

The straightforward method of adding two polynomials of degree  $n$  takes  $\Theta(n)$  time, but the straightforward method of multiplying them takes  $\Theta(n^2)$  time. In this chapter, we shall show how the fast Fourier transform, or FFT, can reduce the time to multiply polynomials to  $\Theta(n \lg n)$ .

The most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the *time domain*: as a function mapping time to amplitude. Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the *frequency domain*. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files. Several fine books delve into the rich area of signal processing; the chapter notes reference a few of them.

### Polynomials

A *polynomial* in the variable  $x$  over an algebraic field  $F$  represents a function  $A(x)$  as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

We call the values  $a_0, a_1, \dots, a_{n-1}$  the *coefficients* of the polynomial. The coefficients are drawn from a field  $F$ , typically the set  $\mathbb{C}$  of complex numbers. A polynomial  $A(x)$  has *degree*  $k$  if its highest nonzero coefficient is  $a_k$ ; we write that  $\text{degree}(A) = k$ . Any integer strictly greater than the degree of a polynomial is a *degree-bound* of that polynomial. Therefore, the degree of a polynomial of degree-bound  $n$  may be any integer between 0 and  $n - 1$ , inclusive.

We can define a variety of operations on polynomials. For *polynomial addition*, if  $A(x)$  and  $B(x)$  are polynomials of degree-bound  $n$ , their *sum* is a poly-

mial  $C(x)$ , also of degree-bound  $n$ , such that  $C(x) = A(x) + B(x)$  for all  $x$  in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

where  $c_j = a_j + b_j$  for  $j = 0, 1, \dots, n-1$ . For example, if we have the polynomials  $A(x) = 6x^3 + 7x^2 - 10x + 9$  and  $B(x) = -2x^3 + 4x - 5$ , then  $C(x) = 4x^3 + 7x^2 - 6x + 4$ .

For **polynomial multiplication**, if  $A(x)$  and  $B(x)$  are polynomials of degree-bound  $n$ , their **product**  $C(x)$  is a polynomial of degree-bound  $2n-1$  such that  $C(x) = A(x)B(x)$  for all  $x$  in the underlying field. You probably have multiplied polynomials before, by multiplying each term in  $A(x)$  by each term in  $B(x)$  and then combining terms with equal powers. For example, we can multiply  $A(x) = 6x^3 + 7x^2 - 10x + 9$  and  $B(x) = -2x^3 + 4x - 5$  as follows:

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ - 2x^3 \qquad \qquad + 4x - 5 \\ \hline - 30x^3 - 35x^2 + 50x - 45 \\ 24x^4 + 28x^3 - 40x^2 + 36x \\ - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

Another way to express the product  $C(x)$  is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j, \tag{30.1}$$

where

$$c_j = \sum_{k=0}^j a_k b_{j-k}. \tag{30.2}$$

Note that  $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$ , implying that if  $A$  is a polynomial of degree-bound  $n_a$  and  $B$  is a polynomial of degree-bound  $n_b$ , then  $C$  is a polynomial of degree-bound  $n_a + n_b - 1$ . Since a polynomial of degree-bound  $k$  is also a polynomial of degree-bound  $k + 1$ , we will normally say that the product polynomial  $C$  is a polynomial of degree-bound  $n_a + n_b$ .

### Chapter outline

Section 30.1 presents two ways to represent polynomials: the coefficient representation and the point-value representation. The straightforward methods for multiplying polynomials—equations (30.1) and (30.2)—take  $\Theta(n^2)$  time when we represent polynomials in coefficient form, but only  $\Theta(n)$  time when we represent them in point-value form. We can, however, multiply polynomials using the coefficient representation in only  $\Theta(n \lg n)$  time by converting between the two representations. To see why this approach works, we must first study complex roots of unity, which we do in Section 30.2. Then, we use the FFT and its inverse, also described in Section 30.2, to perform the conversions. Section 30.3 shows how to implement the FFT quickly in both serial and parallel models.

This chapter uses complex numbers extensively, and within this chapter we use the symbol  $i$  exclusively to denote  $\sqrt{-1}$ .

---

## 30.1 Representing polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent; that is, a polynomial in point-value form has a unique counterpart in coefficient form. In this section, we introduce the two representations and show how to combine them so that we can multiply two degree-bound  $n$  polynomials in  $\Theta(n \lg n)$  time.

### Coefficient representation

A *coefficient representation* of a polynomial  $A(x) = \sum_{j=0}^{n-1} a_j x^j$  of degree-bound  $n$  is a vector of coefficients  $a = (a_0, a_1, \dots, a_{n-1})$ . In matrix equations in this chapter, we shall generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example, the operation of *evaluating* the polynomial  $A(x)$  at a given point  $x_0$  consists of computing the value of  $A(x_0)$ . We can evaluate a polynomial in  $\Theta(n)$  time using *Horner's rule*:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_{n-2} + x_0(a_{n-1}))) \dots).$$

Similarly, adding two polynomials represented by the coefficient vectors  $a = (a_0, a_1, \dots, a_{n-1})$  and  $b = (b_0, b_1, \dots, b_{n-1})$  takes  $\Theta(n)$  time: we just produce the coefficient vector  $c = (c_0, c_1, \dots, c_{n-1})$ , where  $c_j = a_j + b_j$  for  $j = 0, 1, \dots, n-1$ .

Now, consider multiplying two degree-bound  $n$  polynomials  $A(x)$  and  $B(x)$  represented in coefficient form. If we use the method described by equations (30.1) and (30.2), multiplying polynomials takes time  $\Theta(n^2)$ , since we must multiply each coefficient in the vector  $a$  by each coefficient in the vector  $b$ . The operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating a polynomial or adding two polynomials. The resulting coefficient vector  $c$ , given by equation (30.2), is also called the **convolution** of the input vectors  $a$  and  $b$ , denoted  $c = a \otimes b$ . Since multiplying polynomials and computing convolutions are fundamental computational problems of considerable practical importance, this chapter concentrates on efficient algorithms for them.

