thus, Theorem 10.6 implies that x has a multiplicative inverse in Z_n if and only if x is relatively prime to n. In addition, Theorem 10.6 implies that the sequence $0, x, 2x, 3x, \ldots, (n-1)x$ is simply a reordering of the elements of Z_n, that is, it is a permutation of the elements Z_n, as shown in the following.

Corollary 10.7: *Let $x > 0$ be an element of Z_n such that $\gcd(x,n) = 1$. Then*

$$Z_n = \{ix : \ i = 0, 1, \ldots, n-1\}.$$

Proof: See Exercise R-10.7. ∎

In Table 10.4, we show the multiplicative inverses of the elements of Z_{11} as an example. When the multiplicative inverse x^{-1} of x exists in Z_n, the notation y/x in an expression taken modulo n means "$yx^{-1} \bmod n$."

x	0	1	2	3	4	5	6	7	8	9	10
$x^{-1} \bmod 11$		1	6	4	3	9	2	8	7	5	10

Table 10.4: Multiplicative inverses of the elements of Z_{11}.

Fermat's Little Theorem

We now have enough machinery for our first major theorem, which is known as ***Fermat's Little Theorem***.

Theorem 10.8 (Fermat's Little Theorem): *Let p be a prime, and let x be an integer such that $x \bmod p \neq 0$. Then*

$$x^{p-1} \equiv 1 \pmod{p}.$$

Proof: It is sufficient to prove the result for $0 < x < p$, because

$$x^{p-1} \bmod p = (x \bmod p)^{p-1} \bmod p,$$

since we can reduce each subexpression "x" in "x^{p-1}" modulo p.

By Corollary 10.7, we know that for $0 < x < p$, the set $\{1,2,\ldots,p-1\}$ and the set $\{x\cdot 1, x\cdot 2, \ldots, x\cdot(p-1)\}$ contain exactly the same elements. So when we multiply the elements of the sets together, we get the same value, namely, we get

$$1\cdot 2\cdots(p-1) = (p-1)!.$$

In other words,

$$(x\cdot 1)\cdot(x\cdot 2)\cdots(x\cdot(p-1)) \equiv (p-1)! \pmod{p}.$$

If we factor out the x terms, we get

$$x^{p-1}(p-1)! \equiv (p-1)! \pmod{p}.$$

Since p is prime, every nonnull element in Z_p has a multiplicative inverse. Thus, we can cancel the term $(p-1)!$ from both sides, yielding $x^{p-1} \equiv 1 \bmod p$, the desired result. ■

In Table 10.5, we show the powers of the nonnull elements of Z_{11}. We observe the following interesting patterns:

- The last column of the table, with the values $x^{10} \bmod 11$ for $x = 1, \cdots, 10$, contains all ones, as given by Fermat's Little Theorem.
- In row 1, a subsequence of one element (1), is repeated ten times.
- In row 10, a subsequence of two elements, ending with 1, is repeated five times, since $10^2 \bmod 11 = 1$.
- In rows 3, 4, 5, and 9, a subsequence of five elements, ending with 1, is repeated twice.
- In each of the rows 2, 6, 7, and 8, the ten elements are all distinct.
- The lengths of the subsequences forming the rows of the table, and their number of repetitions, are the divisors of 10, that is, 1, 2, 5, and 10.

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

Table 10.5: Successive powers of the elements of Z_{11} modulo 11.

Euler's Theorem

Euler's ***totient function*** of a positive integer n, denoted $\phi(n)$, is defined as the number of positive integers less than or equal to n that are relatively prime to n. That is, $\phi(n)$ is equal to the number of elements in Z_n that have multiplicative inverses in Z_n. If p is a prime, then $\phi(p) = p - 1$. Indeed, since p is prime, each of the numbers $1, 2, \ldots, p-1$ are relatively prime to it, and $\phi(p) = p - 1$.

What if n isn't a prime number? Suppose $n = pq$, where p and q are primes. How many numbers are relatively prime to n? Well, initially, we observe that there are pq positive integers between 1 and n. However, q of them (including n) are multiples of p, and so they have a gcd of p with n. Similarly, there are p multiples of q (again, including n). Those multiples can't be counted in $\phi(n)$. Thus, we see that

$$\phi(n) = pq - q - (p-1) = (p-1)(q-1).$$

Euler's totient function is closely related to an important subset of Z_n known as the ***multiplicative group*** for Z_n, which is denoted as Z_n^*. The set Z_n^* is defined to be the set of integers between 1 and n that are relatively prime to n. If n is prime, then Z_n^* consists of the $n-1$ nonzero elements in Z_n, that is, $Z_n^* = \{1, 2, \ldots, n-1\}$ if n is prime. In general, Z_n^* contains $\phi(n)$ elements.

The set Z_n^* possesses several interesting properties, with one of the most important being that this set is closed under multiplication modulo n. That is, for any pair of elements a and b of Z_n^*, we have that $c = ab \bmod n$ is also in Z_n^*. Indeed, by Theorem 10.6, a and b have multiplicative inverses in Z_n. To see that Z_n^* has this closure property, let $d = a^{-1}b^{-1} \bmod n$. Clearly, $cd \bmod n = 1$, which implies that d is the multiplicative inverse of c in Z_n. Thus, again applying Theorem 10.6, we have that c is relatively prime to n, that is, $c \in Z_n^*$. In algebraic terminology, we say that Z_n^* is a ***group***, which is a shorthand way of saying that each element in Z_n^* has a multiplicative inverse and multiplication in Z_n^* is associative, has an identity, and is closed in Z_n^*.

The fact that Z_n^* has $\phi(n)$ elements and is a multiplicative group naturally leads to an extension of Fermat's Little Theorem. Recall that, in Fermat's Little Theorem, the exponent is $p-1 = \phi(p)$, since p is prime. As it turns out, a generalized form of Fermat's Little Theorem is true, too. This generalized form is presented in the following, which is known as ***Euler's Theorem***.

Theorem 10.9 (Euler's Theorem): *Let n be a positive integer, and let x be an integer such that* $\gcd(x,n) = 1$. *Then*

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

Proof: The proof technique is similar to that of Fermat's Little Theorem. Denote the elements of set Z_n^*, the multiplicative group for Z_n, as $u_1, u_2, \ldots, u_{\phi(n)}$. By the closure property of Z_n^* and Corollary 10.7,

$$Z_n^* = \{xu_i : \ i = 1, \cdots, \phi(n)\},$$

that is, multiplying elements in Z_n^* by x modulo n merely permutes the sequence $u_1, u_2, \ldots, u_{\phi(n)}$. Thus, multiplying together the elements of Z_n^*, we obtain

$$(xu_1) \cdot (xu_2) \cdots (xu_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \pmod{n}.$$

Again, we collect a term $x^{\phi(n)}$ on one side, giving us the congruence

$$x(u_1 u_2 \cdots u_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \pmod{n}.$$

Dividing by the product of the u_i's, gives us $x^{\phi(n)} \equiv 1 \bmod n$. ■

Theorem 10.9 gives a closed-form expression for the multiplicative inverses. Namely, if x and n are relatively prime, we can write

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

Generators

Given a prime p and an integer a between 1 and $p-1$, the ***order*** of a is the smallest exponent $e > 1$ such that

$$a^e \equiv 1 \bmod q.$$

A ***generator*** (also called ***primitive root***) of Z_p is an element g of Z_p with order $p-1$. We use the term "generator" for such an element a, because the repeated exponentiation of a can generate all of Z_p^*. For example, as shown in Table 10.5, the generators of Z_{11} are 2, 6, 7, and 8. Generators play an important role in many computations, including the Fast Fourier Transform algorithm discussed in Section 10.4. The existence of generators is established by the following theorem, stated without proof.

Theorem 10.10: *If p is a prime, then set Z_p has $\phi(p-1)$ generators.*

10.1.4 Modular Exponentiation

We address first exponentiation. The main issue in this case is to find a method other than the obvious brute-force. Before we describe an efficient algorithm, however, let us review the naive algorithm, for it already contains an important technique.

Brute-Force Exponentiation

One of the most important considerations in any exponentiation algorithms is to keep any intermediate results from getting too large. Suppose we want to compute, say $30\,192^{43\,791} \bmod 65\,301$. Multiplying 30 192 by itself 43 791 times and ***then*** taking the result modulo 65 301 will yield unpredictable results in most programming languages due to arithmetic overflows. Thus, we should take the modulo at each iteration, as shown in Algorithm 10.6.

Algorithm NaiveExponentiation(a, p, n):
  Input: Integers a, p, and n
  Output: $r = a^p \bmod n$

```
r ← 1
for i ← 1 to p do
    r ← (r · a) mod n
return r
```

Algorithm 10.6: A brute-force method for modular exponentiation.

This "naive" exponentiation algorithm is correct, but it is not very efficient, for it takes $\Theta(p)$ iterations to compute the modular exponentiation of a number to the power p. With large exponents, this running time is quite slow. Fortunately, there is a better method.

The Repeated Squaring Algorithm

A simple but important observation for an improved exponentiation algorithm is that squaring a number a^p is equivalent to multiplying its exponent p by two. In addition, multiplying two numbers a^p and a^q is equivalent to computing $a^{(p+q)}$. Let us therefore write an exponent p as a binary number $p_{b-1}\ldots p_0$, that is,

$$p = p_{b-1}2^{b-1} + \cdots + p_0 2^0.$$

Of course, each of the p_i's is either 1 or 0. Using the above observation, we can compute $a^p \bmod n$ by a variation of Horner's rule to evaluate polynomials, where the polynomial in question is the above binary expansion of the exponent p. Specifically, define q_i as the number whose binary representation is given by the leftmost i bits of p, that is, q_i is written in binary as $p_{b-1}\ldots p_{b-i}$. Clearly, we have $p = q_b$. Note that $q_1 = p_{b-1}$ and we can define q_i recursively as

$$q_i \quad = \quad 2q^{i-1} + p_{b-i} \quad \text{for} \quad 1 < i \leq b.$$

Thus, we can evaluate $a^p \bmod n$ with the recursive computation, called the ***repeated squaring*** method, given in Algorithm 10.7.

The main idea of this algorithm is to consider each bit of the exponent p in turn by dividing p by two until p goes to zero, squaring the current product Q_i for each such bit. In addition, if the current bit is a one (that is, p is odd), then we multiply in the base, a, as well. To see why this algorithm works, define, for $i = 1,\ldots,b$,

$$Q_i = a^{q_i} \bmod n.$$

From the recursive definition of q_i, we derive the following definition of Q_i:

$$\begin{array}{rcl} Q_i & = & (Q_{i-1}^2 \bmod n)a^{p_{b-i}} \bmod n \quad \text{for} \quad 1 < i \leq b \\ Q_1 & = & a^{p_{b-1}} \bmod n. \end{array} \tag{10.1}$$

It is easy to verify that $Q_b = a^p \bmod n$.

Algorithm FastExponentiation(a,p,n):
  Input: Integers a, p, and n
  Output: $r = a^p \bmod n$

```
if p = 0 then
  return 1
if p is even then
  t ← FastExponentiation(a, p/2, n) { p is even, so t = a^{p/2} mod n }
  return t^2 mod n
t ← FastExponentiation(a, (p−1)/2, n) { p is odd, so t = a^{(p−1)/2} mod n }
return a(t^2 mod n) mod n
```

Algorithm 10.7: Algorithm FastExponentiation for modular exponentiation using the repeated squaring method. Note that, since the modulo operator is applied after each arithmetic operation in method FastExponentiation, the size of the operands of each multiplication and modulo operation is never more than $2\lceil \log_2 n \rceil$ bits.

p	12	6	3	1	0
r	1	12	8	2	1

Table 10.8: Example of an execution of the repeated squaring algorithm for modular exponentiation. For each recursive invocation of FastExponentiation$(2,12,13)$, we show the second argument, p, and the output value $r = 2^p \bmod 13$.

We show a sample execution of the repeated squaring algorithm for modular exponentiation in Table 10.8.

The running time of the repeated squaring algorithm is easy to analyze. Referring to Algorithm 10.7, a constant number of arithmetic operations are performed, excluding those in the recursive call. Also, in each each recursive call, the exponent p gets halved. Thus, the number of recursive calls and arithmetic operations is $O(\log p)$. We may therefore summarize as follows.

Theorem 10.11: *Let a p, and n be positive integers, with $a < n$. The repeated squaring algorithm computes $a^p \bmod n$ using $O(\log p)$ arithmetic operations.*

10.1.5 Modular Multiplicative Inverses

We turn now to the problem of computing multiplicative inverses in Z_n. First, we recall Theorem 10.6, which states that a nonnegative element x of Z_n admits an inverse if and only if $\gcd(x,n) = 1$. The proof of Theorem 10.6 actually suggests a way to compute $x^{-1} \bmod n$. Namely, we should find the integers i and j referred to by Theorem 10.3, such that

$$ix + jn = \gcd(x,n) = 1.$$

If we can find such integers i and j, we immediately obtain

$$i \equiv x^{-1} \bmod n.$$

The computation of the integers i and j referred to by Theorem 10.3 can be done with a variation of Euclid's algorithm, called ***Extended Euclid's Algorithm***.

Extended Euclid's Algorithm

Let a and b be positive integers, and denote with d their greatest common divisor,

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

Let $q = a \bmod b$ and r be the integer such that $a = rb + q$, that is,

$$q = a - rb.$$

Euclid's algorithm is based on the repeated application of the formula

$$d = \gcd(a,b) = \gcd(b,q),$$

which immediately follows from Lemma 10.4.

Suppose that the recursive call of the algorithm, with arguments b and q, also returns integers k and l, such that

$$d = kb + lq.$$

Recalling the definition of r, we have

$$d = kb + lq = kb + l(a - rb) = la + (k - lr)b.$$

Thus, we have

$$d = ia + jb, \qquad \text{for } i = l \text{ and } j = k - lr.$$

This last equation suggests a method to compute the integers i and j. This method, known as the extended Euclid's algorithm, is shown in Algorithm 10.9. We present, in Table 10.10, a sample execution of this algorithm. Its analysis is analogous to that of Euclid's algorithm.

Theorem 10.12: *Let a and b be two positive integers. The extended Euclid's algorithm for computing a triplet of integers (d,i,j) such that*

$$d = \gcd(a,b) = ia + jb,$$

executes $O(\log \max(a,b))$ arithmetic operations.

