CHAPTER 9

ALGEBRAIC AND NUMERICAL ALGORITHMS

One plus one is two.
Two plus two is four.
Four plus four is eight.
Eight plus eight is more than ten.

A child's poem

9.1 Introduction

Whenever we perform an arithmetic operation, we are in fact executing an algorithm. We are usually so familiar with these operations that we take the corresponding algorithms for granted. However, whether it is multiplication, division, or a more complicated arithmetic operation, the straightforward algorithm is not always the best when very large numbers or large sequences of numbers are involved. The same phenomenon that we have seen in the previous chapters occurs here as well: Some algorithms that are good for small input become inefficient when the size of the input grows.

As we have done in previous chapters, we will measure the complexity of an algorithm by the number of "operations" that the algorithm executes. For the most part we will assume that basic arithmetic operations (such as addition, multiplication, and division) take one unit of time. This is a reasonable assumption when the operands can be represented by one or two computer words (e.g., integers that are not too large, single-precision or double-precision real numbers). There are cases, however, when the operands are huge (e.g., 2000 digit integers). In such cases, we have to take into account

the size of the operands, or at least to be aware that the basic operations are not simple. It is possible to design algorithms that look very efficient "on paper," but are in fact very inefficient, because the sizes of the operands are ignored.

The meaning of the "size of the input" is confusing sometimes. Given an integer n on which we want to perform an arithmetic operation, it is natural to think of the value n as the size of the input. However, this is contrary to our usual convention of using the storage requirements of the input for defining its size. The distinction is very important. Adding two 100-digit numbers can be done quickly, even by hand. On the other hand, counting to a value represented by a 100-digit number cannot be done in reasonable time even by the fastest computer. Since a number n can be represented by $\lceil \log_2 n \rceil$ bits, its **size** is defined as $\lceil \log_2 n \rceil$. For example, an algorithm that requires $O(\log n)$ operations when n is the input (for example, an algorithm for computing $2n$) is considered linear, since $O(\log n)$ is a linear function of the size of the input, whereas an algorithm that requires $O(\sqrt{n})$ operations when n is the input (for example, factoring n by trying all numbers less than or equal to $\sqrt{n}$) is considered exponential.

As usual, we concentrate in this chapter on interesting techniques for designing algorithms. We first discuss how to compute powers of a given number. We then present what is probably the oldest known nontrivial algorithm: Euclid's algorithm for finding the greatest common divisor. It is quite amazing that modern computers use a 2200-year old algorithm. We then discuss algorithms for polynomial multiplication and matrix multiplication, and we end the chapter with one of the most important and most beautiful algorithms — the fast Fourier transform.

9.2 Exponentiation

We start with a basic arithmetic operation.

> **The Problem** Given two positive integers n and k, compute n^k.

We can easily reduce the problem to that of computing n^{k-1}, since $n^k = n \cdot n^{k-1}$. Therefore, the problem can be solved by induction on k, and the resulting straightforward algorithm is given in Fig. 9.1. We have reduced the value of k, but not its size. The straightforward algorithm requires k iterations. Since the size of k is $\log_2 k$, the number of interation is exponential in the size of k ($k = 2^{\log_2 k}$). This is not bad for very small values of k, but it is unacceptable for large values of k.

Another way to reduce the problem is to use the fact that $n^k = (n^{k/2})^2$. With this observation, we reduce the problem to one with n and $k/2$. Reducing the value of k by half corresponds to reducing its size by a constant. Thus, the number of multiplications will be linear in the size of k. We now have the skeleton of the algorithm — repeated squaring. The simplest case is for $k = 2^j$ for some integer j:

Algorithm Power *(n, k) ; { first attempt }*
Input: n and k (two positive integers).
Output: P (the value of n^k).

```
begin
   P := n ;
   for i := 1 to k - 1 do
      P := n*P
end
```

Figure 9.1 Algorithm *Power*.

$$n^k = n^{2^j} = \underbrace{\left(\left(\left(n^2\right)^2\right)^{\cdots 2}\right)}_{j \text{ times.}}$$

But what if k is not a power of 2? Consider again the reduction we just used. We started with two parameters n and k, and reduced the problem to a smaller one with n and $k/2$. This reduction is not always valid since $k/2$ may not be an integer. If $k/2$ is not an integer, the reduced problem does not satisfy the conditions of the original problem. But if $k/2$ is not an integer, then $(k-1)/2$ is an integer, and the following reduction is appropriate:

$$n^k = n\left(n^{(k-1)/2}\right)^2.$$

We now have an algorithm. If k is even, we simply square the solution for $k/2$. If k is odd, we square the solution for $(k-1)/2$ and multiply by n. The number of multiplications is at most $2\log_2 k$. The algorithm is given in Fig. 9.2.

Complexity The number of multiplications is $O(\log k)$. As the algorithm progresses, however, the numbers become larger. Therefore, the multiplications become more costly. We leave it to the reader (Exercise 9.12) to analyze the complexity of this algorithm under a more realistic measure for the cost of the multiplications. We now present an application of this algorithm in which the numbers do not grow during the execution of the algorithm.

An Application to Cryptography

The study of cryptography is beyond the scope of this book, and we discuss it briefly. Encryption schemes usually rely on complete secrecy. Any two participants who want to exchange secret messages must agree on the encryption–decryption algorithm and must use secret keys known only to themselves. We want to avoid this need to exchange secret keys between every pair of participants. The following is known as the **RSA public-key encryption scheme** (after Rivest, Shamir, and Adleman [1978], who developed it). The scheme can be used by a group of participants (e.g., computer users)

```
Algorithm Power_by_Repeated_Squaring (n, k) ;
Input: n and k (two positive integers).
Output: P (the value of n^k).

begin
   if k = 1 then P := n
   else
      z := Power_by_Repeated_Squaring (n, k div 2) ;
      if k mod 2 = 0 then
         P := z*z
      else
         P := n*z*z
end
```

Figure 9.2 Algorithm *Power_by_Repeated_Squaring*.

who want to communicate by encrypted messages. Each participant has only two keys, one for encryption and one for decryption (independent of the number of other participants). These keys are chosen as follows. A participant P in the RSA scheme selects two very large prime numbers p and q and computes their product $n = pq$. He then chooses another very large integer d, such that d and $(p-1)(q-1)$ have no common divisor. (See the next section for an algorithm to verify that fact; if d is a random number, then the condition above is likely to occur.) From p, q, and d, it is possible (although not easy) to compute the value of a number e that satisfies

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)}. \tag{9.1}$$

As we shall see next, e will be the encryption key and d the decryption key. The values of n and e are publicized by P in a central directory that everyone can read. (We assume the availability of a trusted directory such that no other person can forge P's keys.) The value of d, as well as the values of p and q, which are not needed anymore, are kept secret by P.

Let M be an integer that corresponds to a message that P wants to encrypt (every message can be translated to a sequence of bits, which can be translated to an integer). Assume that M is smaller than n; otherwise M can be broken into several small messages each smaller than n. The encryption function E_P that P uses is very simple:

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

Since both n and e are made public, everyone can encrypt messages and send them to P. The decryption function D_P is just as simple (but it can be performed only by P, since the value of d is secret):

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

One can prove that (9.1) guarantees that $D_P(E_P(M)) = M$, hence these are valid encryption and decryption functions. Both algorithms thus consist of computing only one power (M^e or C^d) and one division (for the congruence), although these operations are performed on very large numbers. The modulo n operation can be applied at *any step* of the algorithm, and not necessarily at the end. This is true because

$$x \cdot y \pmod{n} = [\,x \pmod{n} \cdot y \pmod{n}\,] \pmod{n},$$

for all integers x, y, and n. Applying the modulo n operation in each step of the computation is very important, since this way the values of the operands do not grow above n. If we use algorithm *Power_by_Repeated-Squaring* of Fig. 9.2, not only do we require only $O(\log e)$ (or $O(\log d)$) multiplications and divisions for computing the power, but each multiplication and division involves numbers that are less than n. We need to modify algorithm *Power_by_Repeated-Squaring* by only changing each multiplication to a multiplication modulo n. Thus, applying the RSA scheme requires only $O(\log n)$ multiplications and divisions of numbers that are less than n.