### Point-value representation

A **point-value representation** of a polynomial  $A(x)$  of degree-bound  $n$  is a set of  $n$  **point-value pairs**

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

such that all of the  $x_k$  are distinct and

$$y_k = A(x_k) \tag{30.3}$$

for  $k = 0, 1, \dots, n-1$ . A polynomial has many different point-value representations, since we can use any set of  $n$  distinct points  $x_0, x_1, \dots, x_{n-1}$  as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all we have to do is select  $n$  distinct points  $x_0, x_1, \dots, x_{n-1}$  and then evaluate  $A(x_k)$  for  $k = 0, 1, \dots, n-1$ . With Horner's method, evaluating a polynomial at  $n$  points takes time  $\Theta(n^2)$ . We shall see later that if we choose the points  $x_k$  cleverly, we can accelerate this computation to run in time  $\Theta(n \lg n)$ .

The inverse of evaluation—determining the coefficient form of a polynomial from a point-value representation—is **interpolation**. The following theorem shows that interpolation is well defined when the desired interpolating polynomial must have a degree-bound equal to the given number of point-value pairs.

#### **Theorem 30.1 (Uniqueness of an interpolating polynomial)**

For any set  $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$  of  $n$  point-value pairs such that all the  $x_k$  values are distinct, there is a unique polynomial  $A(x)$  of degree-bound  $n$  such that  $y_k = A(x_k)$  for  $k = 0, 1, \dots, n-1$ .

**Proof** The proof relies on the existence of the inverse of a certain matrix. Equation (30.3) is equivalent to the matrix equation

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \quad (30.4)$$

The matrix on the left is denoted  $V(x_0, x_1, \dots, x_{n-1})$  and is known as a Vandermonde matrix. By Problem D-1, this matrix has determinant

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j),$$

and therefore, by Theorem D.5, it is invertible (that is, nonsingular) if the  $x_k$  are distinct. Thus, we can solve for the coefficients  $a_j$  uniquely given the point-value representation:

$$a = V(x_0, x_1, \dots, x_{n-1})^{-1} y. \quad \blacksquare$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations. Using the LU decomposition algorithms of Chapter 28, we can solve these equations in time  $O(n^3)$ .

A faster algorithm for  $n$ -point interpolation is based on **Lagrange's formula**:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \quad (30.5)$$

You may wish to verify that the right-hand side of equation (30.5) is a polynomial of degree-bound  $n$  that satisfies  $A(x_k) = y_k$  for all  $k$ . Exercise 30.1-5 asks you how to compute the coefficients of  $A$  using Lagrange's formula in time  $\Theta(n^2)$ .

Thus,  $n$ -point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.<sup>1</sup> The algorithms described above for these problems take time  $\Theta(n^2)$ .

The point-value representation is quite convenient for many operations on polynomials. For addition, if  $C(x) = A(x) + B(x)$ , then  $C(x_k) = A(x_k) + B(x_k)$  for any point  $x_k$ . More precisely, if we have a point-value representation for  $A$ ,

---

<sup>1</sup>Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although the approaches described here are mathematically correct, small differences in the inputs or round-off errors during computation can cause large differences in the result.

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\},$$

and for  $B$ ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$$

(note that  $A$  and  $B$  are evaluated at the *same*  $n$  points), then a point-value representation for  $C$  is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\}.$$

Thus, the time to add two polynomials of degree-bound  $n$  in point-value form is  $\Theta(n)$ .

Similarly, the point-value representation is convenient for multiplying polynomials. If  $C(x) = A(x)B(x)$ , then  $C(x_k) = A(x_k)B(x_k)$  for any point  $x_k$ , and we can pointwise multiply a point-value representation for  $A$  by a point-value representation for  $B$  to obtain a point-value representation for  $C$ . We must face the problem, however, that  $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$ ; if  $A$  and  $B$  are of degree-bound  $n$ , then  $C$  is of degree-bound  $2n$ . A standard point-value representation for  $A$  and  $B$  consists of  $n$  point-value pairs for each polynomial. When we multiply these together, we get  $n$  point-value pairs, but we need  $2n$  pairs to interpolate a unique polynomial  $C$  of degree-bound  $2n$ . (See Exercise 30.1-4.) We must therefore begin with “extended” point-value representations for  $A$  and for  $B$  consisting of  $2n$  point-value pairs each. Given an extended point-value representation for  $A$ ,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\},$$

and a corresponding extended point-value representation for  $B$ ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\},$$

then a point-value representation for  $C$  is

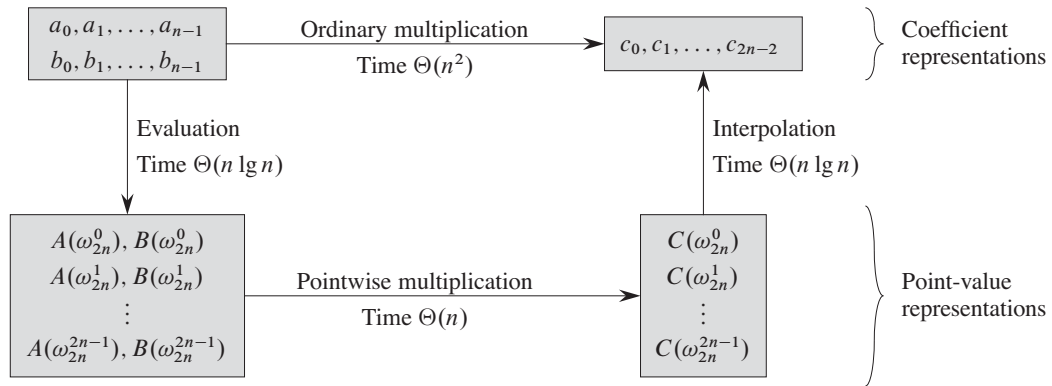
$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}.$$

Given two input polynomials in extended point-value form, we see that the time to multiply them to obtain the point-value form of the result is  $\Theta(n)$ , much less than the time required to multiply polynomials in coefficient form.

Finally, we consider how to evaluate a polynomial given in point-value form at a new point. For this problem, we know of no simpler approach than converting the polynomial to coefficient form first, and then evaluating it at the new point.

### Fast multiplication of polynomials in coefficient form

Can we use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form? The answer hinges



**Figure 30.1** A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The  $\omega_{2n}$  terms are complex  $(2n)$ th roots of unity.

on whether we can convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

We can use any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only  $\Theta(n \lg n)$  time. As we shall see in Section 30.2, if we choose “complex roots of unity” as the evaluation points, we can produce a point-value representation by taking the discrete Fourier transform (or DFT) of a coefficient vector. We can perform the inverse operation, interpolation, by taking the “inverse DFT” of point-value pairs, yielding a coefficient vector. Section 30.2 will show how the FFT accomplishes the DFT and inverse DFT operations in  $\Theta(n \lg n)$  time.