Corollary 10.13: *Let x be an element of Z_n such that $\gcd(x,n) = 1$. The multiplicative inverse of x in Z_n can be computed with $O(\log n)$ arithmetic operations.*

Algorithm ExtendedEuclidGCD(a,b):
Input: Nonnegative integers a and b
Output: Triplet of integers (d,i,j) such that $d = \gcd(a,b) = ia + jb$

```
if b = 0 then
    return (a, 1, 0)
q ← a mod b
Let r be the integer such that a = rb + q
(d, k, l) ← ExtendedEuclidGCD(b, q)
return (d, l, k − lr)
```

Algorithm 10.9: Extended Euclid's algorithm.

a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0
r	1	1	1	2	2	5	
i	12	-7	5	-2	1	0	1
j	-19	12	-7	5	-2	1	0

Table 10.10: Execution of ExtendedEuclidGCD(a,b), for $a = 412$ and $b = 260$, to compute (d,i,j) such that $d = \gcd(a,b) = ia + jb$. For each recursive invocation, we show the arguments a and b, variable r, and output values i and j. The output value d is always $\gcd(412,260) = 4$.

10.1.6 Primality Testing

Prime numbers play an important role in computations involving numbers, including cryptographic computations. But how do we test whether a number n is prime, particularly if it is large?

Testing all possible divisors of n is computationally infeasible for large n. Alternatively, Fermat's Little Theorem (Theorem 10.8) seems to suggest an efficient solution. Perhaps we can somehow use the equation

$$a^{p-1} \equiv 1 \bmod p$$

to form a test for p. That is, let us pick a number a, and raise it to the power $p-1$. If the result is ***not*** 1, then the number p is definitely not prime. Otherwise, there's a chance it is. Would repeating this test for various values of a prove that p is prime? Unfortunately, the answer is "no." There is a class of numbers, called ***Carmichael numbers***, that have the property that $a^{n-1} \equiv 1 \bmod n$ for all $1 \leq a \leq n-1$, but n is composite. The existence of these numbers ruins such a simple test as that proposed above. Example Carmichael numbers are 561 and 1105.

A Template for Primality Testing

While the above "probabilistic" test won't work, there are several related tests that will, by making more sophisticated use of Fermat's Little Theorem. These probabilistic tests of primality are based on the following general approach. Let n be an odd integer that we want to test for primality, and let $\mathsf{witness}(x,n)$ be a Boolean function of a random variable x and n with the following properties:

1. If n is prime, then $\mathsf{witness}(x,n)$ is always false. So if $\mathsf{witness}(x,n)$ is true, then n is definitely composite.
2. If n is composite, then $\mathsf{witness}(x,n)$ is false with probability $q < 1$.

The function witness is said to be a ***compositeness witness function*** with error probability q, for q bounds the probability that witness will incorrectly identify a composite number as possibly prime. By repeatedly computing $\mathsf{witness}(x,n)$ for independent random values of the parameter x, we can determine whether n is prime with an arbitrarily small error probability. The probability that $\mathsf{witness}(x,n)$ would incorrectly return "false" for k independent random x's, when n is a composite number, is q^k. A generic probabilistic primality testing algorithm based on this observation is shown in Algorithm 10.11. This algorithm, which is described using a design technique known as the template method pattern, assumes that we have a compositeness witness function, witness, that satisfies the two conditions above. In order to turn this template into a full-blown algorithm, we need only specify the details of how to pick random numbers x and compute $\mathsf{witness}(x,n)$, the composite witness function.

Algorithm RandomizedPrimalityTesting(n,k):
  Input: Odd integer $n \geq 2$ and confidence parameter k
  Output: An indication of whether n is composite (which is always correct) or prime (which is incorrect with error probability 2^{-k})

```
{This method assumes we have a compositeness witness function witness(x,n)
with error probability q < 1.}
t ← ⌈k/log2(1/q)⌉
for i ← 1 to t do
  x ← random()
  if witness(x,n) then
    return "composite"
return "prime"
```

Algorithm 10.11: A template for a probabilistic primality testing algorithm based on a compositeness witness function witness(x,n). We assume that the auxiliary method random() picks a value at random from the domain of the random variable x.

If the method RandomizedPrimalityTesting$(n,k,\text{witness})$ returns "composite," we know with certainty that n is composite. However, if the method returns "prime," the probability that n is actually composite is no more than 2^{-k}. Indeed, suppose that n is composite but the method returns "prime." We have that the witness function witness(x,n) has evaluated to true for t random values of x. The probability of this event is q^t. From the relation between the confidence parameter k, the number of iterations t, and the error probability q of the witness function established by the first statement of the method, we have that $q^t \leq 2^{-k}$. The second argument, k, of the template method RandomizedPrimalityTesting is a ***confidence parameter***.

The Solovay-Strassen Primality Testing Algorithm

The ***Solovay-Strassen algorithm*** for primality testing is a specialization of the template method RandomizedPrimalityTesting. The compositeness witness function used by this algorithm is based on some number-theoretic facts, which we review below.

Let p be an odd prime. An element $a > 0$ of Z_p is said to be a ***quadratic residue*** if it is the square of some element x of Z_p, that is,

$$a \equiv x^2 \pmod{p}$$

For $a \geq 0$, the ***Legendre symbol*** $\left(\frac{a}{p}\right)$ is defined by:

$$\left(\frac{a}{b}\right) = \begin{cases} 1 & \text{if } a \bmod p \text{ is a quadratic residue} \\ 0 & \text{if } a \bmod p = 0 \\ -1 & \text{otherwise.} \end{cases}$$

The notation for the Legendre symbol should not be confused with the division operation. It can be shown (see Exercise C-10.2) that

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

We generalize the Legendre symbol by removing the restriction that p be a prime. Let n be a positive odd integer with prime decomposition

$$n = p_1^{e_1} \cdots p_k^{e_k}.$$

For $a \geq 0$, the ***Jacobi symbol***,

$$\left(\frac{a}{n}\right),$$

is defined by the following equation:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i} = \left(\frac{a}{p_1}\right)^{e_1} \cdot \left(\frac{a}{p_1}\right)^{e_1} \cdots \left(\frac{a}{p_k}\right)^{e_k}.$$

Like the Legendre symbol, the Jacobi symbol is equal to either 0, 1, or -1. We show in Algorithm 10.12 a recursive method for computing the Jacobi symbol. The justification of its correctness is omitted (see Exercise C-10.5).

Algorithm Jacobi(a,b):
Input: Integers a and b
Output: The value of the Jacobi symbol $\left(\frac{a}{b}\right)$

```
if a = 0 then
  return 0
else if a = 1 then
  return 1
else if a mod 2 = 0 then
  if (b^2 − 1)/8 mod 2 = 0 then
    return Jacobi(a/2, b)
  else
    return −Jacobi(a/2, b)
else if (a − 1)(b − 1)/4 mod 2 = 0 then
  return Jacobi(b mod a, a)
else
  return −Jacobi(b mod a, a)
```

Algorithm 10.12: Recursive computation of the Jacobi symbol.

If n is prime, then the Jacobi symbol $\left(\frac{a}{n}\right)$ is the same as the Legendre symbol. Thus, for any element a of Z_n, we have

$$\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}, \tag{10.2}$$

when n is prime. If n is composite, there may be values of a such that Equation 10.2 is still satisfied. So, if Equation 10.2 is satisfied, then we say that n is an ***Euler pseudo-prime*** with base a. The following lemma, given without proof, gives a property of Euler pseudo-primes that yields a compositeness witness function.

Lemma 10.14: *Let n be a composite number. There are at most $(n-1)/2$ positive values of a in Z_n such that n is an Euler pseudo-prime with base a.*

The Solovay-Strassen primality testing algorithm uses the following compositeness witness function:

$$\mathsf{witness}(x,n) = \begin{cases} \mathit{false} & \text{if } n \text{ is an Euler pseudo-prime with base } x \\ \mathit{true} & \text{otherwise,} \end{cases}$$

where x is a random integer with $1 < x \leq n-1$. By Lemma 10.14, this function has error probability $q \leq 1/2$. The Solovay-Strassen primality testing algorithm can be expressed as a specialization of the template method RandomizedPrimalityTesting (Algorithm 10.11) that redefines the auxiliary methods $\mathsf{witness}(x,n)$ and $\mathsf{random}()$, as shown in Algorithm 10.13.

Algorithm $\mathsf{witness}(x,n)$:
  return $(\mathsf{Jacobi}(x,n) \bmod n) \neq \mathsf{FastExponentiation}\left(x, \frac{n-1}{2}, n\right)$

Algorithm $\mathsf{random}()$:
  return a random integer between 1 and $n-1$

Algorithm 10.13: Solovay-Strassen algorithm obtained by specializing the auxiliary methods of algorithm RandomizedPrimalityTesting (Algorithm 10.11).

The analysis of the running time of the Solovay-Strassen algorithm is simple. Since the error probability of the compositeness witness function is no more than $1/2$, we can set $q = 2$, which implies that the number of iterations equal to the confidence parameter k. At each iteration, the computation of $\mathsf{witness}(x,n)$ takes $O(\log n)$ arithmetic operations (see Theorem 10.11 and Exercise C-10.5). We conclude as follows.

Theorem 10.15: *Given an odd positive integer n and a confidence parameter $k > 0$, the Solovay-Strassen algorithm determines whether n is prime with error probability 2^{-k} by performing $O(k \log n)$ arithmetic operations.*

The Rabin-Miller Primality Testing Algorithm

We now describe the ***Rabin-Miller algorithm*** for primality testing. It is based on Fermat's Little Theorem (Theorem 10.8) and on the following lemma.

Lemma 10.16: *Let $p > 2$ be a prime. If x is an element of Z_p such that*

$$x^2 \equiv 1 \pmod{p},$$

then either

$$x \equiv 1 \pmod{p}$$

or

$$x \equiv -1 \pmod{p}.$$

A ***nontrivial square root of the unity*** in Z_n is defined as an integer $1 < x < n-1$ such that

$$x^2 \equiv 1 \pmod{n}.$$

Lemma 10.16 states that if n is prime, there are no nontrivial square roots of the unity in Z_n.

For an odd integer n, let the binary representation of $n-1$ be

$$r_{b-1}r_{b-2}\cdots r_1 r_0.$$

Define s_i as the number whose binary representation is given by the leftmost i bits of $n-1$, that is, s_i is written in binary as

$$r_{b-1}\cdots r_{b-i}.$$

Given an integer x, define the element X_i of Z_n as

$$X_i = x^{s_i} \bmod n.$$

The Rabin-Miller algorithm defines its compositeness witness function (that is, $\mathsf{witness}(x,n)$) so that it is true if and only if $x^{n-1} \bmod n \neq 1$. X_i is a nontrivial square root of the unity for some $1 < i < b-1$. The computation of this function is easier than it may seem. Indeed, if we compute $x^{n-1} \bmod n$ using the repeated squaring algorithm (Algorithm 10.7), the integers X_i are just a byproduct of the computation (see Exercise C-10.6). The error probability is provided by the following lemma, stated without proof.

Lemma 10.17: *Let n be a composite number. There are at most $(n-1)/4$ positive values of x in Z_n such that the Rabin-Miller compositeness witness function* $\mathsf{witness}(x,n)$ *returns true.*

We conclude as follows.

Theorem 10.18: *Given an odd positive integer n and a confidence parameter $k > 0$, the Rabin-Miller algorithm determines whether n is prime, with error probability 2^{-k}, by performing $O(k \log n)$ arithmetic operations.*

The Rabin-Miller algorithm is widely used in practice for primality testing.

Finding Prime Numbers

A primality testing algorithm can be used to select a random prime in a given range, or with a prespecified number of bits. We exploit the following result from number theory, stated without proof.

Theorem 10.19: *The number, $\pi(n)$, of primes that are less than or equal to n is $\Theta(n/\ln n)$. In fact, if $n \geq 17$, then $n/\ln n < \pi(n) < 1.26n/\ln n$.*

In the above theorem, $\ln n$ is the natural logarithm of n, that is, the logarithm of n in base e, where e is Euler's number, a transcendental number whose first few digits are $2.7182818284590452353 6\ldots$.

A consequence of Theorem 10.19 is that a random integer n is prime with probability $1/\ln n$. Thus, to find a prime with a given number b of bits, we generate random b-bit odd numbers and test them for primality until we find a prime number.

Theorem 10.20: *Given an integer b and a confidence parameter k, a random prime with b bits can be selected with error probability 2^{-k} by performing $O(kb)$ arithmetic operations.*

10.2 Cryptographic Computations

The Internet is enabling a growing number of activities, such as correspondence (email), shopping (Web stores), and financial transactions (online banking), to be performed electronically. However, the Internet itself is an insecure transmission network: data transmitted over the Internet travels through several intermediate specialized computers, called ***routers***, which can observe the data and potentially modify it.

A variety of cryptographic techniques have been developed to support secure communication over an insecure network such as the Internet. In particular, cryptography research has developed the following useful cryptographic computations:

- ***Encryption/decryption***: A message M to be transmitted, called the ***plaintext***, is transformed into an unrecognizable string of characters C, called the ***ciphertext***, before being sent over the network. This transformation is known as ***encryption***. After the ciphertext C is received, it is converted back to the plaintext M using an inverse transformation (that depends on additional secret information). This reverse transformation is called ***decryption***. An essential ingredient in encryption is that it should be computationally infeasible for an outsider to transform C back to M (without knowing the secret information possessed by the receiver).
- ***Digital signatures***: The author of a message M computes a message S that is derived from M and secret information known by the author. The message S is a ***digital signature*** if another party can easily verify that only the author of M could have computed S in a reasonable amount of time.

Using Cryptographic Computations for Information Security Services

The computations of encryption and digital signatures are sometimes combined with other cryptographic computations, some of which we discuss later in this chapter. Still, the two techniques above are already sufficient to support the information security services discussed in the introduction:

- ***Data integrity***: Computing a digital signature S of a message M not only helps us determine the author of M, it also verifies the integrity of M, for a modification to M would produce a different signature. So, to perform a data integrity check we can perform a verification test that S is, in fact, a digital signature for the message M.
- ***Authentication***: The above cryptographic tools can be used for authentication in two possible ways. In ***password*** authentication schemes, a user will type a user-id and password in a client application, with this combination being immediately encrypted and sent to an authenticator. If the encrypted user-id and password combination matches that in a user database, then the individual is authenticated (and the database never stores passwords in plaintext). Alternatively, an authenticator can issue a challenge to a user in the form of a random message M that the user must immediately digitally sign for authentication.
- ***Authorization***: Given a scheme for authentication, we can issue authorizations by keeping lists, called ***access control lists***, that are associated with sensitive data or computations that should be accessed only by authorized individuals. Alternatively, the holder of a right to sensitive data or computations can digitally sign a message C that authorizes a user to perform certain tasks. For example, the message could be of the form, "I U.S. Corporation vice president give person x permission to access our fourth quarter earnings data."
- ***Confidentiality***: Sensitive information can be kept secret from nonauthorized agents by encrypting it.
- ***Nonrepudiation***: If we make the parties negotiating a contract, M, digitally sign that message, then we can have a way of proving that they have seen and agreed to the content of the message M.