There is no known algorithm that can factor a very large number (e.g., of 1000 digits) in a reasonable time (e.g., our lifetime). Thus, the knowledge of the value of n does not imply the knowledge of p and q. It is commonly believed (although there is no known proof of this fact) that it is impossible to compute the function D_P efficiently without the knowledge of any one of d, p, or q.[1] Therefore, by keeping d, p, and q secret, P can receive encrypted messages from anyone without compromising the secrecy of the messages. There are several other advantages of this scheme, which is called a **public-key cryptosystem**.

9.3 Euclid's Algorithm

The **greatest common divisor** of two positive integers n and m, denoted by GCD(n, m), is the unique positive integer k such that (1) k divides both n and m, and (2) all other integers that divide both n and m are smaller than k.

> **The Problem** Find the greatest common divisor of two given positive integers.

As usual, we try to reduce the problem to one of smaller size. Can we somehow make n or m smaller without changing the problem? Euclid noticed the obvious positive answer: If k divides both n and m, then it divides their difference! If $n > m$, then GCD(n, m) = GCD($n-m$, m), and we now have a smaller problem. But, again, we reduced the *values*

[1] It is known that an algorithm for computing d from n and e would lead to an efficient probabilistic algorithm for factoring n, which is a strong evidence that d cannot be compromised (see Bach, Miller, and Shallit [1986]). Potentially, however, there may be another way to compute D_P without the knowledge of d.

of the numbers in question, and not their *sizes*. For the algorithm to be efficient, we must reduce the sizes. For example, if n is very large (say 1000 digits) and $m = 24$, we will need to subtract 24 from n approximately $n/24$ times. This computation will take $O(n)$ steps, which is exponential in the size of n.

Let's look at this algorithm again. We subtract m from n and apply the same algorithm to $n-m$ and m. If $n-m$ is still larger than m, we subtract m again. In other words, we keep subtracting m from n until the result becomes less than m. But this is exactly the same as dividing n by m and looking at the remainder. Division can be done quickly. This leads directly to Euclid's algorithm, which is presented in Fig. 9.3.

Complexity We claim that Euclid's algorithm has linear running time in the size of $n+m$; specifically, its running time (counting each operation as one step independent of the size of the operands) is $O(\log(n+m))$. To prove this claim, it is sufficient to show that the value of a is reduced by half in a constant number of iterations. Let's look at two consecutive iterations of algorithm *GCD*. In the first iteration, a and b $(a > b)$ are changed into b and $a \bmod b$. Then, in the next iteration, they are changed into $a \bmod b$ and $b \bmod (a \bmod b)$. So, in two iterations, the first number a is changed to $a \bmod b$. But, since $a > b$, we have $a \bmod b < a/2$, which establishes the claim.

9.4 Polynomial Multiplication

Let $P = \sum_{i=0}^{n-1} p_i x^i$, and $Q = \sum_{i=0}^{n-1} q_i x^i$, be two polynomials of degree $n-1$. A polynomial is represented by its ordered list of coefficients.

Algorithm GCD *(m, n)*
Input: *m* and *n* (two positive integers).
Output: *gcd* (the gcd of *m* and *n*).

```
begin
   a := max( n, m ) ;
   b := min( n, m ) ;
   r := 1 ;
   while r > 0 do  { r is the remainder }
      r := a mod b ;
      a := b ;
      b := r ;
   gcd := a
end
```

Figure 9.3 Algorithm *GCD*.

The Problem Compute the product of two given polynomials of degree $n-1$.

$$PQ = \left(p_{n-1}x^{n-1} + \cdots + p_0\right)\left(q_{n-1}x^{n-1} + \cdots + q_0\right) = \tag{9.2}$$

$$p_{n-1}q_{n-1}x^{2n-2} + \cdots + \left(p_{n-1}q_{i+1} + p_{n-2}q_{i+2} + \cdots + p_{i+1}q_{n-1}\right)x^{n+i} + \cdots + p_0q_0.$$

We can compute the coefficients of PQ directly from (9.2). It is easy to see that, if we follow (9.2), then the number of multiplications and additions will be $O(n^2)$. Can we do better? We have seen by now so many improvements of straightforward quadratic algorithms that it is not surprising that the answer is positive. A complicated $O(n \log n)$ algorithm will be discussed in Section 9.6. But first, we describe a simple divide-and-conquer algorithm.

For simplicity, we assume that n is a power of 2. We divide each polynomial into two equal-sized parts. Let $P = P_1 + x^{n/2}P_2$, and $Q = Q_1 + x^{n/2}Q_2$, where

$$P_1 = p_0 + p_1x + \cdots + p_{n/2-1}x^{n/2-1}, \quad P_2 = p_{n/2} + p_{n/2+1}x + \cdots + p_{n-1}x^{n/2-1},$$

and

$$Q_1 = q_0 + q_1x + \cdots + q_{n/2-1}x^{n/2-1}, \quad Q_2 = q_{n/2} + q_{n/2+1}x + \cdots + q_{n-1}x^{n/2-1}.$$

We now have

$$PQ = (P_1 + P_2x^{n/2})(Q_1 + Q_2x^{n/2}) = P_1Q_1 + (P_1Q_2 + P_2Q_1)x^{n/2} + P_2Q_2x^n.$$

The expression for PQ now involves products of polynomials of degree $n/2$. We can compute the product of the smaller polynomials (e.g., P_1Q_1) by induction, then add the results to complete the solution. Can we use induction directly? The only constraints are that the smaller problems be exactly the same as the original problem, and that we know how to multiply polynomials of degree 1. Both conditions are clearly satisfied. The total number of operations $T(n)$ required for this algorithm is given by the following recurrence relation:

$$T(n) = 4T(n/2) + O(n), \quad T(1) = 1.$$

The factor 4 comes from the 4 products of the smaller polynomials, and the $O(n)$ comes from adding the smaller polynomials. The solution of this recurrence relation is $O(n^2)$ (see Section 3.5.2), which means that we have not achieved any improvement (see Exercise 9.4).

To get an improvement to the quadratic algorithm we need to solve the problem by solving less than four subproblems. Consider the following multiplication table (the reason we use such an elaborate table for this simple notation will become apparent in the next section).

×	P_1	P_2
Q_1	A	B
Q_2	C	D

We want to compute $A+(B+C)x^{n/2}+Dx^n$. The important observation is that we do not have to compute B and C separately; we need only to know their sum! If we compute the product $E=(P_1+P_2)(Q_1+Q_2)$, then $B+C=E-A-D$. Hence, we need to compute only three products of smaller polynomials: A, D, and E. All the rest can be computed by additions and subtractions, which contribute only $O(n)$ to the recurrence relation anyway. The new recurrence relation is

$$T(n)=3T(n/2)+O(n),$$

which implies $T(n)=O(n^{\log_2 3})=O(n^{1.59})$.