Figure 30.1 shows this strategy graphically. One minor detail concerns degree-bounds. The product of two polynomials of degree-bound  $n$  is a polynomial of degree-bound  $2n$ . Before evaluating the input polynomials  $A$  and  $B$ , therefore, we first double their degree-bounds to  $2n$  by adding  $n$  high-order coefficients of 0. Because the vectors have  $2n$  elements, we use “complex  $(2n)$ th roots of unity,” which are denoted by the  $\omega_{2n}$  terms in Figure 30.1.

Given the FFT, we have the following  $\Theta(n \lg n)$ -time procedure for multiplying two polynomials  $A(x)$  and  $B(x)$  of degree-bound  $n$ , where the input and output representations are in coefficient form. We assume that  $n$  is a power of 2; we can always meet this requirement by adding high-order zero coefficients.

1. *Double degree-bound:* Create coefficient representations of  $A(x)$  and  $B(x)$  as degree-bound  $2n$  polynomials by adding  $n$  high-order zero coefficients to each.

2. *Evaluate:* Compute point-value representations of  $A(x)$  and  $B(x)$  of length  $2n$  by applying the FFT of order  $2n$  on each polynomial. These representations contain the values of the two polynomials at the  $(2n)$ th roots of unity.
3. *Pointwise multiply:* Compute a point-value representation for the polynomial  $C(x) = A(x)B(x)$  by multiplying these values together pointwise. This representation contains the value of  $C(x)$  at each  $(2n)$ th root of unity.
4. *Interpolate:* Create the coefficient representation of the polynomial  $C(x)$  by applying the FFT on  $2n$  point-value pairs to compute the inverse DFT.

Steps (1) and (3) take time  $\Theta(n)$ , and steps (2) and (4) take time  $\Theta(n \lg n)$ . Thus, once we show how to use the FFT, we will have proven the following.

### **Theorem 30.2**

We can multiply two polynomials of degree-bound  $n$  in time  $\Theta(n \lg n)$ , with both the input and output representations in coefficient form. ■

## **Exercises**

### **30.1-1**

Multiply the polynomials  $A(x) = 7x^3 - x^2 + x - 10$  and  $B(x) = 8x^3 - 6x + 3$  using equations (30.1) and (30.2).

### **30.1-2**

Another way to evaluate a polynomial  $A(x)$  of degree-bound  $n$  at a given point  $x_0$  is to divide  $A(x)$  by the polynomial  $(x - x_0)$ , obtaining a quotient polynomial  $q(x)$  of degree-bound  $n - 1$  and a remainder  $r$ , such that

$$A(x) = q(x)(x - x_0) + r .$$

Clearly,  $A(x_0) = r$ . Show how to compute the remainder  $r$  and the coefficients of  $q(x)$  in time  $\Theta(n)$  from  $x_0$  and the coefficients of  $A$ .

### **30.1-3**

Derive a point-value representation for  $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$  from a point-value representation for  $A(x) = \sum_{j=0}^{n-1} a_jx^j$ , assuming that none of the points is 0.

### **30.1-4**

Prove that  $n$  distinct point-value pairs are necessary to uniquely specify a polynomial of degree-bound  $n$ , that is, if fewer than  $n$  distinct point-value pairs are given, they fail to specify a unique polynomial of degree-bound  $n$ . (*Hint:* Using Theorem 30.1, what can you say about a set of  $n - 1$  point-value pairs to which you add one more arbitrarily chosen point-value pair?)



**30.1-5**

Show how to use equation (30.5) to interpolate in time  $\Theta(n^2)$ . (*Hint*: First compute the coefficient representation of the polynomial  $\prod_j (x - x_j)$  and then divide by  $(x - x_k)$  as necessary for the numerator of each term; see Exercise 30.1-2. You can compute each of the  $n$  denominators in time  $O(n)$ .)

**30.1-6**

Explain what is wrong with the “obvious” approach to polynomial division using a point-value representation, i.e., dividing the corresponding  $y$  values. Discuss separately the case in which the division comes out exactly and the case in which it doesn’t.

**30.1-7**

Consider two sets  $A$  and  $B$ , each having  $n$  integers in the range from 0 to  $10n$ . We wish to compute the *Cartesian sum* of  $A$  and  $B$ , defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

Note that the integers in  $C$  are in the range from 0 to  $20n$ . We want to find the elements of  $C$  and the number of times each element of  $C$  is realized as a sum of elements in  $A$  and  $B$ . Show how to solve the problem in  $O(n \lg n)$  time. (*Hint*: Represent  $A$  and  $B$  as polynomials of degree at most  $10n$ .)

**30.2 The DFT and FFT**

In Section 30.1, we claimed that if we use complex roots of unity, we can evaluate and interpolate polynomials in  $\Theta(n \lg n)$  time. In this section, we define complex roots of unity and study their properties, define the DFT, and then show how the FFT computes the DFT and its inverse in  $\Theta(n \lg n)$  time.

**Complex roots of unity**

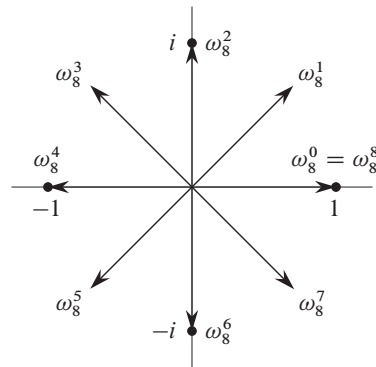
A *complex  $n$ th root of unity* is a complex number  $\omega$  such that

$$\omega^n = 1 .$$

There are exactly  $n$  complex  $n$ th roots of unity:  $e^{2\pi ik/n}$  for  $k = 0, 1, \dots, n - 1$ . To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu} = \cos(u) + i \sin(u) .$$

Figure 30.2 shows that the  $n$  complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value



**Figure 30.2** The values of  $\omega_8^0, \omega_8^1, \dots, \omega_8^7$  in the complex plane, where  $\omega_8 = e^{2\pi i/8}$  is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \quad (30.6)$$

is the *principal  $n$ th root of unity*;<sup>2</sup> all other complex  $n$ th roots of unity are powers of  $\omega_n$ .