This section gives an introduction to cryptographic computations. Conventional names of personae, such as Alice, Bob, and Eve, are used to denote the parties involved in a cryptographic protocol. We focus primarily on ***public-key cryptography***, which is based on the number-theoretic properties and algorithms discussed in the previous section. Still, before we introduce the concepts of public-key cryptography, we briefly discuss an alternate approach to encryption.

10.2.1 Symmetric Encryption Schemes

As mentioned above, a fundamental problem in cryptography is confidentiality, that is, sending a message from Alice to Bob so that a third party, Eve, cannot gain any information from an intercepted copy of the message. Moreover, we have observed that confidentiality can be achieved by ***encryption schemes***, or ***ciphers***, where the message M to be transmitted, called the ***plaintext***, is ***encrypted*** into an unrecognizable string of characters C, called the ***ciphertext***, before being sent over the network. After the ciphertext C is received, it is decrypted back to the plaintext M using an inverse transformation called ***decryption***.

Secret Keys

In describing the details of an encryption scheme, we must explain all the steps needed in order to encrypt a plaintext M into a ciphertext C, and how to then decrypt that ciphertext back to M. Moreover, in order for Eve to be unable to extract M from C, there must be some secret information that is kept private from her.

In traditional cryptography, a common ***secret key*** k is shared by Alice and Bob, and is used to both encrypt and decrypt the message. Such schemes are also called ***symmetric encryption*** schemes, since k is used for both encryption and decryption and the same secret is shared by both Alice and Bob.

Substitution Ciphers

A classic example of a symmetric cipher is a ***substitution cipher***, where the secret key is a permutation π of the characters of the alphabet. Encrypting plaintext M into ciphertext C consists of replacing each character x of M with character $y = \pi(x)$. Decryption can be easily performed by knowing the permutation function π. Indeed, M is derived from C by replacing each character y of C with character $x = \pi^{-1}(y)$. The ***Caesar cipher*** is an early example of a substitution cipher, where each character x is replaced by character

$$y = x + k \bmod n,$$

where n is the size of the alphabet and $1 < k < n$ is the secret key. This substitution scheme is known as the "Caesar cipher," for Julius Caesar is known to have used it with $k = 3$.

Substitution ciphers are quite easy to use, but they are not secure. Indeed, the secret key can be quickly inferred using ***frequency analysis***, based on the knowledge of the frequency of the various letters, or groups of consecutive letters in the text language.

The One-Time Pad

Secure symmetric ciphers exist. Indeed, the most secure cipher known is a symmetric cipher. It is the ***one-time pad***. In this cryptosystem, Alice and Bob each share a random bit string K as large as any message they might wish to communicate. The string K is the symmetric key, for to compute a ciphertext C from a message M, Alice computes

$$C = M \oplus K,$$

where "$\oplus$" denotes the bitwise exclusive-or operator. She can send C to Bob using any reliable communication channel, even one on which Eve is eavesdropping, because the ciphertext C is computationally indistinguishable from a random string. Nevertheless, Bob can easily decrypt the ciphertext message C by computing $C \oplus K$, since

$$\begin{aligned} C \oplus K &= (M \oplus K) \oplus K \\ &= M \oplus (K \oplus K) \\ &= M \oplus \mathbf{0} \\ &= M, \end{aligned}$$

where $\mathbf{0}$ denotes the bit string of all 0's the same length as M. This scheme is clearly a symmetric cipher system, since the key K is used for encryption and decryption.

The one-time pad is computationally efficient, for bitwise exclusive-or is one of the fastest operators that computers can perform. Also, as already mentioned, the one-time pad is incredibly secure. Nevertheless, the one-time pad cryptosystem is not widely used. The main trouble with this system is that Alice and Bob must share a very large secret key. Moreover, the security of the one-time pad depends crucially on the fact that the secret key K is used only once. If K is reused, there are several simple cryptoanalyses that can break this system. For practical cryptosystems, we prefer secret keys that can be reused and are smaller than the messages they encrypt and decrypt.

Other Symmetric Ciphers

Secure and efficient symmetric ciphers do exist. They are referred to by their acronyms or colorful names, such as "3DES," "IDEA," "Blowfish," and "Rijndael" (pronounce "Rhine-doll"). They perform a sequence of complex substitution and permutation transformations on the bits of the plaintext. While these systems are important in many applications, they are only mildly interesting from an algorithmic viewpoint; hence, they are out of the scope of this book. They run in time proportional to the length of the message being encrypted or decrypted. Thus, we mention that these algorithms exist and are fast, but in this book we do not discuss any of these efficient symmetric ciphers in any detail.

10.2.2 Public-Key Cryptosystems

A major problem with symmetric ciphers is ***key transfer***, or how to distribute the secret key for encryption and decryption. In 1976, Diffie and Hellman described an abstract system that would avoid these problems, the ***public-key cryptosystem***. While they didn't actually publish a particular public-key system, they discussed the features of such a system. Specifically, given a message M, encryption function E, and decryption function D, the following four properties must hold:

1. $D(E(M)) = M$.
2. Both E and D are easy to compute.
3. It is computationally infeasible[1] to derive D from E.
4. $E(D(M)) = M$.

In retrospect, these properties seem fairly common sense. The first property merely states that, once a message has been encrypted, applying the decryption procedure will restore it. Property two is perhaps more obvious. In order for a cryptosystem to be practical, encryption and decryption must be computationally fast.

The third property is the start of the innovation. It means that E only goes one way; it is computationally infeasible to invert E, unless you already know D. Thus, the encryption procedure E can be made public. Any party can send a message, while only one knows how to decrypt it.

If the fourth property holds, then the mapping is one-to-one. Thus, the cryptosystem is a solution to the ***digital signature*** problem. Given an electronic message from Bob to Alice, how can we prove that Bob actually sent it? Bob can apply his decryption procedure to some signature message M. Any other party can then verify that Bob actually sent the message by applying the public encryption procedure E. Since only Bob knows the decryption function, only Bob can generate a signature message which can be correctly decoded by the function E.

Public-key cryptography is the basis of modern cryptography. Its economic importance is fast growing, since it provides the security infrastructure of all electronic transactions over the Internet.

The design of public-key cryptosystems can be described in general terms. The idea is to find a very tough problem in computer science, and then somehow tie the cryptosystem to it. Ideally, one arrives at an actual proof that breaking the cryptosystem is computationally equivalent to solving the difficult problem. There's a large class of problems, called ***NP-complete***, which do not have known polynomial time algorithms for their solution. (See Chapter 13.) In fact, it is widely believed that there are none. Then, to generate the particular encryption and decryption keys, we create a particular set of parameters for this problem. Encrypting then means turning the message into an instance of the problem. The recipient can use secret information (the decryption key) to solve the puzzle effortlessly.

[1] The concept of computational difficulty is formalized in Chapter 13.

10.2.3 The RSA Cryptosystem

Some care must be taken in how a computationally difficult problem is tied to a cryptosystem. One of the earlier public-key cryptosystems, the Merkle-Hellman system, linked encryption to something called the knapsack problem, which is ***NP***-complete. Unfortunately, the problems the system generates turn out to be a special subclass of the knapsack problem that can be easily solved. So designing public-key cryptosystems has its share of subtleties.

Probably the most well-known public-key cryptosystem is also one of the oldest, and is tied to the difficulty of factoring large numbers. It is named ***RSA*** after its inventors, Rivest, Shamir, and Adleman.

In this cryptosystem, we begin by selecting two large primes, p and q. Let $n = pq$ be their product and recall that $\phi(n) = (p-1)(q-1)$. Encryption and decryption keys e and d are selected so that

- e and $\phi(n)$ are relatively prime
- $ed \equiv 1 \pmod{\phi(n)}$.

The second condition means that d is the multiplicative inverse of $e \bmod \phi(n)$. The pair of values n and e form the public key, while d is the private key. In practice, e is chosen either randomly or as one of the following numbers: 3, 17, or 65 537.

The rules for encrypting and decrypting with RSA are simple. Let us assume, for simplicity, that the plaintext is an integer M, with $0 < M < n$. If M is a string, we can view it as an integer by concatenating the bits of its characters. The plaintext M is encrypted into ciphertext C with one modular exponentiation using the encryption key e as the exponent:

$$C \leftarrow M^e \bmod n \quad \text{(RSA encryption)}.$$

The decryption of ciphertext C is also performed with an exponentiation, using now the decryption key d as the exponent:

$$M \leftarrow C^d \bmod n \quad \text{(RSA decryption)}.$$

The correctness of the above encryption and decryption rules is justified by the following theorem.

Theorem 10.21: *Let p and q be two odd primes, and define $n = pq$. Let e be relatively prime with $\phi(n)$ and let d be the multiplicative inverse of e modulo $\phi(n)$. For each integer x such that $0 < x < n$,*

$$x^{ed} \equiv x \pmod{n}.$$

Proof: Let $y = x^{ed} \bmod n$. We want to prove that $y = x$. Because of the way we have selected e and d, we can write $ed = k\phi(n) + 1$, for some integer k. Thus, we have

$$y = x^{k\phi(n)+1} \bmod n.$$

We distinguish two cases.

Case 1: x does not divide n. We rewrite y as follows:

$$\begin{aligned} y &= x^{k\phi(n)+1} \bmod n \\ &= xx^{k\phi(n)} \bmod n \\ &= x(x^{\phi(n)} \bmod n)^k \bmod n. \end{aligned}$$

By Theorem 10.9 (Euler's theorem), we have $x^{\phi(n)} \bmod n = 1$, which implies $y = x \cdot 1^k \bmod n = x$.

Case 2: x divides n. Since $n = pq$, with p and q primes, x is a multiple of either p or q. Suppose x is a multiple of p, that is, $x = hp$ for some positive integer h. Clearly, x cannot be a multiple of q as well, since otherwise x would be greater than $n = pq$, a contradiction. Thus, $\gcd(x,q) = 1$ and by Theorem 10.9 (Euler's theorem), we have

$$x^{\phi(q)} \equiv 1 \pmod{q}.$$

Since $\phi(n) = \phi(p)\phi(q)$, raising both sides of the above congruence to the power of $k\phi(q)$, we obtain

$$x^{k\phi(n)} \equiv 1 \pmod{q},$$

which we rewrite as

$$x^{k\phi(n)} = 1 + iq,$$

for some integer i. Multiplying both sides of the above equality by x, and recalling that $x = hp$ and $n = pq$, we obtain:

$$\begin{aligned} x^{k\phi(n)+1} &= x + xiq \\ &= x + hpiq \\ &= x + (hi)n. \end{aligned}$$

Thus, we have

$$y = x^{k\phi(n)+1} \bmod n = x.$$

In either case, we have shown that $y = x$, which concludes the proof of the theorem. ∎

Using RSA for Digital Signatures

The symmetry of the encryption and decryption functions implies that the RSA cryptosystem directly supports digital signatures. Indeed, a digital signature S for message M is obtained by applying the decryption function to M, that is,

$$\boxed{S \leftarrow M^d \bmod n \quad \text{(RSA signature)}.}$$

The verification of the digital signature S is now performed with the encryption function, that is, by checking that

$$\boxed{M \equiv S^e \pmod{n} \quad \text{(RSA verification)}.}$$

The Difficulty of Breaking RSA

Note that, even if we know the value e, we cannot figure out d unless we know $\phi(n)$. Most cryptography researchers generally believe that breaking RSA requires that we compute $\phi(n)$ and that this requires factoring n. While there is no ***proof*** that factorization is computationally difficult, a whole series of famous mathematicians have worked on the problem over the past few hundred years. Especially if n is large (≈ 200 digits), it will take a very long time to factor it. To give you an idea of the state of the art, mathematicians were quite excited when a nationwide network of computers was able to factor the ninth Fermat number, $2^{512}-1$. This number has "only" 155 decimal digits. Barring a major breakthrough, the RSA system will remain secure. For if technology somehow advances to a point where it is feasible to factor 200 digit numbers, we need only choose an n with three or four hundred digits.

Analysis and Setup for RSA Encryption

The running time of RSA encryption, decryption, signature, and verification is simple to analyze. Indeed, each such operation requires a constant number of modular exponentiations, which can be performed with method FastExponentiation (Algorithm 10.7).

Theorem 10.22: *Let n be the modulus used in the RSA cryptosystem. RSA encryption, decryption, signature, and verification each take $O(\log n)$ arithmetic operations.*

To set up the RSA cryptosystem, we need to generate the public and private key pair. Namely, we need to compute the private key (d,p,q) and the public key (e,n) that goes with it. This involves the following computations:

- Selection of two random primes p and q with a given number of bits. This can be accomplished by testing random integers for primality, as discussed at the end of Section 10.1.6.
- Selection of an integer e relatively prime to $\phi(n)$. This can be done by picking random primes less than $\phi(n)$ until we find one that does not divide $\phi(n)$. In practice, it is sufficient to check small primes from a list of known primes (often $e=3$ or $e=17$ will work).
- Computing the multiplicative inverse d of e in $Z_{\phi(n)}$. This can be done using the extended Euclid's algorithm (Corollary 10.13).

We have previously explained algorithms for each of these number theory problems in this chapter.

10.2.4 The El Gamal Cryptosystem

We have seen that the security of the RSA cryptosystem is related to the difficulty of factoring large numbers. It is possible to construct cryptosystems based on other difficult number-theoretic problems. We now consider the El Gamal cryptosystem, named after its inventor, Taher El Gamal, which is based on the difficulty of a problem called the "discrete logarithm."

The Discrete Logarithm

When we're working with the real numbers, $\log_b y$ is the value x, such that $b^x = y$. We can define an analogous discrete logarithm. Given integers b and n, with $b < n$, the ***discrete logarithm*** of an integer y to the base b is an integer x, such that

$$b^x \equiv y \bmod n.$$

The discrete logarithm is also called ***index***, and we write

$$x = \mathrm{ind}_{b,n} y.$$

While it is quite efficient to raise numbers to large powers modulo p (recall the repeated squaring algorithm, Algorithm 10.7), the inverse computation of the discrete logarithm is much harder. The El Gamal system relies on the difficulty of this computation.

El Gamal Encryption

Let p be a prime, and g be a generator of Z_p. The private key x is an integer between 1 and $p-2$. Let $y = g^x \bmod p$. The public key for El Gamal encryption is the triplet (p,g,y). If taking discrete logarithms is as difficult as it is widely believed, releasing $y = g^x \bmod p$ does not reveal x.

To encrypt a plaintext M, a random integer k relatively prime to $p-1$ is selected, and the following pair of values is computed:

$$\begin{array}{l} a \leftarrow g^k \bmod p \\ b \leftarrow My^k \bmod p \end{array} \quad \text{(El Gamal encryption).}$$

The ciphertext C consists of the pair (a,b) computed above.