Notice that the polynomials P_1+P_2 and Q_1+Q_2 are related to the original polynomials in a strange way. They are formed by adding coefficients whose indices differ by $n/2$. This is quite a nonintuitive way to multiply polynomials, yet this algorithm reduces the number of operations significantly for large n.

□ Example 9.1

Let $P=1-x+2x^2-x^3$, and $Q=2+x-x^2+2x^3$. We compute their product using the divide-and-conquer algorithm. We carry the recursion only one step.

$$A=(1-x)\cdot(2+x)=2-x-x^2,$$

$$D=(2-x)\cdot(-1+2x)=-2+5x-2x^2,$$

and

$$E=(3-2x)\cdot(1+3x)=3+7x-6x^2.$$

From E, A, and D, we can easily compute $B+C=E-A-B$:

$$B+C=3+3x-3x^2.$$

Now, $P\cdot Q=A+(B+C)x^{n/2}+Dx^n$, and we have

$$P\cdot Q=2-x-x^2+3x^2+3x^3-3x^4-2x^4+5x^5-2x^6$$

$$=2-x+2x^2+3x^3-5x^4+5x^5-2x^6.$$

Notice that we used 12 multiplications compared to 16 in the straightforward algorithm, and 12 additions and subtractions instead of 9. (We could have reduced the number of multiplications to 9 if we had carried the recursion one more step.) The savings are, of course, much larger when n is large. (The number of additions and subtractions remains within a constant factor of that in the straightforward algorithm, whereas the number of multiplications is reduced by about $n^{0.4}$.) □

9.5 Matrix Multiplication

The product C of two $n\times n$ matrices A and B is defined as follows:

$$c_{ij}=\sum_{k=1}^{n} a_{ik}\cdot b_{kj}. \tag{9.3}$$

The Problem Compute the product $C=A\times B$ of two $n\times n$ matrices of real numbers.

The straightforward way (and seemingly the *only* way) to compute matrix product is to follow (9.3), which requires using n^3 multiplications and $(n-1)n^2$ additions. Notice that n represents the number of rows and columns in the matrix, rather than the size of the input, which is n^2. We now present two different schemes that show the possibilities for improvements.

9.5.1 Winograd's Algorithm

Assume, for simplicity, that n is even. Denote

$$A_i=\sum_{k=1}^{n/2} a_{i,2k-1}\cdot a_{i,2k}, \quad \text{and} \quad B_j=\sum_{k=1}^{n/2} b_{2k-1,j}\cdot b_{2k,j}.$$

After rearranging terms, we get

$$c_{i,j}=\sum_{k=1}^{n/2}(a_{i,2k-1}+b_{2k,j})\cdot(a_{i,2k}+b_{2k-1,j})-A_i-B_j.$$

But the A_is and B_js need to be computed only once for each row or column. To compute all the A_is and B_js requires only n^2 multiplications. The total number of multiplications has thus been reduced to $\frac{1}{2}n^3+n^2$. The number of additions has increased by about $\frac{1}{2}n^3$. This algorithm is thus better than the straightforward algorithm in cases where additions can be performed more quickly than multiplications.

Comments This algorithm shows that rearranging the order of the computation can make a difference, even for expressions, such as matrix multiplication, which have a simple form. The next algorithm carries this idea much farther.

9.5.2 Strassen's Algorithm

We use the divide-and-conquer method in a way similar to the polynomial multiplication algorithm in Section 9.4. For simplicity, we assume that n is a power of 2. Let

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \text{ and } \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix},$$

where the $A_{i,j}$s, $B_{i,j}$s, and $C_{i,j}$s are $n/2 \times n/2$ matrices. We can use the divide-and-conquer approach and reduce the problem to computing the $C_{i,j}$s from the $A_{i,j}$s and the $B_{i,j}$s. That is, we can treat the $n/2 \times n/2$ submatrices as *elements* and consider the whole problem as one of computing a product of two 2×2 matrices of elements. (We have to be careful when we substitute elements for submatrices; this is the subject of Exercise 9.23.) The algorithm for the 2×2 product can be converted to an $n \times n$ product algorithm by substituting a recursive call each time a product of elements appears. The regular algorithm for multiplying two 2×2 matrices uses 8 multiplications. Substituting each multiplication by a recursive call, we get the recurrence relation $T(n) = 8T(n/2) + O(n^2)$, which implies that $T(n) = O(n^{\log_2 8}) = O(n^3)$. This is not surprising since we are using the regular algorithm. If we could only compute the product of two 2×2 matrices with less than 8 multiplications, we would get an algorithm that is asymptotically faster than cubic.

The most important part of the recursion is how many multiplications are required to compute the product of two 2×2 matrices. The number of additions is not as important since they always contribute $O(n^2)$ to the recurrence relation, which is not a factor in determining the asymptotic complexity. (It does affect the constant factor, however.) Strassen found that 7 multiplications are sufficient to compute the product of two 2×2 matrices. Instead of simply writing down the equations leading to Strassen's algorithm, we sketch a method that *could have* been used by Strassen to find it. This method can be used for similar problems.

Computing the product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} p & s \\ r & t \end{bmatrix}$$

is equivalent to computing the product

$$\begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \begin{bmatrix} e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} p \\ r \\ s \\ t \end{bmatrix}. \tag{9.4}$$

We write (9.4) as $A \cdot X = Y$. We are looking for ways to minimize the number of multiplications required to evaluate Y. Let's look for special matrix products that are easy to compute. As it turns out, we need four types of such special products (the last two of which are very similar). They are as follows:

Type	*Product*	*No. of Multiplications*
α)	$\begin{bmatrix} a & a \\ a & a \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e+f) \\ a(e+f) \end{bmatrix}$	1

β)	$\begin{bmatrix} a & a \\ -a & -a \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e+f) \\ -a(e+f) \end{bmatrix}$	1
γ)	$\begin{bmatrix} a & 0 \\ a-b & b \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} ae \\ ae+b(f-e) \end{bmatrix}$	2
δ)	$\begin{bmatrix} a & b-a \\ 0 & b \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e-f)+bf \\ bf \end{bmatrix}$	2

We now look for ways to divide the general matrix product given in (9.4) into several steps of the types listed above. Since these types of products use less than the nominal number of multiplications, we may be able to save something at the end. It takes a lot of trial and error to reach the right combinations. This process is hardly straightforward or even clear, but it is somewhat less than magic. Let

$$B = \begin{bmatrix} b & b & 0 & 0 \\ b & b & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & c & c \\ 0 & 0 & c & c \end{bmatrix},$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c-b & 0 & 0 & c-b \\ b-c & 0 & 0 & b-c \\ 0 & 0 & 0 & 0 \end{bmatrix}, \text{ and } \quad E = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}.$$

Then, $A = (B + C + D + E)$ and therefore $AX = BX + CX + DX + EX$. All the products above, except for EX, can be computed with one multiplication using types α or β. The only problem is to compute EX. But E can be divided into two matrices $E = F + G$, such that F is of type γ and G is of type δ:

$$F = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}$$

So, overall, $AX = (B + C + D + F + G)X$, and we need two products of type α, and one product each of types β, γ, and δ, with a total of 7 multiplications (see also Exercise 9.10).

Complexity We use 7 products of matrices of half the original size, and a constant number of additions of matrices. The additions are less important than the products, because addition of two $n \times n$ matrices can be done in time $O(n^2)$, which is basically a linear time in the size of the matrices. The $O(n^2)$ term is not the dominant factor in the recurrence relation, which is $T(n) = 7T(n/2) + O(n^2)$. The solution of this recurrence relation is $T(n) = O(n^{\log_2 7})$, which is approximately $O(n^{2.81})$. If we use the derivation described above, we obtain 18 additions (see Exercise 9.10). It is possible to reduce the number of additions to 15 (Winograd [1973]), but this reduction does not change the asymptotic running time.