The  $n$  complex  $n$ th roots of unity,

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1},$$

form a group under multiplication (see Section 31.3). This group has the same structure as the additive group  $(\mathbb{Z}_n, +)$  modulo  $n$ , since  $\omega_n^n = \omega_n^0 = 1$  implies that  $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$ . Similarly,  $\omega_n^{-1} = \omega_n^{n-1}$ . The following lemmas furnish some essential properties of the complex  $n$ th roots of unity.

**Lemma 30.3 (Cancellation lemma)**

For any integers  $n \geq 0$ ,  $k \geq 0$ , and  $d > 0$ ,

$$\omega_{dn}^{dk} = \omega_n^k. \quad (30.7)$$

**Proof** The lemma follows directly from equation (30.6), since

$$\begin{aligned} \omega_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= \omega_n^k. \end{aligned}$$

■

<sup>2</sup>Many other authors define  $\omega_n$  differently:  $\omega_n = e^{-2\pi i/n}$ . This alternative definition tends to be used for signal-processing applications. The underlying mathematics is substantially the same with either definition of  $\omega_n$ .

**Corollary 30.4**

For any even integer  $n > 0$ ,

$$\omega_n^{n/2} = \omega_2 = -1.$$

**Proof** The proof is left as Exercise 30.2-1. ■

**Lemma 30.5 (Halving lemma)**

If  $n > 0$  is even, then the squares of the  $n$  complex  $n$ th roots of unity are the  $n/2$  complex  $(n/2)$ th roots of unity.

**Proof** By the cancellation lemma, we have  $(\omega_n^k)^2 = \omega_{n/2}^k$ , for any nonnegative integer  $k$ . Note that if we square all of the complex  $n$ th roots of unity, then we obtain each  $(n/2)$ th root of unity exactly twice, since

$$\begin{aligned} (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k} \omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2. \end{aligned}$$

Thus,  $\omega_n^k$  and  $\omega_n^{k+n/2}$  have the same square. We could also have used Corollary 30.4 to prove this property, since  $\omega_n^{n/2} = -1$  implies  $\omega_n^{k+n/2} = -\omega_n^k$ , and thus  $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$ . ■

As we shall see, the halving lemma is essential to our divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

**Lemma 30.6 (Summation lemma)**

For any integer  $n \geq 1$  and nonzero integer  $k$  not divisible by  $n$ ,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

**Proof** Equation (A.5) applies to complex values as well as to reals, and so we have

$$\begin{aligned}
\sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\
&= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\
&= \frac{(1)^k - 1}{\omega_n^k - 1} \\
&= 0.
\end{aligned}$$

Because we require that  $k$  is not divisible by  $n$ , and because  $\omega_n^k = 1$  only when  $k$  is divisible by  $n$ , we ensure that the denominator is not 0. ■

### The DFT

Recall that we wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound  $n$  at  $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$  (that is, at the  $n$  complex  $n$ th roots of unity).<sup>3</sup> We assume that  $A$  is given in coefficient form:  $a = (a_0, a_1, \dots, a_{n-1})$ . Let us define the results  $y_k$ , for  $k = 0, 1, \dots, n-1$ , by

$$\begin{aligned}
y_k &= A(\omega_n^k) \\
&= \sum_{j=0}^{n-1} a_j \omega_n^{kj}.
\end{aligned} \tag{30.8}$$

The vector  $y = (y_0, y_1, \dots, y_{n-1})$  is the **discrete Fourier transform (DFT)** of the coefficient vector  $a = (a_0, a_1, \dots, a_{n-1})$ . We also write  $y = \text{DFT}_n(a)$ .

### The FFT

By using a method known as the **fast Fourier transform (FFT)**, which takes advantage of the special properties of the complex roots of unity, we can compute  $\text{DFT}_n(a)$  in time  $\Theta(n \lg n)$ , as opposed to the  $\Theta(n^2)$  time of the straightforward method. We assume throughout that  $n$  is an exact power of 2. Although strategies

---

<sup>3</sup>The length  $n$  is actually what we referred to as  $2n$  in Section 30.1, since we double the degree-bound of the given polynomials prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex  $(2n)$ th roots of unity.

for dealing with non-power-of-2 sizes are known, they are beyond the scope of this book.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of  $A(x)$  separately to define the two new polynomials  $A^{[0]}(x)$  and  $A^{[1]}(x)$  of degree-bound  $n/2$ :

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1} , \\ A^{[1]}(x) &= a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1} . \end{aligned}$$

Note that  $A^{[0]}$  contains all the even-indexed coefficients of  $A$  (the binary representation of the index ends in 0) and  $A^{[1]}$  contains all the odd-indexed coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) , \quad (30.9)$$

so that the problem of evaluating  $A(x)$  at  $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$  reduces to

1. evaluating the degree-bound  $n/2$  polynomials  $A^{[0]}(x)$  and  $A^{[1]}(x)$  at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2 , \quad (30.10)$$

and then

2. combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of  $n$  distinct values but only of the  $n/2$  complex  $(n/2)$ th roots of unity, with each root occurring exactly twice. Therefore, we recursively evaluate the polynomials  $A^{[0]}$  and  $A^{[1]}$  of degree-bound  $n/2$  at the  $n/2$  complex  $(n/2)$ th roots of unity. These subproblems have exactly the same form as the original problem, but are half the size. We have now successfully divided an  $n$ -element  $\text{DFT}_n$  computation into two  $n/2$ -element  $\text{DFT}_{n/2}$  computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an  $n$ -element vector  $a = (a_0, a_1, \dots, a_{n-1})$ , where  $n$  is a power of 2.