El Gamal Decryption

The decryption of the ciphertext $C = (a,b)$ in the El Gamal scheme, to retrieve the plaintext M, is simple:

$$M \leftarrow b/a^x \bmod p \quad \text{(El Gamal decryption).}$$

In the above expression, the "division" by a^x should be interpreted in the context of modular arithmetic, that is, M is multiplied by the inverse of a^x in Z_p. The correctness of the El Gamal encryption scheme is easy to verify. Indeed, we have

$$\begin{aligned} b/a^x \bmod p &= My^k(a^x)^{-1} \bmod p \\ &= Mg^{xk}(g^{kx})^{-1} \bmod p \\ &= M. \end{aligned}$$

Using El Gamal for Digital Signatures

A variation of the above scheme provides a digital signature. Namely, a signature for message M is a pair $S = (a,b)$ obtained by selecting a random integer k relatively prime to $p-1$ (which, of course, equals $\phi(p)$) and computing

$$\boxed{\begin{array}{rcl} a & \leftarrow & g^k \bmod p \\ b & \leftarrow & k^{-1}(M - xa) \bmod (p-1) \end{array} \quad \text{(El Gamal signature).}}$$

To verify a digital signature $S = (a,b)$, we check that

$$\boxed{y^a a^b \equiv g^M \pmod p \quad \text{(El Gamal verification).}}$$

The correctness of the El Gamal digital signature scheme can be seen as follows:

$$\begin{aligned} y^a a^b \bmod p &= ((g^x \bmod p)^a \bmod p)((g^k \bmod p)^{k^{-1}(M-xa) \bmod (p-1)} \bmod p) \\ &= g^{xa} g^{kk^{-1}(M-xa) \bmod (p-1)} \bmod p \\ &= g^{xa+M-xa} \bmod p \\ &= g^M \bmod p. \end{aligned}$$

Analysis of El Gamal Encryption

The analysis of the performance of the El Gamal cryptosystem is similar to that of RSA. Namely, we have the following.

Theorem 10.23: *Let n be the modulus used in the El Gamal cryptosystem. El Gamal encryption, decryption, signature, and verification each take $O(\log n)$ arithmetic operations.*

10.3 Information Security Algorithms and Protocols

Once we have some tools, like the fundamental algorithms involving numbers and the public-key encryption methods, we can start to compose them with other algorithms to provide needed information security services. We discuss several such protocols in this section, many of which use, in addition to the algorithms discussed above, the topic we discuss next.

10.3.1 One-way Hash Functions

Public-key cryptosystems are often used in conjunction with a ***one-way hash function***, also called a ***message digest*** or ***fingerprint***. We provide an informal description of such a function next. A formal discussion is beyond the scope of this book.

A ***one-way hash function*** H maps a string (message) M of arbitrary length to an integer $d = H(M)$ with a fixed number of bits, called the ***digest*** of M, that satisfies the following properties:

1. Given a string M, the digest of M can be computed quickly.
2. Given the digest d of M, but not M, it is computationally infeasible to find M.

A one-way hash function is said to be ***collision-resistant*** if, given a string M, it is computationally infeasible to find another string M' with the same digest, and is said to be ***strongly collision-resistant*** if it is computationally infeasible to find two strings M_1 and M_2 with the same digest.

Several functions believed to be strongly collision-resistant, one-way hash functions have been devised. The ones used most in practice are MD5, which produces a 128-bit digest, and SHA-1, which produces a 160-bit digest. We examine now some applications of one-way hashing.

A first application of one-way hash functions is to speed up the construction of digital signatures. If we have a collision-resistant, one-way hash function, we can sign the digest of a message instead of the message itself, that is, the signature S is given by:

$$S = D(H(M)).$$

Except for small messages, hashing the message and signing the digest is faster, in practice, than signing the message directly. Also, this procedure overcomes a significant restriction of the RSA and El Gamal signature schemes, namely that the message M must be less than the modulus n. For example, when using MD5, we can sign messages of arbitrary length using a fixed modulus n that is only required to be greater than 2^{128}.

10.3.2 Timestamping and Authenticated Dictionaries

The next application is ***timestamping***. Alice has a document M and wants to obtain a certification that the document M exists at the present time t.

In one method for timestamping, Alice uses the services of a trusted third party, Trevor, to provide timestamping. Alice can send M to Trevor and have him sign a new document M' consisting of the concatenation of M and t. While this approach works, it has the drawback that Trevor can see M. A collision-resistant, one-way hash function H can eliminate the problem. Alice computes the digest d of M using H, and asks Trevor to sign a new message M'' consisting of the concatenation of d and t.

Authenticated Dictionaries

In fact, we can define another method for timestamping that does not require that we fully trust Trevor. This alternative method is based on the concept of an ***authenticated dictionary***.

In an authenticated dictionary, the third party, Trevor, collects a dictionary database of items. In the timestamping application, the items are the digests of documents that need to be timestamped as existing on a certain date. In this case, however, we do not trust Trevor's signed statement that a digest d of Alice's document M exists on a certain date t. Instead, Trevor computes a digest D of the entire dictionary and he publishes D in a location where timestamping cannot be questioned (such as the classified section of a well-known newspaper). In addition, Trevor responds to Alice with partial digest D' that summarizes all the items in the dictionary except for Alice's document digest d.

For this scheme to work effectively, there should be a function f such that $D = f(D', d)$, with f being easy to compute (for Alice). But this function should be one-way in the sense that, given an arbitrary y, it should be computationally difficult (for Trevor) to compute an x such that $D = f(x, y)$. Given such a function, we can rely on Trevor to compute the digest of all the documents he receives and publish that digest to the public location. For it is computationally infeasible for Trevor to fabricate a response that would indicate that a value d was in the dictionary when in fact it was not. Thus, the key component of this protocol is the existence of the one-way function f. In the remainder of this subsection we explore a possible method for constructing a function f suitable for an authenticated dictionary.

Hash Trees

An interesting data structure approach, known as the ***hash tree*** scheme, can be used to implement an authenticated dictionary. This structure supports the initial construction of the dictionary database followed by query operations or membership responses for each item.

A hash tree T for a set S stores the elements of S at the external nodes of a complete binary tree T and a hash value $h(v)$ at each node v, which combines the hash of its children using a well-known, one-way hash function. In the timestamping application the items stored at the external nodes of T are themselves digests of documents to be timestamped as existing on a particular day. The authenticated dictionary for S consists of the hash tree T plus the publication of the value $h(r)$ stored in the root r of T. An element x is proven to belong to S by reporting the values stored at the nodes on the path in T from the node storing x to the root, together with the values of all nodes that have siblings on this path.

Given such a path p, Alice can recompute the hash value $h(r)$ of the root. Moreover, since T is a complete binary tree, she need only perform $O(\log n)$ calls to the hash function h to compute this value, where n is the number of elements in S.

10.3.3 Coin Flipping and Bit Commitment

We now present a protocol that allows Alice and Bob to ***flip a random coin*** by exchanging email messages or otherwise communicating over a network. Let H be a strongly collision-resistant, one-way hash function. The interaction between Alice and Bob consists of the following steps:

1. Alice picks a number x and computes the digest $d = H(x)$, sending d to Bob.
2. After receiving d, Bob sends Alice his guess of whether x is odd or even.
3. Alice announces the result of the coin flip: if Bob has guessed correctly, the result is heads; if not, it is tails. She also sends to Bob x as proof of the result.
4. Bob verifies that Alice has not cheated, that is, that $d = H(x)$.

The strong collision-resistance requirement is essential, since otherwise Alice could come up with two numbers, one odd and one even, with the same digest d, and would be able to control the outcome of the coin flip.

Related to coin flipping is ***bit commitment***. In this case, Alice wants to commit to a value n (which could be a single bit or an arbitrary string) without revealing it to Bob. Once Alice reveals n, Bob wants to verify that she has not cheated. For example, Alice may want to prove to Bob that she can predict whether a certain stock will be up ($n = 1$), down ($n = -1$), or unchanged ($n = 0$) tomorrow. Using a strongly collision-resistant, one-way hash function H, the protocol goes as follows:

1. She sends Bob x plus the digest of the concatenation of x, y, and n. In keeping with tradition in cryptographic literature, we denote the concatenation of strings a and b with "$a||b$" in this chapter. Thus, using this notation, Alice sends Bob $d = H(x||y||n)$. Note that Bob is unable to figure out n from x and d.
2. At the close of trading the next day, when n becomes publicly known, Alice sends y to Bob for verification.
3. Bob verifies that Alice has not cheated, that is, $d = H(x||y||n)$.

10.3.4 The Secure Electronic Transaction (SET) Protocol

Our final application is significantly more complex and involves the combined use of encryption, digital signatures, and one-way hashing. It is actually a simplified version of the ***SET*** (secure electronic transaction) protocol for secure credit card payment over the Internet.

Alice wants to purchase a book from Barney (an Internet bookstore) using a credit card issued by Lisa (a bank). Alice is concerned about privacy: on one hand, she does not want Barney to see her credit card number; on the other hand, she does not want Lisa to know which book she purchased from Barney. However, she wants Barney to send her the book, and Lisa to send the payment to Barney. Finally, Alice also wants to ensure that the communication between Barney, Lisa, and herself is kept confidential even if someone is eavesdropping over the Internet. Using a strongly collision-resistant, one-way hash function H, the protocol is described in Algorithm 10.14.

Properties of the SET Protocol

The following observations show that the confidentiality, integrity, nonrepudiation, and authentication requirements of the protocol are satisfied.

- Barney cannot see Alice's credit card number, which is stored in the payment slip P. Barney has the digest p of P. However, he cannot compute P from p since H is one-way. Barney also has the ciphertext C_L of a message that contains P. However, he cannot decrypt C_L since he does not have Lisa's private key.
- Lisa cannot see the book ordered by Alice, which is stored in the purchase order O. Lisa has the digest o of O. However, she cannot compute O from o since H is one-way.
- The digital signature S provided by Alice serves a dual purpose. It allows Barney to verify the authenticity of Alice's purchase order O, and Alice to verify the authenticity of Alice's payment slip P.
- Alice cannot deny that she ordered the specific book indicated in O and to have charged her credit card for the given amount indicated in P. Indeed, since H is collision-resistant, she cannot forge a different purchase order and/or payment slip that hash to the same digests o and p.
- All communication between the parties uses public-key encryption and ensures confidentiality, even in the presence of eavesdroppers.

Thus, although it is somewhat intricate, the SET protocol illustrates how cryptographic computations can be composed to perform a nontrivial electronic commerce operation.

1. Alice prepares two documents, a purchase order O stating that she intends to order the book from Barney, and a payment slip P, providing Lisa the card number to be used in the transaction, and the amount to be charged. Alice computes digests
$$\begin{aligned} o &= H(O) \\ p &= H(P), \end{aligned}$$
and produces a digital signature S for the digest of the concatenation of o and p, that is,
$$S = D_A(H(o||p)) = D(H(H(O)||H(P))),$$
where D_A is the function used by Alice to sign, based on her private key. Alice encrypts the concatenation of o, P, and S with Lisa's public key, which yields ciphertext
$$C_L = E_L(o||P||S).$$
She also encrypts with Barney's public key the concatenation of O, p, and S, yielding ciphertext
$$C_B = E_B(O||p||S).$$
She sends to Barney C_L and C_B.
2. Barney retrieves O, p, and S by decrypting C_B with his private key. He verifies the authenticity of the purchase order O with Alice's public key by checking that
$$E_A(S) = H(H(O)||p),$$
and forwards C_L to Lisa.
3. Lisa retrieves o, P, and S by decrypting C_L with her private key. She verifies the authenticity of the payment slip P with Alice's public key by checking that
$$E_A(S) = H(o||H(P)),$$
and verifies that P indicates a payment to Barney. She then creates an authorization message M that consists of a transaction number, Alice's name, and the amount she agreed to pay. Lisa computes the signature T of M, and sends the pair (M,T) encrypted with Barney's public key to Barney, that is, $C_M = E_B(M||T)$.
4. Barney retrieves M and T by decrypting C_M and verifies the authenticity of the authorization message M with Lisa's public key, by checking that $E_L(T) = M$. He verifies that the name in M is Alice's, and that the amount is the correct price of the book. He fulfills the order by sending the book to Alice and requests the payment from Lisa by sending her the transaction number encrypted with Lisa's public key.
5. Lisa pays Barney and charges Alice's credit card account.

Algorithm 10.14: Simplified version of the SET protocol.

10.3.5 Key Distribution and Exchange

A public-key cryptosystem assumes that public keys are known to all the parties. For example, if Alice wants to send a confidential message to Bob, she needs to know Bob's public key. Similarly, if Alice wants to verify Bob's digital signature, she needs Bob's public key as well. How does Alice get Bob's public key? Bob can just send it to Alice, but if Eve can intercept the communication between Bob and Alice, she could replace Bob's public key with her key, and thus trick Alice into revealing her message meant for Bob, or believing that a message was signed by Bob, while it was signed instead by Eve.

Digital Certificates

A solution to the problem requires the introduction of a third party, Charlie, who is trusted by all the participants in a protocol. It is further assumed that each participant has Charlie's public key. Charlie issues each participant a ***certificate***, which is a statement digitally signed by Charlie that contains the name of the participant and its public key. Bob now sends Alice the certificate issued to him by Charlie. Alice extracts Bob's public key from the certificate and verifies its authenticity by using Charlie's public key (recall the assumption that each participant has Charlie's public key). Certificates are widely used in practical public-key applications. Their format is described in the ***X.509*** ITU (International Telecommunication Union) standard. In addition to the subject of the certificate and its public key, a certificate also contains a unique serial number and an expiration date. An issuer of certificates is called a ***certificate authority*** (CA).

In a realistic setting, the protocols described in the previous section should be modified by introducing certificates to distribute public keys. Also, the certificates should be validated using the CA's public key.

Certificate Revocation

Private keys are sometimes lost, stolen, or otherwise compromised. When this happens, the CA should revoke the certificate for that key. For example, Bob may have kept his private key in a file on his laptop computer. If the laptop is stolen, Bob should request the CA to immediately revoke his certificate, since otherwise the thief could impersonate Bob. The CA periodically publishes a ***certificate revocation list*** (CRL), which consists of the signed list of the serial numbers of all the unexpired certificates that have been revoked together with a timestamp.

When validating a certificate, a participant should also get the latest CRL from the CA, and verify that that the certificate has not been revoked. The age of the CRL (difference between the current time and the timestamp) provides a measure of risk for the participant examining a certificate. Alternately, there are several online schemes for checking the validity of a given digital certificate using a networked server that stores revocation information.