Comments There are three major drawbacks to Strassen's algorithm:

1. Empirical studies indicate that n needs to be at least 100 to make Strassen's algorithm faster than the straightforward $O(n^3)$ algorithm (Cohen and Roth [1976]).
2. Strassen's algorithm is less stable than the straightforward algorithm. That is, for similar errors in the input, Strassen's algorithm will probably create larger errors in the output.
3. Strassen's algorithm is obviously much more complicated and harder to implement than the straightforward algorithm. Furthermore, Strassen's algorithm cannot be easily parallelized, whereas the regular algorithm can.

Nevertheless, Strassen's algorithm is important. It is faster than the regular algorithm for large n, and it can be used for other problems involving matrices, such as matrix inversion and determinant computation. We will see in Chapter 10 that several other problems are equivalent to matrix multiplication. Strassen's algorithm can be improved in practice by using it only for large matrices and stopping the recursion when the size of the matrices become smaller than about 100. This is similar to the idea of selecting the base of the induction with care, which we discussed in Section 6.4.4 and Section 6.11.3. Strassen's algorithm also opened the door to other algorithms and raised many questions about similar problems that seemed unsolvable.

9.5.3 Boolean Matrices

In this section, we consider the special case of computing the product of two $n \times n$ Boolean matrices. All elements are 0 or 1, and the sum and product are defined by the following rules (which correspond to *or* and *and* respectively):

+	0	1
0	0	1
1	1	1

×	0	1
0	0	0
1	0	1

These definitions of sum and product are of course different from the usual integer sum and product; hence, algorithms designed for integers normally cannot be used for Booleans. One problem with the definition of a Boolean sum is that subtraction is not well defined (both $0+1$ and $1+1$ are defined as 1; hence, $1-1$ can be both 1 and 0).

Therefore, Strassen's algorithm cannot be used for Boolean matrices, because it requires subtraction. However, there is a trick that allows us to use Strassen's algorithm. We consider every bit as an integer modulo $n+1$, where n is the size of the matrices, and we use the rules of addition and multiplication of such integers. So, for example, if $n=4$, then $1+1=2$, $1+1+1=3$, and $1+1+1+1+1=0$. It turns out that, if we compute the matrix product according to these rules and if we substitute every nonzero entry in the final result by a 1, then we get the Boolean product. This is so, essentially, because we will not "overflow" the number $n+1$ (we omit the proof). (More precisely, the integers modulo k form a **ring**, which is an algebraic structure with definitions of sums and products that satisfy certain properties; Strassen's algorithm can be applied to any ring; see Aho, Hopcroft, and Ullman [1974] for more details.) Thus, the complexity of Boolean matrix multiplication is also $O(n^{2.81})$. The use of Strassen's algorithm, however, requires integer operations rather than Boolean operations. Next, we present two algorithms that utilize the properties of Boolean operations to improve the running time of Boolean matrix multiplication. These algorithms are more practical in most situations than Strassen's algorithm for Boolean matrix multiplication.

Since Boolean operands require only one bit of storage, we can store k operands in one computer word of size k. In particular, since we assume that n is stored in one computer word, we can store k bits for $k \leq \log_2 n$ in one word. The regular algorithm for matrix multiplication consists of n^2 row-by-column products (or **inner products**), as defined in (9.3). The ijth inner product consists of computing $\sum_{m=1}^{n} a_{im} \cdot b_{mj}$. Assume, for simplicity, that k divides n. We can divide each inner product into a sum of n/k products, each of which involves Boolean vectors of size k. Finding the inner product of two Boolean vectors of size k is simpler than, say, multiplying two k-bit integers. We assume that a multiplication of k-bit integers takes one unit of time; thus, it is not unreasonable to assume that computing an inner product of two Boolean vectors of size k takes one unit of time. (For example, an inner product can be computed in two steps: first, we compute the *and* of the two vectors, then we check whether the result is all 0s.) Nevertheless, we usually do not want to make the algorithm dependent on special assumptions concerning the computer primitives (besides the four basic arithmetic operations). Next, we show how to avoid the need for such assumption. Then, we combine this idea with another idea to improve Boolean matrix multiplication even further. Both ideas illustrate interesting techniques for algorithm design.

The first idea is to precompute all possible Boolean inner products of size k. There are 2^{2k} possible products, since they involve two Boolean vectors of size k. We can compute all of them in time $O(k2^{2k})$ (we can actually do better than that; see Exercise 9.24), and store all the results in a two-dimensional table of bits of size $2^k \times 2^k$. The product of the two vectors a and b is stored at entry (i_a, i_b), where i_a is the integer represented by the k bits of a and i_b is the integer represented by the k bits of b. From now on, we will not make a distinction between i_a and a (or i_b and b), since they are represented in exactly the same way. Thus, given two Boolean vectors of size k, we can compute their product by simply looking at the table. If we can access a table of size 2^{2k} in $O(1)$ time, then each inner product of size k can be computed in constant time (once

the table is constructed). For example, let $k = \lfloor \log_2 n/2 \rfloor$. In that case, the size of the table is $O(n)$, and constructing it requires $O(n \log n)$ time. The assumption that we can access a table of size $O(n)$ in constant time is not unusual. We have already made this assumption (implicitly) many times before. We usually assume that, if n is the size of the input, then we can store a number with $\log_2 n$ bits in one computer word (or a constant number of computer words). Once the table is constructed, we can compute a Boolean inner product of size n in time $O(n/k) = O(n/\log n)$. Notice that the table depends only on the value of k and not on the matrices. So, computing the product of two Boolean matrices can be done in time $O(n^3/\log n)$ and extra storage of $O(n)$. We can also choose k to be $\lfloor \log_2 n \rfloor$, in which case the table size is $O(n^2)$, but we save an extra factor of 2 in the multiplication algorithm. However, if we can afford an extra space of size $O(n^2)$, we can find a faster algorithm.

Consider two $n \times n$ Boolean matrices A and B. The usual way to view matrix multiplication is as defined in (9.3): We perform n^2 inner products, each involves a row of A and a column of B. We can also multiply the two matrices by multiplying columns of A with rows of B in the following way. Denote the rth column of A by $A_C[r]$, and the rth row of B by $B_R[r]$. Consider $A_C[r]$ as an $n \times 1$ matrix, and $B_R[r]$ as a $1 \times n$ matrix. The product of $A_C[r]$ with $B_R[r]$ is an $n \times n$ matrix, whose ijth entry is the product of the ith entry of $A_C[r]$ with the jth entry of $B_R[r]$ (see Fig. 9.4). It is easy to see that

$$A \cdot B = \sum_{r=1}^{n} A_C[r] \cdot B_R[r]. \tag{9.5}$$

The expression (9.5) is equivalent to (9.3) in the sense that the same products and additions are performed, but they are performed in a different order.