RECURSIVE-FFT( $a$ )

```

1   $n = a.length$            //  $n$  is a power of 2
2  if  $n == 1$ 
3      return  $a$ 
4   $\omega_n = e^{2\pi i/n}$ 
5   $\omega = 1$ 
6   $a^{[0]} = (a_0, a_2, \dots, a_{n-2})$ 
7   $a^{[1]} = (a_1, a_3, \dots, a_{n-1})$ 
8   $y^{[0]} = \text{RECURSIVE-FFT}(a^{[0]})$ 
9   $y^{[1]} = \text{RECURSIVE-FFT}(a^{[1]})$ 
10 for  $k = 0$  to  $n/2 - 1$ 
11      $y_k = y_k^{[0]} + \omega y_k^{[1]}$ 
12      $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$ 
13      $\omega = \omega \omega_n$ 
14 return  $y$            //  $y$  is assumed to be a column vector

```

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis of the recursion; the DFT of one element is the element itself, since in this case

$$\begin{aligned}
 y_0 &= a_0 \omega_1^0 \\
 &= a_0 \cdot 1 \\
 &= a_0 .
 \end{aligned}$$

Lines 6–7 define the coefficient vectors for the polynomials  $A^{[0]}$  and  $A^{[1]}$ . Lines 4, 5, and 13 guarantee that  $\omega$  is updated properly so that whenever lines 11–12 are executed, we have  $\omega = \omega_n^k$ . (Keeping a running value of  $\omega$  from iteration to iteration saves time over computing  $\omega_n^k$  from scratch each time through the **for** loop.) Lines 8–9 perform the recursive  $\text{DFT}_{n/2}$  computations, setting, for  $k = 0, 1, \dots, n/2 - 1$ ,

$$\begin{aligned}
 y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) , \\
 y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) ,
 \end{aligned}$$

or, since  $\omega_{n/2}^k = \omega_n^{2k}$  by the cancellation lemma,

$$\begin{aligned}
 y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) , \\
 y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) .
 \end{aligned}$$

Lines 11–12 combine the results of the recursive DFT<sub>*n*/2</sub> calculations. For  $y_0, y_1, \dots, y_{n/2-1}$ , line 11 yields

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned} \quad (\text{by equation (30.9)}) .$$

For  $y_{n/2}, y_{n/2+1}, \dots, y_{n-1}$ , letting  $k = 0, 1, \dots, n/2 - 1$ , line 12 yields

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && (\text{since } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && (\text{since } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) && (\text{by equation (30.9)}) . \end{aligned}$$

Thus, the vector  $y$  returned by RECURSIVE-FFT is indeed the DFT of the input vector  $a$ .

Lines 11 and 12 multiply each value  $y_k^{[1]}$  by  $\omega_n^k$ , for  $k = 0, 1, \dots, n/2 - 1$ . Line 11 adds this product to  $y_k^{[0]}$ , and line 12 subtracts it. Because we use each factor  $\omega_n^k$  in both its positive and negative forms, we call the factors  $\omega_n^k$  **twiddle factors**.

To determine the running time of procedure RECURSIVE-FFT, we note that exclusive of the recursive calls, each invocation takes time  $\Theta(n)$ , where  $n$  is the length of the input vector. The recurrence for the running time is therefore

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Thus, we can evaluate a polynomial of degree-bound  $n$  at the complex  $n$ th roots of unity in time  $\Theta(n \lg n)$  using the fast Fourier transform.

### Interpolation at the complex roots of unity

We now complete the polynomial multiplication scheme by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product  $y = V_n a$ , where  $V_n$  is a Vandermonde matrix containing the appropriate powers of  $\omega_n$ :

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

The  $(k, j)$  entry of  $V_n$  is  $\omega_n^{kj}$ , for  $j, k = 0, 1, \dots, n-1$ . The exponents of the entries of  $V_n$  form a multiplication table.

For the inverse operation, which we write as  $a = \text{DFT}_n^{-1}(y)$ , we proceed by multiplying  $y$  by the matrix  $V_n^{-1}$ , the inverse of  $V_n$ .

**Theorem 30.7**

For  $j, k = 0, 1, \dots, n-1$ , the  $(j, k)$  entry of  $V_n^{-1}$  is  $\omega_n^{-kj}/n$ .

**Proof** We show that  $V_n^{-1}V_n = I_n$ , the  $n \times n$  identity matrix. Consider the  $(j, j')$  entry of  $V_n^{-1}V_n$ :

$$\begin{aligned} [V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n. \end{aligned}$$

This summation equals 1 if  $j' = j$ , and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that we rely on  $-(n-1) \leq j' - j \leq n-1$ , so that  $j' - j$  is not divisible by  $n$ , in order for the summation lemma to apply. ■

Given the inverse matrix  $V_n^{-1}$ , we have that  $\text{DFT}_n^{-1}(y)$  is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

for  $j = 0, 1, \dots, n-1$ . By comparing equations (30.8) and (30.11), we see that by modifying the FFT algorithm to switch the roles of  $a$  and  $y$ , replace  $\omega_n$  by  $\omega_n^{-1}$ , and divide each element of the result by  $n$ , we compute the inverse DFT (see Exercise 30.2-4). Thus, we can compute  $\text{DFT}_n^{-1}$  in  $\Theta(n \lg n)$  time as well.

We see that, by using the FFT and the inverse FFT, we can transform a polynomial of degree-bound  $n$  back and forth between its coefficient representation and a point-value representation in time  $\Theta(n \lg n)$ . In the context of polynomial multiplication, we have shown the following.



**Theorem 30.8 (Convolution theorem)**

For any two vectors  $a$  and  $b$  of length  $n$ , where  $n$  is a power of 2,

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)),$$

where the vectors  $a$  and  $b$  are padded with 0s to length  $2n$  and  $\cdot$  denotes the componentwise product of two  $2n$ -element vectors. ■

**Exercises****30.2-1**

Prove Corollary 30.4.

**30.2-2**

Compute the DFT of the vector  $(0, 1, 2, 3)$ .

**30.2-3**

Do Exercise 30.1-1 by using the  $\Theta(n \lg n)$ -time scheme.

**30.2-4**

Write pseudocode to compute  $\text{DFT}_n^{-1}$  in  $\Theta(n \lg n)$  time.

**30.2-5**

Describe the generalization of the FFT procedure to the case in which  $n$  is a power of 3. Give a recurrence for the running time, and solve the recurrence.

**30.2-6 ★**

Suppose that instead of performing an  $n$ -element FFT over the field of complex numbers (where  $n$  is even), we use the ring  $\mathbb{Z}_m$  of integers modulo  $m$ , where  $m = 2^{tn/2} + 1$  and  $t$  is an arbitrary positive integer. Use  $\omega = 2^t$  instead of  $\omega_n$  as a principal  $n$ th root of unity, modulo  $m$ . Prove that the DFT and the inverse DFT are well defined in this system.