Using Public Keys for Symmetric Key Exchange

Public key cryptography overcomes the critical bottleneck of symmetric key cryptosystems, since, in a public key cryptosystem, there is no need to distribute secret keys before engaging in a secure communication. Unfortunately, this advantage comes at a cost, as the existing public-key cryptosystems take much longer to encrypt and decrypt than existing symmetric cryptosystems. Thus, in practice, public-key cryptosystems are often used in conjunction with symmetric cryptosystems, to overcome the challenge of exchanging secret keys in order to set up a symmetric cryptographic communication channel.

For example, if Alice wants to set up a secure communication channel with Bob, she and Bob can perform the following set of steps.

1. Alice computes a random number x, computes her digital signature S of x, and encrypts the pair (x,S) using Bob's public key, sending the resulting ciphertext C to Bob.
2. Bob decrypts C using his private key, and verifies that Alice sent it to him by checking the signature S.
3. Bob can then show Alice that he has indeed received x, by encrypting x with Alice's public key, and sending it back to her.

From this point on, they can use the number x as the secret key in a symmetric cryptosystem.

Diffie-Hellman Secret Key Exchange

If Alice and Bob are communicating using a medium that is reliable but perhaps not private, there is another scheme they can use to compute a secret key that they can then share for future symmetric encryption communications. This scheme is called ***Diffie-Hellman key exchange***, and consists of the following steps:

1. Alice and Bob agree (publicly) on a large prime n and a generator g in Z_n.
2. Alice chooses a random number x and sends Bob $B = g^x \bmod n$.
3. Bob chooses a random number y and sends Alice $A = g^y \bmod n$.
4. Alice computes $K = A^x \bmod n$.
5. Bob computes $K' = B^y \bmod n$.

Clearly, $K = K'$, so Alice and Bob can now use K (respectively, K') to communicate using a symmetric cryptosystem.

10.4 The Fast Fourier Transform

A common bottleneck computation in many cryptographic systems is the multiplication of large integers and polynomials. The fast Fourier transform is a surprising and efficient algorithm for multiplying such objects. We describe this algorithm first for multiplying polynomials and we then show how this approach can be extended to large integers.

A polynomial represented in ***coefficient form*** is described by a coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ as follows:

$$p(x) = \sum_{i=0}^{n-1} a_i x^i.$$

The ***degree*** of such a polynomial is the largest index of a nonzero coefficient a_i. A coefficient vector of length n can represent polynomials of degree at most $n-1$.

The coefficient representation is natural, in that it is simple and allows for several polynomial operations to be performed quickly. For example, given a second polynomial described using a coefficient vector $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$ as

$$q(x) = \sum_{i=0}^{n-1} b_i x^i,$$

we can easily add $p(x)$ and $q(x)$ component-wise to produce their sum,

$$p(x) + q(x) = \sum_{i=0}^{n-1} (a_i + b_i) x^i.$$

Likewise, the coefficient form for $p(x)$ allows us to evaluate $p(x)$ efficiently, by ***Horner's rule*** (Exercise C-1.16), as

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x a_{n-1}) \cdots)).$$

Thus, with the coefficient representation, we can add and evaluate degree-$(n-1)$ polynomials in $O(n)$ time.

Multiplying two polynomials $p(x)$ and $q(x)$, as defined above in coefficient form, is not straightforward, however. To see the difficulty, consider $p(x)q(x)$:

$$p(x)q(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 + \cdots + a_{n-1} b_{n-1} x^{2n-2}.$$

That is,

$$p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad \text{where} \quad c_i = \sum_{j=0}^{i} a_j b_{i-j}, \ \text{ for } i = 0, 1, \ldots, 2n-2.$$

This equation defines a vector $\mathbf{c} = [c_0, c_1, \ldots, c_{2n-1}]$, which we call the ***convolution*** of the vectors $\mathbf{a}$ and $\mathbf{b}$. For symmetry reasons, we view the convolution as a vector of size $2n$, defining $c_{2n-1} = 0$. We denote the convolution of $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a} * \mathbf{b}$. If we apply the definition of the convolution directly, then it will take us $\Theta(n^2)$ time to multiply the two polynomials p and q.

The ***fast Fourier transform*** (***FFT***) algorithm allows us to perform this multiplication in $O(n \log n)$ time. The improvement of the FFT is based on an interesting observation. Namely, that another way of representing a degree-$(n-1)$ polynomial is by its value on n distinct inputs. Such a representation is unique because of the following theorem.

Theorem 10.24 [The Interpolation Theorem for Polynomials]: *Given a set of n points in the plane, $S = \{(x_0,y_0),(x_1,y_1),(x_2,y_2),\ldots,(x_{n-1},y_{n-1})\}$, such that the x_i's are all distinct, there is a unique degree-$(n-1)$ polynomial $p(x)$ with $p(x_i) = y_i$, for $i = 0,1,\ldots,n-1$.*

Suppose, then, that we can represent a polynomial not by its coefficients, but instead by its value on a collection of different inputs. This theorem suggests an alternative method for multiplying two polynomials p and q. In particular, evaluate p and q for $2n$ different inputs $x_0,x_1,\ldots,x_{2n-1}$ and compute the representation of the product of p and q as the set

$$\{(x_0, p(x_0)q(x_0)), (x_1, p(x_1)q(x_1)), \ldots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\}.$$

Such a computation would clearly take just $O(n)$ time given the $2n$ input-output pairs for each of p and q.

The challenge, then, to effectively using this approach to multiply p and q is to come up quickly with $2n$ input-output pairs for p and q. Applying Horner's rule to $2n$ different inputs would take us $\Theta(n^2)$ time, which is not asymptotically any faster than using the convolution directly. So Horner's rule is of no help here. Of course, we have full freedom in how we choose the set of $2n$ inputs for our polynomials. That is, we have full discretion to choose inputs that are easy to evaluate. For example, $p(0) = a_0$ is a simple case. But we have to choose a set of $2n$ easy inputs to evaluate p on, not just one. Fortunately, the mathematical concept we discuss next provides a convenient set of inputs that are collectively easier to use to evaluate a polynomial than applying Horner's rule $2n$ times.

10.4.1 Primitive Roots of Unity

A number ω is a ***primitive nth root of unity***, for $n \geq 2$, if it satisfies the following properties:

1. $\omega^n = 1$, that is, ω is an nth root of 1.
2. The numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are distinct.

Note that this definition implies that a primitive nth root of unity has a multiplicative inverse, $\omega^{-1} = \omega^{n-1}$, for

$$\omega^{-1}\omega = \omega^{n-1}\omega = \omega^n = 1.$$

Thus, we can speak in a well-defined fashion of negative exponents of ω, as well as positive ones.

The notion of a primitive nth root of unity may, at first, seem like a strange definition with few examples. But it actually has several important instances. One important one is the complex number

$$e^{2\pi i/n} = \cos(2\pi/n) + \mathbf{i}\sin(2\pi/n),$$

which is a primitive nth root of unity, when we take our arithmetic over the complex numbers, where $\mathbf{i} = \sqrt{-1}$.

Primitive nth roots of unity have a number of important properties, including the following three ones.

Lemma 10.25 (Cancellation Property): *If ω is an nth root of unity, then, for any integer $k \neq 0$, with $-n < k < n$,*

$$\sum_{j=0}^{n-1} \omega^{kj} = 0.$$

Proof: Since $\omega^k \neq 1$,

$$\sum_{j=0}^{n-1} \omega^{kj} = \frac{(\omega^k)^n - 1}{\omega^k - 1} = \frac{(\omega^n)^k - 1}{\omega^k - 1} = \frac{1^k - 1}{\omega^k - 1} = \frac{1-1}{\omega^k - 1} = 0$$

■

Lemma 10.26 (Reduction Property): *If ω is a primitive $(2n)$th root of unity, then ω^2 is a primitive nth root of unity.*

Proof: If $1, \omega, \omega^2, \ldots, \omega^{2n-1}$ are distinct, then $1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n-1}$ are also distinct. ■

Lemma 10.27 (Reflective Property): *If ω is a primitive nth root of unity and n is even, then*

$$\omega^{n/2} = -1.$$

Proof: By the cancellation property, for $k = n/2$,

$$\begin{aligned} 0 &= \sum_{j=0}^{n-1} \omega^{(n/2)j} \\ &= \omega^0 + \omega^{n/2} + \omega^n + \omega^{3n/2} + \cdots + \omega^{(n/2)(n-2)} + \omega^{(n/2)(n-1)} \\ &= \omega^0 + \omega^{n/2} + \omega^0 + \omega^{n/2} + \cdots + \omega^0 + \omega^{n/2} \\ &= (n/2)(1 + \omega^{n/2}). \end{aligned}$$

Thus, $0 = 1 + \omega^{n/2}$. ■

An interesting corollary to the reflective property, which motivates its name, is the fact that if ω is a primitive nth root of unity and $n \geq 2$ is even, then

$$\omega^{k+n/2} = -\omega^k.$$

10.4.2 The Discrete Fourier Transform

Let us now return to the problem of evaluating a polynomial defined by a coefficient vector **a** as

$$p(x) = \sum_{i=0}^{n-1} a_i x^i,$$

for a carefully chosen set of input values. The technique we discuss in this section, called the ***Discrete Fourier Transform*** (DFT), is to evaluate $p(x)$ at the nth roots of unity, $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$. Admittedly, this gives us just n input-output pairs, but we can "pad" our coefficient representation for p with 0's by setting $a_i = 0$, for $n \leq i \leq 2n-1$. This padding would let us view p as a degree-$(2n-1)$ polynomial, which would in turn let us use the primitive $(2n)$th roots of unity as inputs for a DFT for p. Thus, if we need more input-output values for p, let us assume that the coefficient vector for p has already been padded with as many 0's as necessary.

Formally, the Discrete Fourier Transform for the polynomial p represented by the coefficient vector **a** is defined as the vector **y** of values

$$y_j = p(\omega^j),$$

where ω is a primitive nth root of unity. That is,

$$y_j = \sum_{i=0}^{n-1} a_i \omega^{ij}.$$

In the language of matrices, we can alternatively think of the vector **y** of y_j values and the vector **a** as column vectors, and say that

$$\mathbf{y} = F\mathbf{a},$$

where F is an $n \times n$ matrix such that $F[i,j] = \omega^{ij}$.

The Inverse Discrete Fourier Transform

Interestingly, the matrix F has an inverse, F^{-1}, so that $F^{-1}(F(\mathbf{a})) = \mathbf{a}$ for all **a**. The matrix F^{-1} allows us to define an ***inverse Discrete Fourier Transform***. If we are given a vector **y** of the values of a degree-$(n-1)$ polynomial p at the nth roots of unity, $\omega^0, \omega^1, \ldots, \omega^{n-1}$, then we can recover a coefficient vector for p by computing

$$\mathbf{a} = F^{-1}\mathbf{y}.$$

Moreover, the matrix F^{-1} has a simple form, in that $F^{-1}[i,j] = \omega^{-ij}/n$. Thus, we can recover the coefficient a_i as

$$a_i = \sum_{j=0}^{n-1} y_j \omega^{-ij}/n.$$

The following lemma justifies this claim, and is the basis of why we refer to F and F^{-1} as "transforms."

Lemma 10.28: *For any vector* $\mathbf{a}$, $F^{-1} \cdot F\mathbf{a} = \mathbf{a}$.

Proof: Let $A = F^{-1} \cdot F$. It is enough to show that $A[i, j] = 1$ if $i = j$, and $A[i, j] = 0$ if $i \neq j$. That is, $A = I$, where I is the ***identity matrix***. By the definitions of F^{-1}, F, and matrix multiplication,

$$A[i, j] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-ik} \omega^{kj}.$$

If $i = j$, then this equation reduces to

$$A[i, i] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^0 = \frac{1}{n} \cdot n = 1.$$

So, consider the case when $i \neq j$, and let $m = j - i$. Then the ijth entry of A can be written as

$$A[i, j] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{mk},$$

where $-n < m < n$ and $m \neq 0$. By the cancellation property for a primitive nth root of unity, the right-hand side of the above equation reduces to 0; hence,

$$A[i, j] = 0,$$

for $i \neq j$. ■

Given the DFT and the inverse DFT, we can now define our approach to multiplying two polynomials p and q.

The Convolution Theorem

To use the discrete Fourier transform and its inverse to compute the convolution of two coefficient vectors, **a** and **b**, we apply the following steps, which we illustrate in a schematic diagram, as shown in Figure 10.15.

1. Pad **a** and **b** each with n 0's and view them as column vectors to define
$$\mathbf{a}' = [a_0, a_1, \ldots, a_{n-1}, 0, 0, \ldots, 0]^T$$
$$\mathbf{b}' = [b_0, b_1, \ldots, b_{n-1}, 0, 0, \ldots, 0]^T.$$
2. Compute the Discrete Fourier Transforms $\mathbf{y} = F\mathbf{a}'$ and $\mathbf{z} = F\mathbf{b}'$.
3. Multiply the vectors **y** and **z** component-wise, defining the simple product $\mathbf{y} \cdot \mathbf{z} = F\mathbf{a}' \cdot F\mathbf{b}'$, where
$$(\mathbf{y} \cdot \mathbf{z})[i] = (F\mathbf{a}' \cdot F\mathbf{b}')[i] = F\mathbf{a}'[i] \cdot F\mathbf{b}'[i] = y_i \cdot z_i,$$
for $i = 1, 2, \ldots, 2n - 1$.
4. Compute the inverse Discrete Fourier Transform of this simple product. That is, compute $\mathbf{c} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.