We now partition the columns of A and the rows of B into n/k equal-sized groups. (We assume for simplicity that n/k is an integer; otherwise, there will be an extra smaller group.) In other words, we divide A into $A_1, A_2, \ldots, A_{n/k}$, such that each A_i is an $n \times k$ matrix, and we divide B into $B_1, B_2, \ldots, B_{n/k}$, such that each B_i is an $k \times n$ matrix. It is easy to see that

$$A \cdot B = \sum_{i=1}^{n/k} A_i \cdot B_i. \tag{9.6}$$

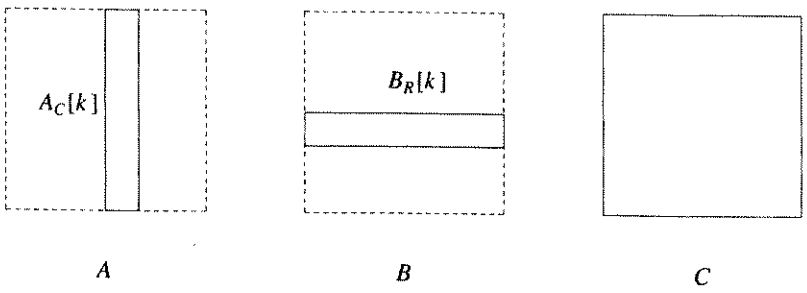

Figure 9.4 Multiplying matrices columns by rows.

The problem now is how to compute $C_i = A_i \cdot B_i$ efficiently. We describe this computation by an example (see Fig. 9.5).

The first row of C_i is exactly the same as the third row of B_i, because the first row of A_i has a 1 only in column 3. Similarly, the second row of C_i is the Boolean sum of the second and third rows of B_i. It is easy to see that the jth row of C_i is a Boolean sum of rows of B_i according to the jth row of A_i. Instead of computing each row of C_i in a straightforward way, we use a method, similar to the algorithm we described earlier, for precomputing all possibilities. There are k entries in each row of A_i, so there are 2^k possible combinations of rows of B_i. Let $k = \log_2 n$, and assume again that k is an integer. We precompute all $2^k = 2^{\log_2 n} = n$ combinations, and store the results in a table. In contrast to the first algorithm, this table contains n rows rather than n bits; thus, the storage requirement is $O(n^2)$. Also, this table depends on B_i, and must be constructed for each B_i. To find row j of C_i, we look at row j of A_i and see the combination of rows of B_i that need to be added. This combination can be represented as an integer corresponding to the binary representation of row j of A_i (e.g., the first row of A_i in Fig. 9.5 corresponds to 1, the second row corresponds to 3, the third row corresponds to 4, and so on). This integer is the address in the table where row j of C_i is stored. It takes $O(1)$ time to find a row of C_i in the table, and $O(n)$ time to copy this row to the appropriate row in C_i. Thus, computing C_i can be done in time $O(n^2)$.

We now show that all the combinations of sums of rows of B_i can be computed in time $O(n \cdot 2^k)$. Each combination of rows corresponds to a k-bit integer. We assume, by induction, that we know how to compute the sums of combinations of rows corresponding to integers that are less than i. Computing the sum corresponding to 0 is trivial. Assume that the binary representation of $i-1$ is $xxxx011111$ — namely, its least significant 0 is followed by j 1s. The sum of rows corresponding to i is equal to the sum of rows corresponding to $xxxx000000$ plus the row corresponding to 0000100000. Since $xxxx000000$ is less than i, we know its corresponding sum by induction, and we need only to add one row to it. It takes n Boolean additions to add a row, and we have 2^k combinations. Hence, all the precomputing can be done with $O(n \cdot 2^k)$ operations. If $k = \log_2 n$, then the running time is $O(n^2)$. This algorithm is known as the **four-Russians**

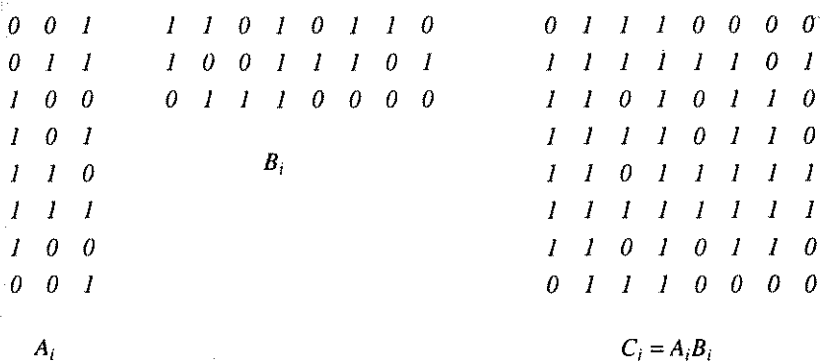

Figure 9.5 Boolean matrix multiplication.

algorithm (Arlazarov et al. [1970]), after the nationality and number of its inventors. The algorithm is given in Fig. 9.6.

```
Algorithm Boolean_Matrix_Multiplication (A, B, n, k) ;
Input: A, B (two n × n Boolean matrices), and k (an integer).
Output: C (the product of A and B).
    { we assume, for simplicity, that k divides n }
begin
    Initialize the matrix C to 0 ;
    for i := 0 to n/k − 1 do
        Construct Table_i ;
        { Table_i is an 2^k array of Boolean vectors of size n which contains
          all possible combinations of sums of k rows of B_i; see the text }
        m := i * k ;
        for j := 1 to n do
            Let Addr be the k-bit number
                A[j, m+1] A[j, m+2] ··· A[j, m+k] ;
            add Table_i[Addr] to row j in C
end
```

Figure 9.6 Algorithm *Boolean_Matrix_Multiplication*.

Complexity To compute $A \cdot B$ we have to compute the n/k products $A_i \cdot B_i$. Since each such product takes $O(n^2)$ time and constructing the table takes $O(n \cdot 2^k)$ time, the total running time of the algorithm is $O(n^3/k + n^2 \cdot 2^k/k)$. If $k = \log_2 n$, then the running time is $O(n^3/\log n)$.

Next, we show how to combine the ideas of the first algorithm with the ideas of the second algorithm to improve the running time by another $O(\log n)$ factor. The main step in algorithm *Boolean_Matrix_Multiplication* (Fig 9.6) involves additions of a row from a table to C. We can perform this addition in time $O(n/m)$ by using the same trick of precomputing all possible additions. (This may not be necessary if a Boolean addition is a primitive operation that can be performed quickly; the algorithm, however, does not depend on this assumption.) We first construct a two-dimensional table *Add_Table* of size $2^m \times 2^m$ that includes all possible additions of two Boolean vectors of size m. In other words, the (i, j)th entry in *Add_Table* is the Boolean sum of i and j. (Again, i and j are used both as integers and as Boolean vectors.) It is easy to see that *Add_Table* can be constructed in time and space $O(m \cdot 2^{2m})$. Notice that, unlike the tables we used in algorithm *Boolean_Matrix_Multiplication* (Fig. 9.6), *Add_Table* is independent of A and B; it depends only on the value of m. We now divide each row of B_i into n/m groups, each of size m (we assume again, for simplicity, that m divides n). We consider each group as a m-bit integer; thus, each row of B_i is represented by an n/m-tuple of integers. All the steps of the algorithm will be performed on these tuples.

To add two vectors of size n, we use *Add_Table* to add the corresponding two n/m-tuples in n/m steps. Each step consists of taking two m-bit numbers and fetching the corresponding entry in *Add_Table* (which contains their sum). Such a step can be performed in constant time, as long as the size of the computer word is at most $2m$. We use this trick both for constructing the tables for the regular four-Russians algorithm, and for adding the rows during the execution of the algorithm. If we select m to be approximately equal to $\lfloor \log_2 n/2 \rfloor$, then $2^{2m} = O(n)$ and, since we assume that we can represent n in one computer word, we can represent a $2m$-bit number in one word. For this choice of m, the running time of the improved algorithm is $O(n^3/\log^2 n)$.