**30.2-7**

Given a list of values  $z_0, z_1, \dots, z_{n-1}$  (possibly with repetitions), show how to find the coefficients of a polynomial  $P(x)$  of degree-bound  $n + 1$  that has zeros only at  $z_0, z_1, \dots, z_{n-1}$  (possibly with repetitions). Your procedure should run in time  $O(n \lg^2 n)$ . (*Hint*: The polynomial  $P(x)$  has a zero at  $z_j$  if and only if  $P(x)$  is a multiple of  $(x - z_j)$ .)

**30.2-8 ★**

The *chirp transform* of a vector  $a = (a_0, a_1, \dots, a_{n-1})$  is the vector  $y = (y_0, y_1, \dots, y_{n-1})$ , where  $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$  and  $z$  is any complex number. The

DFT is therefore a special case of the chirp transform, obtained by taking  $z = \omega_n$ . Show how to evaluate the chirp transform in time  $O(n \lg n)$  for any complex number  $z$ . (*Hint*: Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left( a_j z^{j^2/2} \right) \left( z^{-(k-j)^2/2} \right)$$

to view the chirp transform as a convolution.)

### 30.3 Efficient FFT implementations

Since the practical applications of the DFT, such as signal processing, demand the utmost speed, this section examines two efficient FFT implementations. First, we shall examine an iterative version of the FFT algorithm that runs in  $\Theta(n \lg n)$  time but can have a lower constant hidden in the  $\Theta$ -notation than the recursive version in Section 30.2. (Depending on the exact implementation, the recursive version may use the hardware cache more efficiently.) Then, we shall use the insights that led us to the iterative implementation to design an efficient parallel FFT circuit.

#### An iterative FFT implementation

We first note that the **for** loop of lines 10–13 of RECURSIVE-FFT involves computing the value  $\omega_n^k y_k^{[1]}$  twice. In compiler terminology, we call such a value a *common subexpression*. We can change the loop to compute it only once, storing it in a temporary variable  $t$ .

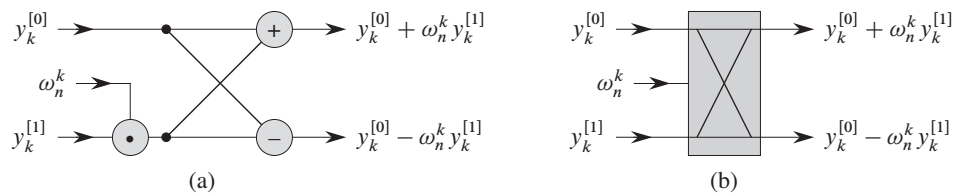
```

for  $k = 0$  to  $n/2 - 1$ 
     $t = \omega y_k^{[1]}$ 
     $y_k = y_k^{[0]} + t$ 
     $y_{k+(n/2)} = y_k^{[0]} - t$ 
     $\omega = \omega \omega_n$ 

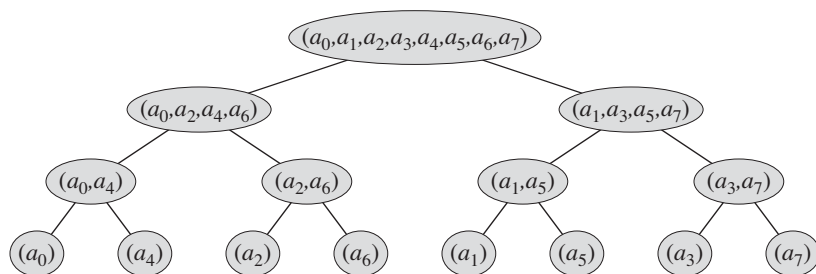
```

The operation in this loop, multiplying the twiddle factor  $\omega = \omega_n^k$  by  $y_k^{[1]}$ , storing the product into  $t$ , and adding and subtracting  $t$  from  $y_k^{[0]}$ , is known as a *butterfly operation* and is shown schematically in Figure 30.3.

We now show how to make the FFT algorithm iterative rather than recursive in structure. In Figure 30.4, we have arranged the input vectors to the recursive calls in an invocation of RECURSIVE-FFT in a tree structure, where the initial call is for  $n = 8$ . The tree has one node for each call of the procedure, labeled



**Figure 30.3** A butterfly operation. (a) The two input values enter from the left, the twiddle factor  $\omega_n^k$  is multiplied by  $y_k^{[1]}$ , and the sum and difference are output on the right. (b) A simplified drawing of a butterfly operation. We will use this representation in a parallel FFT circuit.



**Figure 30.4** The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for  $n = 8$ .

by the corresponding input vector. Each RECURSIVE-FFT invocation makes two recursive calls, unless it has received a 1-element vector. The first call appears in the left child, and the second call appears in the right child.

Looking at the tree, we observe that if we could arrange the elements of the initial vector  $a$  into the order in which they appear in the leaves, we could trace the execution of the RECURSIVE-FFT procedure, but bottom up instead of top down. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds  $n/2$  2-element DFTs. Next, we take these  $n/2$  DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFTs with one 4-element DFT. The vector then holds  $n/4$  4-element DFTs. We continue in this manner until the vector holds two  $(n/2)$ -element DFTs, which we combine using  $n/2$  butterfly operations into the final  $n$ -element DFT.

To turn this bottom-up approach into code, we use an array  $A[0..n-1]$  that initially holds the elements of the input vector  $a$  in the order in which they appear

in the leaves of the tree of Figure 30.4. (We shall show later how to determine this order, which is known as a bit-reversal permutation.) Because we have to combine DFTs on each level of the tree, we introduce a variable  $s$  to count the levels, ranging from 1 (at the bottom, when we are combining pairs to form 2-element DFTs) to  $\lg n$  (at the top, when we are combining two  $(n/2)$ -element DFTs to produce the final result). The algorithm therefore has the following structure:

```

1  for  $s = 1$  to  $\lg n$ 
2      for  $k = 0$  to  $n - 1$  by  $2^s$ 
3          combine the two  $2^{s-1}$ -element DFTs in
               $A[k \dots k + 2^{s-1} - 1]$  and  $A[k + 2^{s-1} \dots k + 2^s - 1]$ 
              into one  $2^s$ -element DFT in  $A[k \dots k + 2^s - 1]$ 

```

We can express the body of the loop (line 3) as more precise pseudocode. We copy the **for** loop from the RECURSIVE-FFT procedure, identifying  $y^{[0]}$  with  $A[k \dots k + 2^{s-1} - 1]$  and  $y^{[1]}$  with  $A[k + 2^{s-1} \dots k + 2^s - 1]$ . The twiddle factor used in each butterfly operation depends on the value of  $s$ ; it is a power of  $\omega_m$ , where  $m = 2^s$ . (We introduce the variable  $m$  solely for the sake of readability.) We introduce another temporary variable  $u$  that allows us to perform the butterfly operation in place. When we replace line 3 of the overall structure by the loop body, we get the following pseudocode, which forms the basis of the parallel implementation we shall present later. The code first calls the auxiliary procedure BIT-REVERSE-COPY( $a, A$ ) to copy vector  $a$  into array  $A$  in the initial order in which we need the values.