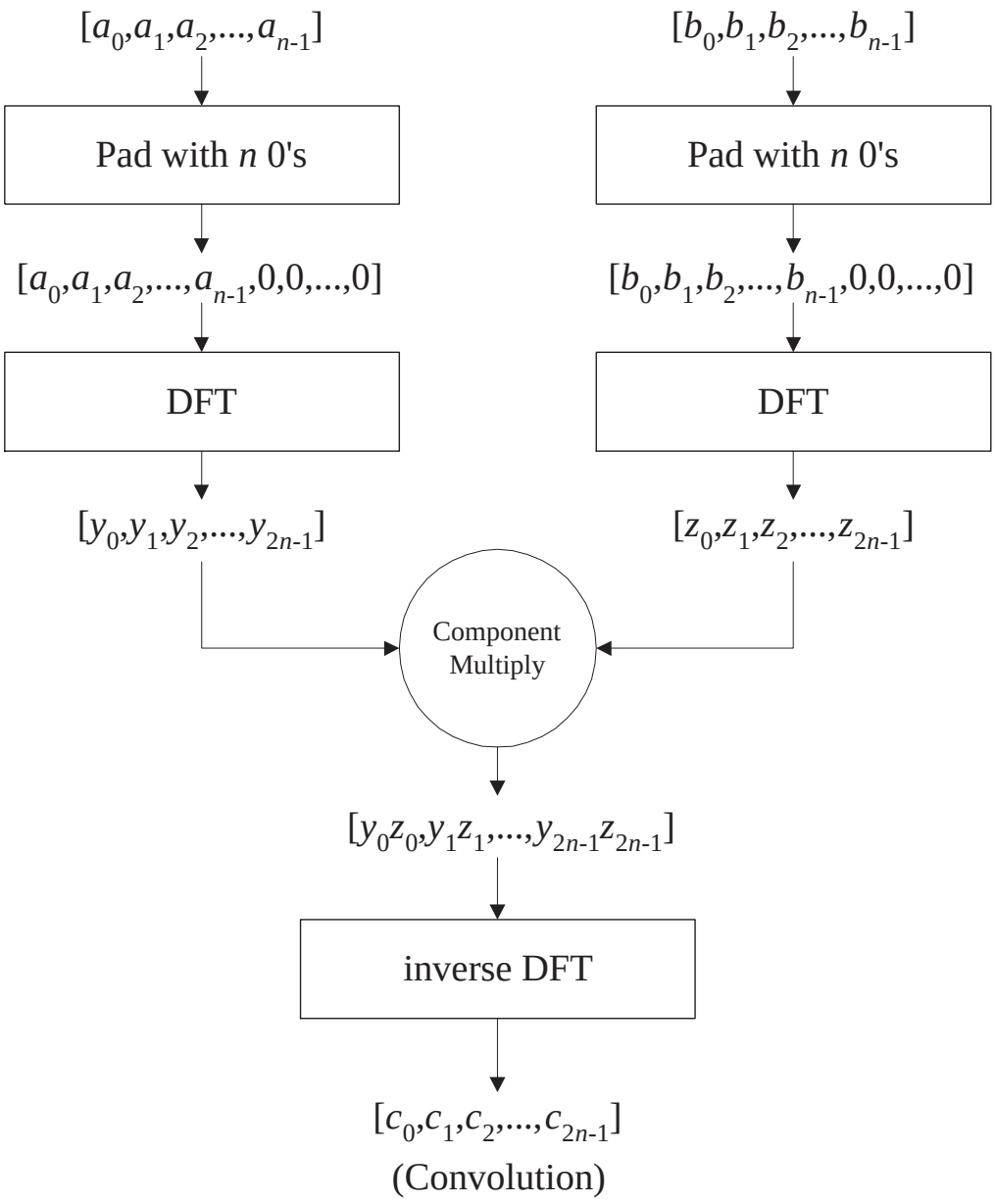


Figure 10.15: An illustration of the Convolution Theorem, to compute $\mathbf{c} = \mathbf{a} * \mathbf{b}$.

The reason the above approach works is because of the following.

Theorem 10.29 [The Convolution Theorem]: *Suppose we are given two n-length vectors $\mathbf{a}$ and $\mathbf{b}$ padded with 0's to $2n$-length vectors $\mathbf{a}'$ and $\mathbf{b}'$, respectively. Then $\mathbf{a} * \mathbf{b} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.*

Proof: We will show that $F(\mathbf{a} * \mathbf{b}) = F\mathbf{a}' \cdot F\mathbf{b}'$. So, consider $A = F\mathbf{a}' \cdot F\mathbf{b}'$. Since the second halves of $\mathbf{a}'$ and $\mathbf{b}'$ are padded with 0's,

$$\begin{aligned} A[i] &= \left(\sum_{j=0}^{n-1} a_j \omega^{ij}\right) \cdot \left(\sum_{k=0}^{n-1} b_k \omega^{ik}\right) \\ &= \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)}, \end{aligned}$$

for $i = 0, 1, \ldots, 2n-1$. Consider, next, $B = F(\mathbf{a} * \mathbf{b})$. By the definition of convolution and the DFT,

$$B[i] = \sum_{l=0}^{2n-1}\sum_{j=0}^{2n-1} a_j b_{l-j} \omega^{il}.$$

Substituting k for $l - j$, and changing the order of the summations, we get

$$B[i] = \sum_{j=0}^{2n-1}\sum_{k=-j}^{2n-1-j} a_j b_k \omega^{i(j+k)}.$$

Since b_k is undefined for $k < 0$, we can start the second summation above at $k = 0$. In addition, since $a_j = 0$ for $j > n-1$, we can lower the upper limit in the first summation above to $n-1$. But once we have made this substitution, note that the upper limit on the second summation above is always at least n. Thus, since $b_k = 0$ for $k > n-1$, we may lower the upper limit on the second summation to $n-1$. Therefore,

$$B[i] = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)},$$

which proves the theorem. ∎

We now have a method for computing the multiplication of two polynomials that involves computing two DFTs, doing a simple linear-time component-wise multiplication, and computing an inverse DFT. Thus, if we can find a fast algorithm for computing the DFT and its inverse, then we will have a fast algorithm for multiplying two polynomials. We describe such a fast algorithm, which is known as the "fast Fourier transform," next.

10.4.3 The Fast Fourier Transform Algorithm

The ***Fast Fourier Transform*** (FFT) Algorithm computes a Discrete Fourier Transform (DFT) of an n-length vector in $O(n \log n)$ time. In the FFT algorithm, we apply the divide-and-conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree-$(n-1)$ polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two degree-$(n/2-1)$ polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

and noting that we can combine these two polynomials into p using the equation

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

The DFT evaluates $p(x)$ at each of the nth roots of unity, $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$. Note that, by the reduction property, the values $(\omega^2)^0, \omega^2, (\omega^2)^2, (\omega^2)^3, \ldots, (\omega^2)^{n-1}$ are $(n/2)$th roots of unity. Thus, we can evaluate each of $p^{\text{even}}(x)$ and $p^{\text{odd}}(x)$ at these values, and we can reuse those same computations in evaluating $p(x)$. This observation is used in Algorithm 10.16 (FFT) to define the, which takes as input an n-length coefficient vector $\mathbf{a}$ and a primitive nth root of unity ω. For the sake of simplicity, we assume that n is a power of two.

Algorithm $\mathsf{FFT}(\mathbf{a}, \omega)$:

Input: An n-length coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ and a primitive nth root of unity ω, where n is a power of 2

Output: A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the nth roots of unity

if $n = 1$ **then**
  return $\mathbf{y} = \mathbf{a}$.
$x \leftarrow \omega^0$ {x will store powers of ω, so initially $x = 1$.}
{Divide Step, which separates even and odd indices}
$\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
$\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
{Recursive Calls, with ω^2 as $(n/2)$th root of unity, by the reduction property}
$\mathbf{y}^{\text{even}} \leftarrow \mathsf{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$
$\mathbf{y}^{\text{odd}} \leftarrow \mathsf{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$
{Combine Step, using $x = \omega^i$}
for $i \leftarrow 0$ to $n/2 - 1$ **do**
  $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
  $y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$ {Uses reflective property}
  $x \leftarrow x \cdot \omega$
return $\mathbf{y}$

Algorithm 10.16: Recursive FFT algorithm.

The Correctness of the FFT Algorithm

The pseudo-code description in Algorithm 10.16 for the FFT algorithm is deceptively simple, so let us say a few words about why it works correctly. First, note that the base case of the recursion, when $n = 1$, correctly returns a vector $\mathbf{y}$ with the one entry, $y_0 = a_0$, which is the leading and only term in the polynomial $p(x)$ in this case.

In the general case, when $n \geq 2$, we separate $\mathbf{a}$ into its even and odd instances, $\mathbf{a}^{\text{even}}$ and $\mathbf{a}^{\text{odd}}$, and recursively call the FFT using ω^2 as the $(n/2)$th root of unity. As we have already mentioned, the reduction property of a primitive nth root of unity, allows us to use ω^2 in this way. Thus, we may inductively assume that

$$\begin{aligned} y_i^{\text{even}} &= p^{\text{even}}(\omega^{2i}) \\ y_i^{\text{odd}} &= p^{\text{odd}}(\omega^{2i}). \end{aligned}$$

Let us therefore consider the for-loop that combines the values from the recursive calls. Note that in the i iteration of the loop, $x = \omega^i$. Thus, when we perform the assignment statement

$y_i \leftarrow y_i^{\text{even}} + x y_i^{\text{odd}}$,

we have just set

$$\begin{aligned} y_i &= p^{\text{even}}((\omega^2)^i) + \omega^i \cdot p^{\text{odd}}((\omega^2)^i) \\ &= p^{\text{even}}((\omega^i)^2) + \omega^i \cdot p^{\text{odd}}((\omega^i)^2) \\ &= p(\omega^i), \end{aligned}$$

and we do this for each index $i = 0, 1, \ldots, n/2 - 1$. Similarly, when we perform the assignment statement

$y_{i+n/2} \leftarrow y_i^{\text{even}} - x y_i^{\text{odd}}$,

we have just set

$$y_{i+n/2} = p^{\text{even}}((\omega^2)^i) - \omega^i \cdot p^{\text{odd}}((\omega^2)^i).$$

Since ω^2 is a primitive $(n/2)$th root of unity, $(\omega^2)^{n/2} = 1$. Moreover, since ω is itself a primitive nth root of unity,

$$\omega^{i+n/2} = -\omega^i,$$

by the reflection property. Thus, we can rewrite the above identity for $y_{i+n/2}$ as

$$\begin{aligned} y_{i+n/2} &= p^{\text{even}}((\omega^2)^{i+(n/2)}) - \omega^i \cdot p^{\text{odd}}((\omega^2)^{i+(n/2)}) \\ &= p^{\text{even}}((\omega^{i+(n/2)})^2) + \omega^{i+n/2} \cdot p^{\text{odd}}((\omega^{i+(n/2)})^2) \\ &= p(\omega^{i+n/2}), \end{aligned}$$

and this will hold for each $i = 0, 1, \ldots, n/2 - 1$. Thus, the vector $\mathbf{y}$ returned by the FFT algorithm will store the values of $p(x)$ at each of the nth roots of unity.

Analyzing the FFT Algorithm

The FFT algorithm follows the divide-and-conquer paradigm, dividing the original problem of size n into two subproblems of size $n/2$, which are solved recursively. We assume that each arithmetic operation performed by by algorithms takes $O(1)$ time. The divide step as well as the combine step for merging the recursive solutions, each take $O(n)$ time. Thus, we can characterize the running time $T(n)$ of the FFT algorithm using the recurrence equation

$$T(n) = 2T(n/2) + bn,$$

for some constant $b > 0$. By the Master Theorem (5.6), $T(n)$ is $O(n \log n)$. Therefore, we can summarize our discussion as follows.

Theorem 10.30: *Given an n-length coefficient vector $\mathbf{a}$ defining a polynomial $p(x)$, and a primitive nth root of unity, ω, the* FFT *algorithm evaluates $p(x)$ at each of the nth roots of unity, ω^i, for $i = 0, 1, \ldots, n-1$, in $O(n \log n)$ time.*

There is also an inverse FFT algorithm, which computes the inverse DFT in $O(n \log n)$ time. The details of this algorithm are similar to those for the FFT algorithm and are left as an exercise (R-10.14). Combining these two algorithms in our approach to multiplying two polynomials $p(x)$ and $q(x)$, given their n-length coefficient vectors, we have an algorithm for computing this product in $O(n \log n)$ time.

By the way, this approach for using the FFT algorithm and its inverse to compute the product of two polynomials can be extended to the problem of computing the product of two large integers. We discuss this method next.

10.4.4 Multiplying Big Integers

Let us revisit the problem studied in Section 5.2.2. Namely, suppose we are given two big integers I and J that use at most N bits each, and we are interested in computing $I \cdot J$. The main idea of the method we describe in this section is to use the FFT algorithm to compute this product. Of course, a major challenge in utilizing the FFT algorithmic design pattern in this way is to define integer arithmetic so that it gives rise to primitive roots of unity (see, for example, Exercise C-10.8).

The algorithm we describe here assumes that we can break I and J up into words of $O(\log N)$ bits each, such that arithmetic on each word can be done using built-in operations in our computer model in constant time. This is a reasonable assumption, since it takes $\lceil \log N \rceil$ bits just to represent the number N itself. The number system we use for the FFT in this case is to perform all arithmetic modulo p, for a suitably chosen prime number p, which itself can be represented in a single word of at most $O(\log N)$ bits. The specific prime modulus p we choose is to find small integers $c \geq 1$ and $n \geq 1$, such that $p = cn + 1$ is prime and $N/\lfloor \log p \rfloor \leq n/2$. For the sake of the simplicity of our description, let us additionally assume that n is a power of two.

In general, we would expect c in the above definition of p to be $O(\log n)$, since a fundamental theorem from number theory states that a random odd number in the range $[1,n]$ is prime with probability $\Omega(1/\log n)$. That is, we expect p to be represented with $O(\log N)$ bits. Given this prime number p, our intent is to perform all arithmetic in the FFT modulo p. Since we are performing this arithmetic on the words of a large vector of integers, we also want the size of each word to be less than half of that used to represent p, so we can represent the product of any two words without having the modulo-p arithmetic "wipe out" any pairwise products of words. For example, the following values of p work well for modern computers:

- If $n \leq 2^{10} = 1024$, then we choose $p = 25 \cdot 2^{10} + 1 = 25\,601$ as our prime modulus, which can be represented using a 15-bit word. Moreover, since $p > 2^{14}$ in this case, such a choice allows us to multiply numbers whose representations are as large as 2^{10} words of 7 bits each.
- If $n \leq 2^{27} = 134\,217\,728$, then we choose $p = 15 \cdot 2^{27} + 1 = 2\,013\,265\,921$ as our prime modulus, which uses a 31-bit word. Moreover, since $p > 2^{30}$ in this case, we can multiply numbers whose representations are as large as 2^{27} words of 15 bits each, or 240MB.