Comments We presented an interesting method of computing all possibilities instead of the usual wisdom of computing only what is needed. We also demonstrated that changing the order of the computation can lead to a better algorithm. The trick of computing all possible combinations can be applied in the same manner to other algebraic functions on bit strings that cannot be performed directly by the hardware.

9.6 The Fast Fourier Transform

As an introduction to the fast Fourier transform, we quote from John Lipson's excellent book:

> An algorithm may be appreciated on a number of grounds; on technological grounds because it efficiently solves an important practical problem, on aesthetic grounds because it is elegant, or even on dramatic grounds because it opens up new and unexpected areas of applications. The *fast Fourier transform* (popularly referred to as the "FFT"), perhaps because it is strong on *all* of these departments, has emerged as one of the "super" algorithms of Computer Science since its discovery in the mid sixties. (Lipson [1981], page 293.)

The FFT algorithm is by no means simple, and its development is not straightforward. We concentrate on only one application of the FFT — polynomial multiplication.

The Problem Given two polynomials $p(x)$ and $q(x)$, compute their product $p(x) \cdot q(x)$.

The problem, as stated above, is not well defined. We have not specified the representation of the polynomials. We usually represent a polynomial $p = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$ by the list of its coefficients in increasing order of degrees. This representation is definitely adequate, but it is not the only one possible. Consider, for example, a polynomial of degree 1, which is a linear function $a_1 x + a_0$. This linear function is usually specified by the two coefficients a_1 and a_0. But, since the function corresponds to a line in the plane, it can also be specified by any

two (nonequal) points on that line. In the same way, any polynomial of degree n is uniquely defined by $n+1$ points. For example, the second-degree polynomial $p(x)=x^2+3x+1$ is defined by the points (1,5), (2,11), and (3,19), and it is the *only* second degree polynomial that includes all those points. These three points are not the only three points that define this polynomial; any three points on the corresponding curve will do.

This representation is attractive for polynomial multiplication because multiplying the values of points is easy. For example, the polynomial $q(x)=2x^2-x+3$ can be represented by (1,4), (2,9), and (3,18). We right away know that the product $p(x)\cdot q(x)$ has the values (1,20), (2,99), and (3,342). These three points are not enough to represent $p(x)\cdot q(x)$ since it has degree 4. We can overcome this problem by requiring five points from each of the smaller polynomials; for example, we can add the points (0,1) and $(-1,-1)$ to $p(x)$, and (0, 3) and $(-1, 6)$ to $q(x)$. We can then easily obtain five points that belong to the product — (1, 20), (2, 99), (3, 342), (0, 3), and $(-1,-6)$ — by making only five scalar multiplications! Using this idea, we can compute the product of two polynomials of degree n, given in this representation, with only $O(n)$ multiplications.

The main problem with this approach is that we cannot simply change the representation to fit only one application. We must be able, for example, to evaluate the polynomial at given points. This is much harder to do for this representation than it is when the coefficients are given. However, if we could convert efficiently from one representation to another, then we would have a very good polynomial multiplication algorithm. This is what the FFT achieves.

Converting from coefficients to points can be done by polynomial evaluation. We can compute the value of a polynomial $p(x)$, given by its list of coefficients, at any given point by Horner's rule (Section 5.2) using n multiplications. We need to evaluate $p(x)$ at n arbitrary points, so we require n^2 multiplications. Converting from points to coefficients is called **interpolation**, and it also generally requires $O(n^2)$ operations. The key idea here (as in so many other examples in this book) is that we do not have to use *arbitrary* points; we are free to choose *any* set of n distinct points we want. The fast Fourier transform chooses a very special set of points such that both steps, evaluation and interpolation, can be done quickly.

The Forward Fourier Transform

We first consider the evaluation problem. We need to evaluate two $n-1$ degree polynomials, each at $2n-1$ points, so that their product, which is a $2n-2$ degree polynomial, can be interpolated. However, we can always represent an $n-1$ degree polynomial as a $2n-2$ degree polynomial by setting the first $n-1$ (leading) coefficients to zero. So, without loss of generality, we assume that the problem is to evaluate an arbitrary polynomial $P=\sum_{i=0}^{n-1} a_i x^i$ of degree $n-1$ at n distinct points. We want to find n points for which the polynomials are easy to evaluate. We assume, for simplicity, that n is a power of 2.

We use matrix terminology to simplify the notation. The evaluation of the polynomial P above for the n points $x_0, x_1, \ldots, x_{n-1}$ can be represented as the following matrix by vector multiplication:

$$\begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & (x_{n-1})^2 & \cdots & (x_{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{bmatrix}.$$

The question is whether we can choose the values of $x_0, x_1, \ldots, x_{n-1}$ in a way that simplifies this multiplication. Consider two arbitrary rows i and j. We would like to make them as similar as possible to save multiplications. We cannot make $x_i=x_j$, because the values must be different, but we can make $(x_i)^2=(x_j)^2$ by letting $x_j=-x_i$. This is a good choice, because every even power of x_i will be equal to the same even power of x_j. We may be able to save one-half of the multiplications involved with row j. Furthermore, we can do the same for other pairs of rows. Our goal is to have n special rows for which the computation above requires only $n/2$ vector products. If we can do that, then we may be able to cut the problem size by half, which will lead to a very efficient algorithm. Let's try to pose this problem in terms of two separate subproblems of half the size.

We want to divide the original problem into two subproblems of size $n/2$, according to the scheme described above. This is illustrated in the following expression.

$$\begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n/2-1} & (x_{n/2-1})^2 & \cdots & (x_{n/2-1})^{n-1} \\ 1 & -x_0 & (-x_0)^2 & \cdots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \cdots & (-x_1)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & -x_{n/2-1} & (-x_{n/2-1})^2 & \cdots & (-x_{n/2-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n/2-1}) \\ P(-x_{n/2-1}) \end{bmatrix}. \quad (9.7)$$

The $n \times n$ matrix in (9.7) is divided into two submatrices, each of size $n/2 \times n$. These two matrices are very similar. For each i, such that $0 \le i < n/2$, we have $x_i=-x_{n/2+i}$. The coefficients of the even powers are exactly the same in both submatrices, so they need to be computed only once. The coefficients of the odd powers are not the same, but they are exactly the negation of each other! We would like to write the expressions for $P(x_i)$ and $P(-x_i)$ for $0 \le i < n/2$ in terms of the even and odd coefficients:

$$P(x) = E + O = \sum_{i=0}^{n/2-1} a_{2i} x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1} x^{2i+1}.$$

The "even" polynomial (E) can be written as a regular polynomial of degree $n/2-1$

with the even coefficients of P:

$$E = \sum_{i=0}^{n/2-1} a_{2i}(x^2)^i = P_e(x^2).$$

The "odd" polynomial (O) can be written in the same way:

$$O = x \sum_{i=0}^{n/2-1} a_{2i+1}(x^2)^i = x\,P_o(x^2).$$

So, overall, we have the following expression:

$$P(x) = P_e(x^2) + x\,P_o(x^2), \tag{9.8}$$

where P_e (P_o) are the $n/2-1$ degree polynomials with the coefficients of the even (odd) powers of P. When we substitute $-x$ for x in (9.8), we get $P(-x) = P_e(x^2) + (-x)P_o(x^2)$. To evaluate (9.7), we need to compute $P(x_i)$ and $P(-x_i)$, for $0 \le i < n/2$. To do that, we need to compute only $n/2$ values of $P_e(x^2)$ and $n/2$ values of $P_o(x^2)$, and to perform $n/2$ additions, $n/2$ subtractions, and n multiplications. So, we have two subproblems of size $n/2$, and $O(n)$ additional computations.