ITERATIVE-FFT( $a$ )

```

1  BIT-REVERSE-COPY( $a, A$ )
2   $n = a.length$            //  $n$  is a power of 2
3  for  $s = 1$  to  $\lg n$ 
4       $m = 2^s$ 
5       $\omega_m = e^{2\pi i/m}$ 
6      for  $k = 0$  to  $n - 1$  by  $m$ 
7           $\omega = 1$ 
8          for  $j = 0$  to  $m/2 - 1$ 
9               $t = \omega A[k + j + m/2]$ 
10              $u = A[k + j]$ 
11              $A[k + j] = u + t$ 
12              $A[k + j + m/2] = u - t$ 
13              $\omega = \omega \omega_m$ 
14  return  $A$ 

```

How does BIT-REVERSE-COPY get the elements of the input vector  $a$  into the desired order in the array  $A$ ? The order in which the leaves appear in Figure 30.4

is a **bit-reversal permutation**. That is, if we let  $\text{rev}(k)$  be the  $\lg n$ -bit integer formed by reversing the bits of the binary representation of  $k$ , then we want to place vector element  $a_k$  in array position  $A[\text{rev}(k)]$ . In Figure 30.4, for example, the leaves appear in the order 0, 4, 2, 6, 1, 5, 3, 7; this sequence in binary is 000, 100, 010, 110, 001, 101, 011, 111, and when we reverse the bits of each value we get the sequence 000, 001, 010, 011, 100, 101, 110, 111. To see that we want a bit-reversal permutation in general, we note that at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree. Stripping off the low-order bit at each level, we continue this process down the tree, until we get the order given by the bit-reversal permutation at the leaves.

Since we can easily compute the function  $\text{rev}(k)$ , the BIT-REVERSE-COPY procedure is simple:

BIT-REVERSE-COPY( $a, A$ )

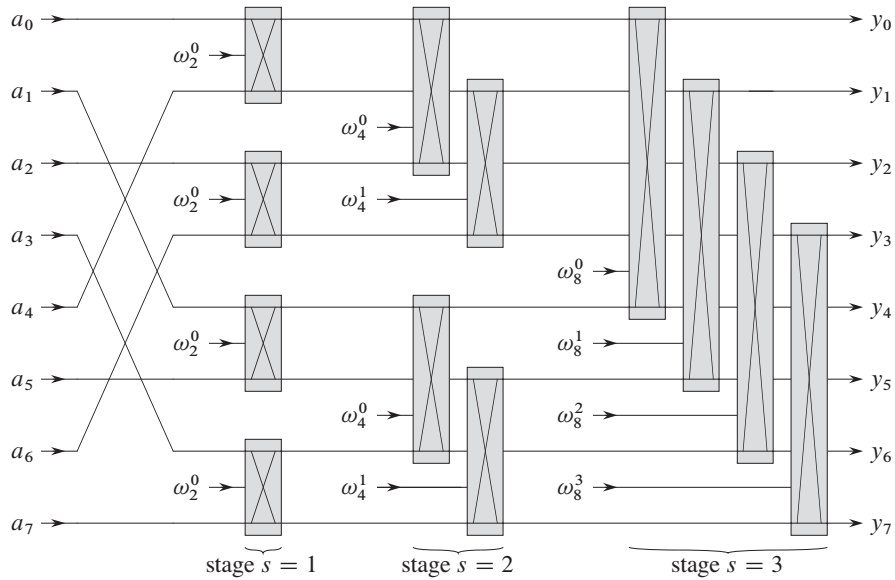
```

1   $n = a.length$ 
2  for  $k = 0$  to  $n - 1$ 
3       $A[\text{rev}(k)] = a_k$ 

```

The iterative FFT implementation runs in time  $\Theta(n \lg n)$ . The call to BIT-REVERSE-COPY( $a, A$ ) certainly runs in  $O(n \lg n)$  time, since we iterate  $n$  times and can reverse an integer between 0 and  $n - 1$ , with  $\lg n$  bits, in  $O(\lg n)$  time. (In practice, because we usually know the initial value of  $n$  in advance, we would probably code a table mapping  $k$  to  $\text{rev}(k)$ , making BIT-REVERSE-COPY run in  $\Theta(n)$  time with a low hidden constant. Alternatively, we could use the clever amortized reverse binary counter scheme described in Problem 17-1.) To complete the proof that ITERATIVE-FFT runs in time  $\Theta(n \lg n)$ , we show that  $L(n)$ , the number of times the body of the innermost loop (lines 8–13) executes, is  $\Theta(n \lg n)$ . The **for** loop of lines 6–13 iterates  $n/m = n/2^s$  times for each value of  $s$ , and the innermost loop of lines 8–13 iterates  $m/2 = 2^{s-1}$  times. Thus,

$$\begin{aligned}
 L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
 &= \sum_{s=1}^{\lg n} \frac{n}{2} \\
 &= \Theta(n \lg n).
 \end{aligned}$$



**Figure 30.5** A circuit that computes the FFT in parallel, here shown on  $n = 8$  inputs. Each butterfly operation takes as input the values on two wires, along with a twiddle factor, and it produces as outputs the values on two wires. The stages of butterflies are labeled to correspond to iterations of the outermost loop of the ITERATIVE-FFT procedure. Only the top and bottom wires passing through a butterfly interact with it; wires that pass through the middle of a butterfly do not affect that butterfly, nor are their values changed by that butterfly. For example, the top butterfly in stage 2 has nothing to do with wire 1 (the wire whose output is labeled  $y_1$ ); its inputs and outputs are only on wires 0 and 2 (labeled  $y_0$  and  $y_2$ , respectively). This circuit has depth  $\Theta(\lg n)$  and performs  $\Theta(n \lg n)$  butterfly operations altogether.