Given a prime $p = cn + 1$, for reasonably small $c \geq 1$, and n defined so that $N/\lfloor \log p \rfloor$ is $O(n)$, we define $m = \lfloor (\log p)/2 \rfloor$. We view I and J respectively as being vectors **a** and **b** of words that use m bits each, extended with as many 0's as needed to make **a** and **b** both have length n, with at least the last half of their higher-order terms being 0. We can write I and J as

$$\begin{aligned} I &= \sum_{i=0}^{n-1} a_i 2^{mi} \\ J &= \sum_{i=0}^{n-1} b_i 2^{mi}. \end{aligned}$$

Moreover, we choose n so that the first $n/2$ of the a_i's and b_i's are nonzero at most. Thus, we can represent the product $K = I \cdot J$, as

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}.$$

Once we have a prime number $p = cn+1$, for a reasonably small integer $c \geq 1$, we find an integer x that is a generator of the group Z_p^* (see Section 10.1). That is, we find x such that $x^i \bmod p$ is different for $i = 0, 1, \ldots, p-1$. Given such a generator, x, then we can use $\omega = x^c \bmod p$ as a primitive nth root of unity (assuming all multiplication and addition is done modulo p). That is, each of $(x^c)^i \bmod p$ are distinct for $i = 0, 1, 2, \ldots, n-1$, but, by Fermat's Little Theorem (Theorem 10.8),

$$\begin{aligned} (x^c)^n \bmod p &= x^{cn} \bmod p \\ &= x^{p-1} \bmod p \\ &= 1. \end{aligned}$$

In order to compute the product of the big N-bit integers I and J, recall that view I and J, respectively, as extended n-length vectors **a** and **b** of words having m bits each (with at least the $n/2$ higher-order words being all 0's). Note that, since $m = \lfloor \log p/2 \rfloor$, we have that $2^m < p$. Thus, each of the terms in **a** and **b** is already reduced modulo p without us having to do any additional work, and any product of two of these terms is also reduced modulo p.

To compute $K = I \cdot J$, we then apply the Convolution Theorem, using the FFT and inverse FFT algorithms, to compute the convolution **c** of **a** and **b**. In this computation we use ω, as defined above, as the primitive nth root of unity, and we perform all the internal arithmetic (including the component-wise products of the transformed versions of **a** and **b**) modulo p.

The terms in the convolution **c** are all less than p, but in order to build a representative of $K = I \cdot J$, we actually need each term to be represented using exactly m bits. Thus, after we have the convolution **c**, we must compute the product K as

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}.$$

This final computation is not as difficult as it looks, however, since multiplying by a power of two in binary is just a shifting operation. Moreover, p is $O(2^{m+1})$, so the above summation just involves propagating some groups of carry bits from one term to the next. Thus, this final construction of a binary representation of K as a vector of words of m bits each can be done in $O(n)$ time. Since applying the Convolution Theorem as described above takes $O(n \log n)$ time, this gives us the following.

Theorem 10.31: *Given two N-bit integers I and J, we can compute the product $K = I \cdot J$ in $O(N)$ time, assuming that arithmetic involving words of size $O(\log N)$ can be done in constant time.*

Proof: The number n is chosen so that it is $O(N/\log N)$. Thus a running time of $O(n \log n)$ is $O(N)$, assuming that arithmetic operations involving words of size $O(\log N)$ can be done in constant time. ∎

In some cases, we cannot assume that arithmetic involving $O(\log n)$-size words can be done in constant time, but instead, we must pay constant time for every bit operation. In this model it is still possible to use the FFT to multiply two N-bit integers, but the details are somewhat more complicated and the running time increases to $O(N \log N \log\log N)$.

In Section 10.5, we study some important implementation issues related to the FFT algorithm, including an experimental analysis of how well it performs in practice.

10.5 Java Example: FFT

There are a number of interesting implementation issues related to the FFT algorithm, which we discuss in this section. We begin with the description of a big integer class that performs multiplication using the recursive version of the FFT, as described above in pseudo-code in Algorithm 10.16. We show the declaration of this class and its important instance variables and constants in Code Fragment 10.17. A significant detail in these declarations includes our defining the primitive nth root of unity, OMEGA. Our choice of 31^{15} derives from the fact that 31 is a generator for Z_p^* for the prime number $15 \cdot 2^{27} + 1$. We know 31 is a generator for this Z_p^*, because of a theorem from number theory. This theorem states that an integer x is a generator of Z_p^* if and only if $x^{\phi(p)/q} \bmod p$ is not equal to 1 for any prime divisor q of $\phi(p)$, where $\phi(p)$ is the Euler totient function defined in Section 10.1. For our particular p, $\phi(p) = 15 \cdot 2^{27}$; hence, the only prime divisors q we need to consider are 2, 3, and 5.

```
import java.lang.*;
import java.math.*;
import java.util.*;

public class BigInt {
  protected int signum=0;              // neg = -1, 0 = 0, pos = 1
  protected int[] mag;                 // magnitude in little-endian format
  public final static int MAXN=134217728;  // Maximum value for n
  public final static int ENTRYSIZE=15;    // Bits per entry in mag
  protected final static long P=2013265921; // The prime 15*2^{27}+1
  protected final static int OMEGA=440564289; // Root of unity 31^{15} mod P
  protected final static int TWOINV=1006632961; // 2^{-1} mod P
```

Code Fragment 10.17: Declarations and instance variables for a big integer class that supports multiplication using the FFT algorithm.

A Recursive FFT Implementation

We show the multiplication method of our big integer class, BigInt, in Code Fragment 10.18, and, in Code Fragment 10.19, we show our implementation of the recursive FFT algorithm. Note that we use a variable, prod, to store the product that is used in the subsequent two expressions. By factoring out this common subexpression we can avoid performing this repeated computation twice. In addition, note that all modular arithmetic is performed as **long** operations. This requirement is due to the fact that P uses 31 bits, so that we may need to perform additions and subtractions that would overflow a standard-size integer. Instead, we perform all arithmetic as **long** operations and then store the result back to an **int**.

```
public BigInt multiply(BigInt val) {
  int n = makePowerOfTwo(Math.max(mag.length,val.mag.length))*2;
  int signResult = signum * val.signum;
  int[] A = padWithZeros(mag,n);   // copies mag into A padded w/ 0's
  int[] B = padWithZeros(val.mag,n); // copies val.mag into B padded w/ 0's
  int[] root = rootsOfUnity(n);      // creates all n roots of unity
  int[] C = new int[n];              // result array for A*B
  int[] AF = new int[n];             // result array for FFT of A
  int[] BF = new int[n];             // result array for FFT of B
  FFT(A,root,n,0,AF);
  FFT(B,root,n,0,BF);
  for (int i=0; i<n; i++)
    AF[i] = (int)(((long)AF[i]*(long)BF[i]) % P);   // Component multiply
  reverseRoots(root);                // Reverse roots to create inverse roots
  inverseFFT(AF,root,n,0,C);         // Leaves inverse FFT result in C
  propagateCarries(C);               // Convert C to right no. bits per entry
  return new BigInt(signResult,C);
  }
```

Code Fragment 10.18: The multiplication method for a big integer class that supports multiplication using a recursive FFT algorithm.

```
public static void FFT(int[] A, int[] root, int n, int base, int[] Y) {
  int prod;
  if (n==1) {
    Y[base] = A[base];
    return;
    }
  inverseShuffle(A,n,base);    // inverse shuffle to separate evens and odds
  FFT(A,root,n/2,base,Y);      // results in Y[base] to Y[base+n/2-1]
  FFT(A,root,n/2,base+n/2,Y); // results in Y[base+n/2] to Y[base+n-1]
  int j = A.length/n;
  for (int i=0; i<n/2; i++) {
    prod = (int)(((long)root[i*j]*Y[base+n/2+i]) % P);
    Y[base+n/2+i] = (int)(((long)Y[base+i] + P - prod) % P);
    Y[base+i] = (int)(((long)Y[base+i] + prod) % P);
    }
}
public static void inverseFFT(int[] A, int[] root, int n, int base, int[] Y) {
  int inverseN = modInverse(n);      // n^{-1}
  FFT(A,root,n,base,Y);
  for (int i=0; i<n; i++)
    Y[i] = (int)(((long)Y[i]*inverseN) % P);
}
```

Code Fragment 10.19: A recursive implementation of the FFT algorithm.

Avoiding Repeated Array Allocation

The pseudo-code for the recursive FFT algorithm calls for the allocation of several new arrays, including $\mathbf{a}^{\text{even}}$, $\mathbf{a}^{\text{odd}}$, $\mathbf{y}^{\text{even}}$, $\mathbf{y}^{\text{odd}}$, and $\mathbf{y}$. Allocating all of these arrays with each recursive call could prove to be a costly amount of extra work. If it can be avoided, saving this additional allocation of arrays could significantly improve the constant factors in the $O(n \log n)$ time (or $O(N)$ time) performance of the FFT algorithm.

Fortunately, the structure of FFT allows us to avoid this repeated array allocation. Instead of allocating many arrays, we can use a single array, A, for the input coefficients and use a single array, Y, for the answers. The main idea that allows for this usage is that we can think of the arrays A and Y as partitioned into subarrays, each one associated with a different recursive call. We can identify these subarrays using just two variables, base, which identifies the base address of the subarray, and n, which identifies the size of the subarray. Thus, we can avoid the overhead associated with allocating lots of small arrays with each recursive call.

The Inverse Shuffle

Having decided that we will not allocate new arrays during the FFT recursive calls, we must deal with the fact that the FFT algorithm involves performing separate computations on even and odd indices of the input array. In the pseudo-code of Algorithm 10.16, we use new arrays $\mathbf{a}^{\text{even}}$ and $\mathbf{a}^{\text{odd}}$, but now we must use subarrays in A for these vectors. Our solution for this memory management problem is to take the current n-cell subarray in A, and divide it into two subarrays of size $n/2$. One of the subarrays will have the same base as A, while the other has base $\mathsf{base}+n/2$. We move the elements at even indices in A to the lower half and we move elements at odd indices in A to the upper half. In doing so, we define an interesting permutation known as the ***inverse shuffle***. This permutation gets its name from its resemblance to the inverse of the permutation we would get by cutting the array A in half and shuffling it perfectly as if it were a deck of cards. (See Figure 10.20.)

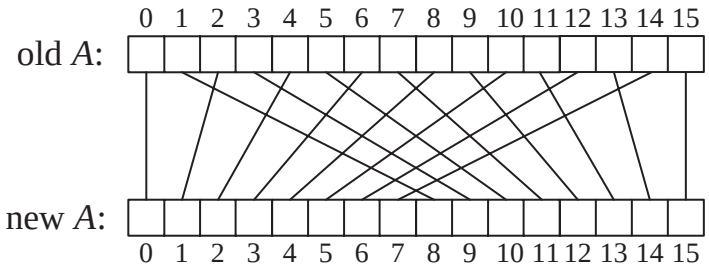


Figure 10.20: An illustration of the inverse shuffle permutation.

Precomputing Roots of Unity and Other Optimizations

There are a number of additional optimizations that we utilized in our FFT implementation. In Code Fragment 10.21, we show some of the important support methods for our implementation, including the computation of the inverse n^{-1}, the inverse shuffle permutation, the precomputation of all the nth roots of unity, propagation of the carries after the convolution has been computed modulo p.

```
protected static int modInverse(int n) { // assumes n is power of two
  int result = 1;
  for (long twoPower = 1; twoPower < n; twoPower *= 2)
    result = (int)(((long)result*TWOINV) % P);
  return result;
}
protected static void inverseShuffle(int[] A, int n, int base) {
  int shift;
  int[] sp = new int[n];
  for (int i=0; i<n/2; i++) { // Unshuffle A into the scratch space
    shift = base + 2*i;
    sp[i] = A[shift];          // an even index
    sp[i+n/2] = A[shift+1];  // an odd index
    }
  for (int i=0; i<n; i++)
    A[base+i] = sp[i];         // copy back to A
}
protected static int[] rootsOfUnity(int n) { //assumes n is power of 2
  int t = MAXN;
  int nthroot = OMEGA;
  for (int t = MAXN; t>n; t /= 2)  // Find prim. nth root of unity
    nthroot = (int)(((long)nthroot*nthroot) % P);
  int[] roots = new int[n];
  int r = 1;           // r will run through all nth roots of unity
  for (int i=0; i<n; i++) {
    roots[i] = r;
    r = (int)(((long)r*nthroot) % P);
    }
  return roots;
}
protected static void propagateCarries(int[] A) {
  int i, carry;
  carry = 0;
  for (i=0; i<A.length; i++) {
    A[i] = A[i] + carry;
    carry = A[i] >>> ENTRYSIZE;
    A[i] = A[i] - (carry << ENTRYSIZE);
    }
  }
```

Code Fragment 10.21: Support methods for a recursive FFT.

An Iterative FFT Implementation

There are additional time improvements that can be made in the FFT algorithm, which involve replacing the recursive version of the algorithm with an iterative version. Our iterative FFT is a part of an alternate big integer class, called FastInt. The multiplication method for this class is shown in Code Fragment 10.22.

```
public FastInt multiply(FastInt val) {
  int n = makePowerOfTwo(Math.max(mag.length,val.mag.length))*2;
  logN = logBaseTwo(n);              // Log of n base 2
  reverse = reverseArray(n,logN);    // initialize reversal lookup table
  int signResult = signum * val.signum;
  int[] A = padWithZeros(mag,n);   // copies mag into A padded w/ 0's
  int[] B = padWithZeros(val.mag,n); // copies val.mag into B padded w/ 0's
  int[] root = rootsOfUnity(n);      // creates all n roots of unity
  FFT(A,root,n);                     // Leaves FFT result in A
  FFT(B,root,n);                     // Leaves FFT result in B
  for (int i=0; i<n; i++)
    A[i] = (int) (((long)A[i]*B[i]) % P);     // Component-wise multiply
  reverseRoots(root);                // Reverse roots to create inverse roots
  inverseFFT(A,root,n);              // Leaves inverse FFT result in A
  propagateCarries(A);               // Convert A to right no. of bits/entry
  return new FastInt(signResult,A);
}
```

Code Fragment 10.22: The multiplication method for an iterative FFT.

Computing the FFT in Place

From Code Fragment 10.22, we already can see a difference between this and the multiply method for the recursive version of the FFT. Namely, we are now performing the FFT in-place. That is, the array A is being used for both the input and output values. This saves us the extra work of copying between output and input arrays. In addition, we compute the logarithm of n, base two, and store this in a static variable, as this logarithm is used repeatedly in calls to the iterative FFT algorithm.

Avoiding Recursion

The main challenge in avoiding recursion in an in-place version of the FFT algorithm is that we have to figure out a way of performing all the inverse shuffles in the input array A. Rather than performing each inverse shuffle with each iteration, we instead perform all the inverse shuffles in advance, assuming that n, the size of the input array, is a power of two.