Can we continue with the same scheme recursively? If we could, then we would get the familiar recurrence relation $T(n) = 2T(n/2) + O(n)$, resulting in an $O(n \log n)$ algorithm. But this is not so easy. We reduced the problem of computing $P(x)$ (a polynomial of degree $n-1$) at n points to that of computing $P_e(x^2)$ and $P_o(x^2)$ (both polynomials of degree $n/2-1$) at $n/2$ points. This is a valid reduction, except for one small thing. The values of x in $P(x)$ can be chosen arbitrarily, but the values of x^2 which are needed, for example, in $P_e(x^2)$, can only be positive. Since we obtained this reduction by using negative numbers, this poses a problem. Let's extract from (9.7) the matrix that corresponds to the computation of $P_e((x_i)^2)$:

$$\begin{bmatrix} 1 & (x_0)^2 & (x_0)^4 & \cdot\cdot & (x_0)^{n-2} \\ 1 & (x_1)^2 & (x_1)^4 & \cdot\cdot & (x_1)^{n-2} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & (x_{n/2-1})^2 & (x_{n/2-1})^4 & \cdot\cdot & (x_{n/2-1})^{n-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} P_e(x_0) \\ P_e(x_1) \\ \cdot \\ \cdot \\ \cdot \\ P_e(x_{n/2-1}) \end{bmatrix}.$$

If we try the same trick on this subproblem, we need to set $(x_{n/4})^2 = -(x_0)^2$. Since squares are always positive, this seems impossible. But it is not impossible if we use complex numbers which include $\sqrt{-1}$. We again divide the problem into two parts and let $x_{j+n/4} = \sqrt{-1}\,x_j$, for $0 \le j < n/4$. This partition satisfies the same properties as did the first partition. Hence, we can solve the problem of size $n/2$ by solving two subproblems of size $n/4$ and $O(n)$ additional computation.

If we want to carry this process one step further, we need a number that is equal to $\sqrt{\sqrt{-1}}$; that is, a number z such that $z^8 = 1$, and $z^j \ne 1$ for $0 < j < 8$ (which implies that $z^4 = -1$, and $z^2 = \sqrt{-1}$). In general, we need a number that satisfies the condition above for n rather than for 8. Such a number is called a **primitive *n*th root of unity**. We denote it by ω. (We do not include n in the notation for simplicity; we will use the same ω throughout this section.) ω satisfies the following conditions:

$$\omega^n = 1, \text{ and } \omega^j \ne 1 \text{ for } 0 < j < n. \tag{9.9}$$

The n points that we choose as $x_0, x_1, \ldots, x_{n-1}$ are $1, \omega, \omega^2, \ldots, \omega^{n-1}$. Therefore, we want to compute the following product:

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \cdot \\ \cdot \\ P(\omega^{n-1}) \end{bmatrix}.$$

This product is called the **Fourier transform** of $(a_0, a_1, \ldots, a_{n-1})$. First, we notice that indeed for any j, $0 \le j < n/2$, we have $x_{j+n/2} = \omega^{n/2} x_j = -x_j$. So the reduction that we applied initially to the problem of size n is still valid. Furthermore, the subproblems resulting from that reduction have $n/2$ points, which are $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$. But this is exactly the problem of size $n/2$ in which we substitute ω^2 for ω. The conditions in (9.9) imply that ω^2 is a primitive $(n/2)$th root of unity. Therefore, we can continue recursively, and the complexity of the algorithm is $O(n \log n)$. A high-level view of the algorithm is presented in Fig. 9.7.

Algorithm Fast_Fourier_Transform $(n, a_0, a_1, \ldots, a_{n-1}, \omega,$ ***var*** $V)$;
Input: n (an integer), $a_0, a_1, \ldots, a_{n-1}$ (a sequence of *elements* whose type depends on the application), and ω (a primitive nth root of unity).
Output: V (an array in the range $[0..n-1]$ of output elements).
{ we assume that n is a power of 2 }

```
begin
  if n = 1 then
    V[0] := a0 ;
  else
    Fast_Fourier_Transform(n/2, a0, a2, ..., an-2, ω², U ) ;
    Fast_Fourier_Transform(n/2, a1, a3, ..., an-1, ω², W ) ;
    for j := 0 to n/2 - 1 do { follow (9.8) for x = ω^j }
      V[j] := U[j] + ω^j W[j] ;
      V[j+n/2] := U[j] - ω^j W[j]
end
```

Figure 9.7 Algorithm *Fast_Fourier_Transform*.

□ Example 9.2

We show how to compute the Fourier transform for the polynomial (0, 1, 2, 3, 4, 5, 6, 7). To avoid confusion, we denote the subproblems by $P_{j_0,j_1,\ldots,j_k}(x_0, x_1, \ldots, x_k)$, where $j_0, j_1, \ldots, j_k$ denote the coefficients of the polynomials, and $x_0, x_1, \ldots, x_k$ denote the values for which we need to evaluate the polynomials. So, in particular, this example involves computing $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$. (This notation is quite awkward, but it contains all the information we need.) The main recurrence we use is (9.8).

The first step reduces $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$ to $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ and $P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6)$. We continue recursively and reduce $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ to $P_{0,4}(1, \omega^4)$ and $P_{2,6}(1, \omega^4)$. $P_{0,4}(1, \omega^4)$ is then reduced to $P_0(1)$, which is clearly 0, and $P_4(1)$, which is clearly 4. We can now combine the results to get

$$P_{0,4}(1) = P_0(1) + 1 \cdot P_4(1) = 0 + 1 \cdot 4 = 4,$$

and

$$P_{0,4}(\omega^4) = P_0(\omega^4) + \omega^4 P_4(\omega^4) = 0 + \omega^4 \cdot 4.$$

Since $\omega^4 = -1$, we get $P_{0,4}(\omega^4) = -4$, and, overall, $P_{0,4}(1, \omega^4) = (4, -4)$. In the same manner, we get $P_{2,6}(1, \omega^4) = (8, -4)$.

We now combine the two vectors above to compute $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$:

$$P_{0,2,4,6}(1) = P_{0,4}(1) + 1 \cdot P_{2,6}(1) = 4 + 8 = 12.$$

$$P_{0,2,4,6}(\omega^2) = P_{0,4}(\omega^4) + \omega^2 \cdot P_{2,6}(\omega^4) = -4 + \omega^2(-4).$$

$$P_{0,2,4,6}(\omega^4) = P_{0,4}(\omega^8) + \omega^4 \cdot P_{2,6}(\omega^8) = P_{0,4}(1) - 1 \cdot P_{2,6}(1) = 4 - 8 = -4.$$

$$P_{0,2,4,6}(\omega^6) = P_{0,4}(\omega^{12}) + \omega^6 \cdot P_{2,6}(\omega^{12}) = P_{0,4}(\omega^4) - \omega^2 \cdot P_{2,6}(\omega^4) = -4 - \omega^2(-4).$$

So, overall

$$P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6) = (\,12, -4(1+\omega^2), -4, -4(1-\omega^2)\,).$$

In the same way, we find that

$$P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6) = (\,16, -4(1+\omega^2), -4, -4(1-\omega^2)\,).$$

To compute $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$, we need to compute 8 values. For example, $P_{0,1,2,3,4,5,6,7}(1) = 12 + 1 \cdot 16 = 28$, and, in the same manner, $P_{0,1,2,3,4,5,6,7}(\omega^4) = 12 - 1 \cdot 16 = -4$; $P_{0,1,2,3,4,5,6,7}(\omega) = (-4(1+\omega^2)) + \omega \cdot (-4(1+\omega^2))$, and, in the same manner, $P_{0,1,2,3,4,5,6,7}(\omega^5) = (-4(1+\omega^2)) - \omega \cdot (-4(1+\omega^2))$, and so on. We leave the rest to the reader. □

The Inverse Fourier Transform

The algorithm for the fast Fourier transform solves only half of our problem. We can evaluate the two given polynomials $p(x)$ and $q(x)$ at the points $1, \omega, \ldots, \omega^{n-1}$ quickly, multiply the resulting values, and find the values of the product polynomial $p(x) \cdot q(x)$ at those points. But we still need to interpolate the coefficients of the product polynomial from the evaluation points. Fortunately, the interpolation problem turns out to be very similar to the evaluation problem, and an almost identical algorithm can solve it.

Consider again the matrix notation. When we are given the coefficients $(a_0, a_1, \ldots, a_{n-1})$ of the polynomial, and we want to compute the values of the polynomial at the n points $1, \omega, \omega^2, \ldots, \omega^{n-1}$, we compute the matrix by the following vector product:

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \cdot \\ \cdot \\ P(\omega^{n-1}) \end{bmatrix}.$$

On the other hand, when the values of the polynomial $(P(1), P(\omega), \ldots, P(\omega^{n-1})) = (v_0, v_1, \ldots, v_{n-1})$ are given, and we want to compute the coefficients, we need to solve the following system of equations for $a_0, a_1, \ldots, a_{n-1}$:

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \cdot \\ \cdot \\ v_{n-1} \end{bmatrix}. \tag{9.10}$$

Solving systems of equations is usually quite time consuming ($O(n^3)$ for the general case), but this is a special system of equations. Let's write this matrix equation as $V(\omega) \cdot \bar{a} = \bar{v}$, where $V(\omega)$ is the matrix in the left side, $\bar{a} = (a_0, a_1, \ldots, a_{n-1})$, and $\bar{v} = (v_0, v_1, \ldots, v_{n-1})$. The solution for $\bar{a}$ can be written as $\bar{a} = [V(\omega)]^{-1} \cdot \bar{v}$, provided that $V(\omega)$ has an inverse. It turns out that $V(\omega)$ always has an inverse; furthermore, its inverse has a very simple form (we omit the proof):

□ Theorem 9.1

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega}).$$ □

Therefore, to solve the system of equations (9.10), we need to compute only one matrix by vector product. This task is greatly simplified by the following theorem.

□ Theorem 9.2

If ω is a primitive nth root of unity, then $1/\omega$ is also a primitive nth root of unity. □

Therefore, we can compute the product $V(1/\omega)\bar{v}$ by using the algorithm for the fast Fourier transform, substituting $1/\omega$ for ω. This transform is called the **inverse Fourier transform**.

Complexity Overall, the product of two polynomials can be computed with $O(n \log n)$ operations. Notice that we need to be able to add and multiply complex numbers.

9.7 Summary

The algorithms presented in this chapter are a small sample of known algebraic and numerical algorithms. We have seen again that the straightforward algorithms are not necessarily the best. Strassen's algorithm is one of the most striking examples of a nonintuitive algorithm for a seemingly simple problem. We have seen several more examples of the use of induction, and, in particular, of the use of divide-and-conquer algorithms.

The four-Russians algorithm suggests an interesting technique, which is not based on induction. The main idea is to compute all possible combinations of certain terms, even if not all of them are needed. This technique is useful in cases where computing all (or many) combinations together costs much less than computing each one separately. Another technique, which is common particularly for problems involving matrices, is the use of reductions between problems. This method is described, with examples, in Chapter 10.

Bibliographic Notes and Further Reading

The best source for arithmetic and algebraic algorithms is Knuth [1981]. Other books include Aho, Hopcroft, and Ullman [1974], Borodin and Munro [1975], Winograd [1980], and Lipson [1981].

The algorithm for computing powers by repeated squaring is very old; it appeared in Hindu writings circa 200 B.C. (see Knuth [1981] page 441). The RSA public-key encryption scheme is due to Rivest, Shamir, and Adleman [1978]. The idea of public-key encryption schemes was introduced by Diffie and Hellman [1976]. Euclid's algorithm appeared first in Euclid's *Elements*, Book 7 (circa 300 B.C.), but it was probably known even before then (see Knuth [1981], page 318). The divide-and-conquer algorithm for multiplying two polynomials was developed by Karatsuba and Ofman [1962] (in the context of multiplying two large numbers).

Winograd's algorithm appeared in Winograd [1968] (see also Winograd [1970]). Strassen's algorithm appeared in Strassen [1969]. The constant c in the asymptotic running time $O(n^c)$ for matrix multiplication has been reduced several times since 1969 (first by Pan [1978]). The best-known algorithm at this time — in terms of asymptotic running times — is by Coppersmith and Winograd [1987], and its running time is $\mathcal{O}(n^{2.376})$. Unfortunately, as the $\mathcal{O}$ notation indicates, this algorithm is not practical. For more on the complexity of matrix multiplication and related topics see Pan [1984]. A discussion on the implementation of Strassen's algorithm can be found in Cohen and Roth [1976].

The four-Russians algorithm is due to Arlazarov, Dinic, Kronrod, and Faradzev [1970]. The improvement of the four-Russians algorithm by using addition tables has probably been observed by many people; it is mentioned, without details, in Rytter [1985], where a similar technique is used for context-free language recognition. The same idea was also used to improve sequence comparisons algorithms (Masek and Paterson [1983], Myers [1988]). The solution of Exercise 9.26 appears in Atkinson and Santoro [1988]. Fischer and Meyer [1971] showed a reduction between Boolean matrix multiplication and the transitive-closure problem.

The algorithm for the fast Fourier transform was introduced by Cooley and Tuckey [1965], although the origins of the method can be traced to Runge and König [1924]. For more information on the fast Fourier transform, see Brigham [1974] and Elliott and Rao [1982].

Drill Exercises

9.1 Discuss the relationship between algorithm *Power_by_Repeated_Squaring* (Fig. 9.2) for computing n^k and the binary representation of k.

9.2 Algorithm *Power_by_Repeated_Squaring* (Fig. 9.2) for computing n^k does not necessarily lead to the minimal number of multiplications. Show an example of computing n^k ($k > 10$) with fewer number of multiplications.

9.3 Let x be a positive rational number that is represented by the pair (a, b) such that $x = a/b$. Design an algorithm to compute the smallest representation of x; that is, the representation (a, b) with the smallest possible values of a and b. For example, if $x = 24/84 = 6/21 = 2/7$, then $(2, 7)$ is the smallest representation of x.

9.4 Prove that the straightforward divide-and-conquer algorithm for polynomial multiplication that computes all four products of the smaller polynomials makes exactly the same operations as does the straightforward algorithm that follows (9.1). Assume that n is a power of 2.

9.5 Find the product $P(x) \cdot Q(x)$, by hand, using the divide-and-conquer polynomial multiplication algorithm presented in Section 9.4.

$$P(x) = x + 2x^2 + 3x^3 + \cdots + 15x^{15}.$$

$$Q(x) = 16 + 15x + 14x^2 + \cdots + 2x^{14} + 1x^{15}.$$

How many operations are required overall?

9.6 A divide-and-conquer technique can be used to multiply two binary numbers. Describe such an algorithm, and discuss the differences between it and the polynomial multiplication algorithm.