In order to figure out the net effect of the permutation we would get by repeated and recursive inverse shuffle operations, let us consider how the inverse shuffles move data around with each recursive call. In the first recursive call, of course, we perform an inverse shuffle on the entire array A. Note how this permutation

operates at the bit level of the indices in A. It brings all elements at addresses that have a 0 as their least significant bit to the bottom half of A. Likewise, it brings all elements at addresses that have a 1 as their least significant bit to the top half of A. That is, if an element starts out at an address with b as its least significant bit, then it ends up at an address with b as its most significant bit. The least significant bit in an address is the determiner of which half of A an element winds up in. In the next level of recursion, we repeat the inverse shuffle on each half of A. Viewed again at the bit level, for $b = 0,1$, these recursive inverse shuffles take elements originally at addresses with b as their second least significant bit, and move them to addresses that have b as their second most significant bit. Likewise, for $b = 0,1$, the ith levels of recursion move elements originally at address with b as their ith least significant bit, to addresses with b as their ith most significant bit. Thus, if an element starts out at an address with binary representation $[b_{l-1} \ldots b_2 b_1 b_0]$, then it ends up at an address with binary representation $[b_0 b_1 b_2 \ldots b_{l-1}]$, where $l = \log_2 n$. That is, we can perform all the inverse shuffles in advance just by moving elements in A to the address that is the bit reversal of their starting address in A. To perform this permutation, we build a permutation array, reverse, in the multiply method, and then use this in the bitReversal method called inside the FFT method to permute the elements in the input array A according to this permutation. The resulting iterative version of the FFT algorithm is shown in Code Fragment 10.23.

```
public static void FFT(int[] A, int[] root, int n) {
  int prod,term,index;          // Values for common subexpressions
  int subSize = 1;              // Subproblem size
  bitReverse(A,logN);           // Permute A by bit reversal table
  for (int lev=1; lev<=logN; lev++) {
    subSize *= 2;               // Double the subproblem size.
    for (int base=0; base<n-1; base += subSize) { // Iterate subproblems
      int j = subSize/2;
      int rootIndex = A.length/subSize;
      for (int i=0; i<j; i++) {
        index = base + i;
        prod = (int) (((long)root[i*rootIndex]*A[index+j]) % P);
        term = A[index];
        A[index+j] = (int) (((long)term + P - prod) % P);
        A[index] = (int) (((long)term + prod) % P);
        }
      }
    }
}
public static void inverseFFT(int[] A, int[] root, int n) {
  int inverseN = modInverse(n);      // n^{-1}
  FFT(A,root,n);
  for (int i=0; i<n; i++)
    A[i] = (int) (((long)A[i]*inverseN) % P);
}
```

Code Fragment 10.23: An iterative implementation of the FFT algorithm.

```
protected static void bitReverse(int[] A, int logN) {
  int[] temp = new int[A.length];
  for (int i=0; i<A.length; i++)
    temp[reverse[i]] = A[i];
  for (int i=0; i<A.length; i++)
    A[i] = temp[i];
}
protected static int[] reverseArray(int n, int logN) {
  int[] result = new int[n];
  for (int i=0; i<n; i++)
    result[i] = reverse(i,logN);
  return result;
}
protected static int reverse(int N, int logN) {
  int bit=0;
  int result=0;
  for (int i=0; i<logN; i++) {
    bit = N & 1;
    result = (result << 1) + bit;
    N = N >>> 1;
    }
  return result;
  }
```

Code Fragment 10.24: Support methods for an iterative implementation of the FFT algorithm. Other support methods are like those in the recursive implementation.

We show the additional support methods used by the iterative FFT algorithm in Code Fragment 10.24. All of these support methods deal with the operation of computing the reverse permutation table and then using it to perform the bit-reversal permutation on A.

Experimental Results

Of course, the goal of using the FFT algorithm for big integer multiplication is to perform this operation in $O(n \log n)$ time, on integers represented as n-word vectors, as opposed to the standard $O(n^2)$ multiplication algorithm taught in grade schools. In addition, we designed an iterative version of the FFT algorithm with the goal of improving the constant factors in the running time of this multiplication method. To test these goals in practice, we designed a simple experiment.

In this experiment, we randomly generated ten big integers that consisted of 2^s words of 15 bits each, for $s = 7, 8, \ldots, 16$, and we multiplied them in consecutive pairs, producing nine products for each value of s. We timed the execution time for performing these products, and compared these times to those for multiplying integers represented with the same number of bits in a standard implementation of the Java BigInteger class. The results of this experiment are shown in Figure 10.25. The running times are shown in milliseconds. The experiment was performed in

the Sun Java virtual machine for JDK 1.2 on a Sun Ultra5 with a 360MHz processor and 128MB of memory.

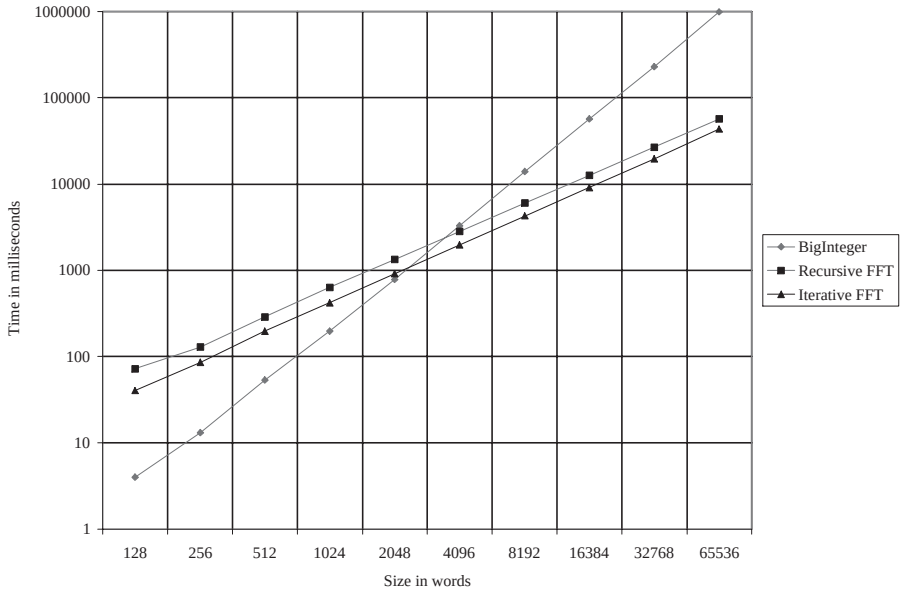


Figure 10.25: Running times for integer multiplication. The chart shows running times for multiplying integers made up of various numbers of words of 15 bits each (for the FFT), or the equivalent size for a standard BigInteger implementation. Note that the scales are logarithmic for both the *x*- and *y*-axes.

Note that we display the results of the experiment on a log-log scale. This choice corresponds to the power test (Section 1.6.2) and allows us to see the relative costs of the different implementations much clearer than we would see on a standard linear scale. Since $\log n^c = c \log n$, the slope of a line on a log-log scale correlates exactly with the exponent of a polynomial in a standard linear scale. Likewise the height of a line on a log-log scale corresponds to a constant of proportionality. The chart in Figure 10.25 displays time using base-ten on the *y*-axis and base-two on the *x*-axis. Using the fact that $\log_2 10$ is approximately 3.322, we indeed see that the running time of the standard multiplication algorithm is $\Theta(n^2)$, whereas the running time of the FFT algorithms is close to linear. Likewise, we note that the constant factor in the iterative FFT algorithm implementation is about 70% of the constant for the recursive FFT algorithm implementation. Also note the significant trade-off that exists between the FFT-based methods and the standard multiplication algorithm. At the small end of the scale, the FFT-based methods are ten times slower than the standard algorithm, but at the high end of the scale they are more than ten times faster!

10.6 Exercises

Reinforcement

R-10.1 Prove Theorem 10.1.

R-10.2 Show the execution of method EuclidGCD$(14300, 5915)$ by constructing a table similar to Table 10.2.

R-10.3 Write a nonrecursive version of Algorithm EuclidGCD.

R-10.4 Show the execution of method EuclidBinaryGCD$(14300, 5915)$ by constructing a table similar to Table 10.2.

R-10.5 Show the existence of additive inverses in Z_p, that is, prove that for each $x \in Z_p$, there is a $y \in Z_p$, such that $x + y \bmod p = 0$.

R-10.6 Construct the multiplication table of the elements of Z_{11}, where the element in row i and column j ($0 \leq i, j \leq 10$) is given by $i \cdot j \bmod 11$.

R-10.7 Prove Corollary 10.7.

R-10.8 Give an alternative proof of Theorem 10.6 and Corollary 10.7 that does not use Theorem 10.3.

R-10.9 Show the execution of method FastExponentiation$(5, 12, 13)$ by constructing a table similar to Table 10.8.

R-10.10 Write a nonrecursive version of Algorithm ExtendedEuclidGCD.

R-10.11 Extend Table 10.10 with two rows giving the values of ia and jb at each step of the algorithm and verify that $ia + jb = 1$.

R-10.12 Show the execution of method ExtendedEuclidGCD$(412, 113)$ by constructing a table similar to Table 10.10.

R-10.13 Compute the multiplicative inverses of the numbers 113, 114, and 127 in Z_{299}.

R-10.14 Describe the inverse FFT algorithm, which computes the inverse DFT in $O(n \log n)$ time. That is, show how to reverse the roles of $\mathbf{a}$ and $\mathbf{y}$ and change the assignments so that, for each output index, we have

$$a_i = \frac{1}{n} \sum_{j=1}^{n-1} y_j \omega^{-ij}.$$

R-10.15 Prove a more general form of the reduction property of primitive roots of unity. Namely, show that, for any integer $c > 0$, if ω is a primitive (cn)th root of unity, then ω^c is a primitive nth root of unity.

R-10.16 Write in the form $a + b\mathbf{i}$ the complex nth roots of unity for $n = 4$ and $n = 8$.

R-10.17 What is the bit-reversal permutation, reverse, for $n = 16$?

R-10.18 Use the FFT and inverse FFT to compute the convolution of $\mathbf{a} = [1, 2, 3, 4]$ and $\mathbf{b} = [4, 3, 2, 1]$. Show the output of each component as in Figure 10.15.

R-10.19 Use the convolution theorem to compute the product of the polynomials $p(x) = 3x^2 + 4x + 2$ and $q(x) = 2x^3 + 3x^2 + 5x + 3$.

R-10.20 Compute the discrete Fourier transform of the vector $[5,4,3,2]$ using arithmetic modulo $17 = 2^4 + 1$. Use the fact that 5 is a generator of Z_{17}^*.

R-10.21 Construct a table showing an example of the RSA cryptosystem with parameters $p = 17$, $q = 19$, and $e = 5$. The table should have two rows, one for the plaintext M and the other for the ciphertext C. The columns should correspond to integer values in the range $[10,20]$ for M.

Creativity

C-10.1 Justify the correctness of the binary Euclid's algorithm (Algorithm 10.3) and analyze its running time.

C-10.2 Let p be an odd prime.

a. Show that Z_p has exactly $(p-1)/2$ quadratic residues.
b. Show that
$$\left(\frac{a}{b}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$
c. Give a randomized algorithm for finding a quadratic residue of Z_p in expected $O(1)$ time.
d. Discuss the relationship between quadratic residues and the quadratic probing technique for collision resolution in hash tables (see Section 2.5.5 and Exercise C-2.35).

C-10.3 Let p be a prime. Give an efficient alternative algorithm for computing the multiplicative inverse of an element of Z_p that is not based on the extended Euclid's algorithm. What is the running time of your algorithm?

C-10.4 Show how to modify Algorithm ExtendedEuclidGCD to compute the multiplicative inverse of an element in Z_n using arithmetic operations on operands with at most $2\lceil \log_2 n \rceil$ bits.

C-10.5 Prove the correctness of method Jacobi(a,b) (Algorithm 10.12) for computing the Jacobi symbol. Also, show that this method executes $O(\log \max(a,b)$ arithmetic operations.

C-10.6 Give a pseudo-code description of the compositeness witness function of the Rabin-Miller algorithm.

C-10.7 Describe a divide-and-conquer algorithm, not based on the FFT, for multiplying two degree-n polynomials with integer coefficients in $O(n^{\log_2 3})$ time, assuming that elementary arithmetic operations on any two integers run in constant time.

C-10.8 Prove that $\omega = 2^{4b/m}$ is a primitive mth root of unity when multiplication is taken modulo $(2^{2b}+1)$, for any integer $b > 0$.

C-10.9 Given degree-n polynomials $p(x)$ and $q(x)$, describe an $O(n \log n)$-time method for multiplying the derivative of $p(x)$ by the derivative of $q(x)$.

C-10.10 Describe a version of the FFT that works when n is a power of 3 by dividing the input vector into three subvectors, recursing on each one, and then merging the subproblem solutions. Derive a recurrence equation for the running time of this algorithm and solve this recurrence using the Master Theorem.

C-10.11 Suppose you are given a set of real numbers $X = \{x_0, x_1, \ldots, x_{n-1}\}$. Note that, by the Interpolation Theorem for Polynomials, there is a unique degree-$(n-1)$ polynomial $p(x)$, such that $p(x_i) = 0$ for $i = 0, 1, \ldots, n-1$. Design a divide-and-conquer algorithm that can construct a coefficient representation of this $p(x)$ in $O(n \log^2 n)$ time.

Projects

P-10.1 Write a class that contains methods for modular exponentiation and computing modular inverses.

P-10.2 Implement the randomized primality testing algorithms by Rabin-Miller and by Solovay-Strassen. Test the results of these algorithms on randomly generated 32-bit integers using the confidence parameters 7, 10, and 20, respectively.

P-10.3 Implement a simplified RSA cryptosystem for integer messages with a Java class that provides methods for encrypting, decrypting, signing, and verifying a signature.

P-10.4 Implement a simplified El Gamal cryptosystem for integer messages with a Java class that provides methods for encrypting, decrypting, signing, and verifying a signature.

Chapter Notes

An introduction to number theory is provided in books by Koblitz [123] and Kranakis [125]. The classic textbook on numerical algorithms is the second volume of Knuth's series on ***The Art of Computer Programming*** [121]. Algorithms for number theoretic problems are also presented in the books by Bressoud and Wagon [40] and by Bach and Shallit [20]. The Solovay-Strassen randomized primality testing algorithm appears in [190, 191]. The Rabin-Miller algorithm is presented in [171].

The book by Schneier [180] describes cryptographic protocols and algorithms in detail. Applications of cryptography to network security are covered in the book by Stallings [192]. The RSA cryptosystem is named after the initials of its inventors, Rivest, Shamir, and Adleman [173]. The El Gamal cryptosystem is named after its inventor [75]. The hash tree structure was introduced by Merkle [153, 154]. The one-way ***accumulator*** function is introduced in [27, 179].

The Fast Fourier Transform (FFT) appears in a paper by Cooley and Tukey [54]. It is also discussed in books by Aho, Hopcroft, and Ullman [7], Baase [19], Cormen, Leiserson, and Rivest [55], Sedgewick [182, 183], and Yap [213], all of which were influential in the discussion given above. In particular, the fast integer multiplication algorithm implementation given above is fashioned after the QuickMul algorithm of Yap and Li. For information on additional applications of the FFT, the interested reader is referred to books by Brigham [41] and Elliott and Rao [66], and the chapter by Emiris and Pan [67]. Sections 10.1 and 10.2 are based in part on a section of an unpublished manuscript of Achter and Tamassia [1].