### A parallel FFT circuit

We can exploit many of the properties that allowed us to implement an efficient iterative FFT algorithm to produce an efficient parallel algorithm for the FFT. We will express the parallel FFT algorithm as a circuit. Figure 30.5 shows a parallel FFT circuit, which computes the FFT on  $n$  inputs, for  $n = 8$ . The circuit begins with a bit-reverse permutation of the inputs, followed by  $\lg n$  stages, each stage consisting of  $n/2$  butterflies executed in parallel. The *depth* of the circuit—the maximum number of computational elements between any output and any input that can reach it—is therefore  $\Theta(\lg n)$ .

The leftmost part of the parallel FFT circuit performs the bit-reverse permutation, and the remainder mimics the iterative ITERATIVE-FFT procedure. Because each iteration of the outermost **for** loop performs  $n/2$  independent butterfly operations, the circuit performs them in parallel. The value of  $s$  in each iteration within

ITERATIVE-FFT corresponds to a stage of butterflies shown in Figure 30.5. For  $s = 1, 2, \dots, \lg n$ , stage  $s$  consists of  $n/2^s$  groups of butterflies (corresponding to each value of  $k$  in ITERATIVE-FFT), with  $2^{s-1}$  butterflies per group (corresponding to each value of  $j$  in ITERATIVE-FFT). The butterflies shown in Figure 30.5 correspond to the butterfly operations of the innermost loop (lines 9–12 of ITERATIVE-FFT). Note also that the twiddle factors used in the butterflies correspond to those used in ITERATIVE-FFT: in stage  $s$ , we use  $\omega_m^0, \omega_m^1, \dots, \omega_m^{m/2-1}$ , where  $m = 2^s$ .

### Exercises

#### 30.3-1

Show how ITERATIVE-FFT computes the DFT of the input vector  $(0, 2, 3, -1, 4, 5, 7, 9)$ .

#### 30.3-2

Show how to implement an FFT algorithm with the bit-reversal permutation occurring at the end, rather than at the beginning, of the computation. (*Hint*: Consider the inverse DFT.)

#### 30.3-3

How many times does ITERATIVE-FFT compute twiddle factors in each stage? Rewrite ITERATIVE-FFT to compute twiddle factors only  $2^{s-1}$  times in stage  $s$ .

#### 30.3-4 ★

Suppose that the adders within the butterfly operations of the FFT circuit sometimes fail in such a manner that they always produce a zero output, independent of their inputs. Suppose that exactly one adder has failed, but that you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. How efficient is your method?

---

## Problems

### 30-1 Divide-and-conquer multiplication

- Show how to multiply two linear polynomials  $ax + b$  and  $cx + d$  using only three multiplications. (*Hint*: One of the multiplications is  $(a + b) \cdot (c + d)$ .)
- Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound  $n$  in  $\Theta(n^{\lg 3})$  time. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

- c. Show how to multiply two  $n$ -bit integers in  $O(n^{\lg 3})$  steps, where each step operates on at most a constant number of 1-bit values.

### 30-2 Toeplitz matrices

A **Toeplitz matrix** is an  $n \times n$  matrix  $A = (a_{ij})$  such that  $a_{ij} = a_{i-1, j-1}$  for  $i = 2, 3, \dots, n$  and  $j = 2, 3, \dots, n$ .

- a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?
- b. Describe how to represent a Toeplitz matrix so that you can add two  $n \times n$  Toeplitz matrices in  $O(n)$  time.
- c. Give an  $O(n \lg n)$ -time algorithm for multiplying an  $n \times n$  Toeplitz matrix by a vector of length  $n$ . Use your representation from part (b).
- d. Give an efficient algorithm for multiplying two  $n \times n$  Toeplitz matrices. Analyze its running time.

### 30-3 Multidimensional fast Fourier transform

We can generalize the 1-dimensional discrete Fourier transform defined by equation (30.8) to  $d$  dimensions. The input is a  $d$ -dimensional array  $A = (a_{j_1, j_2, \dots, j_d})$  whose dimensions are  $n_1, n_2, \dots, n_d$ , where  $n_1 n_2 \cdots n_d = n$ . We define the  $d$ -dimensional discrete Fourier transform by the equation

$$y_{k_1, k_2, \dots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \dots, j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

for  $0 \leq k_1 < n_1, 0 \leq k_2 < n_2, \dots, 0 \leq k_d < n_d$ .

- a. Show that we can compute a  $d$ -dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, we first compute  $n/n_1$  separate 1-dimensional DFTs along dimension 1. Then, using the result of the DFTs along dimension 1 as the input, we compute  $n/n_2$  separate 1-dimensional DFTs along dimension 2. Using this result as the input, we compute  $n/n_3$  separate 1-dimensional DFTs along dimension 3, and so on, through dimension  $d$ .
- b. Show that the ordering of dimensions does not matter, so that we can compute a  $d$ -dimensional DFT by computing the 1-dimensional DFTs in any order of the  $d$  dimensions.



- c. Show that if we compute each 1-dimensional DFT by computing the fast Fourier transform, the total time to compute a  $d$ -dimensional DFT is  $O(n \lg n)$ , independent of  $d$ .

### 30-4 Evaluating all derivatives of a polynomial at a point

Given a polynomial  $A(x)$  of degree-bound  $n$ , we define its  $t$ th derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0, \\ \frac{d}{dx}A^{(t-1)}(x) & \text{if } 1 \leq t \leq n-1, \\ 0 & \text{if } t \geq n. \end{cases}$$

From the coefficient representation  $(a_0, a_1, \dots, a_{n-1})$  of  $A(x)$  and a given point  $x_0$ , we wish to determine  $A^{(t)}(x_0)$  for  $t = 0, 1, \dots, n-1$ .

- a. Given coefficients  $b_0, b_1, \dots, b_{n-1}$  such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

show how to compute  $A^{(t)}(x_0)$ , for  $t = 0, 1, \dots, n-1$ , in  $O(n)$  time.

- b. Explain how to find  $b_0, b_1, \dots, b_{n-1}$  in  $O(n \lg n)$  time, given  $A(x_0 + \omega_n^k)$  for  $k = 0, 1, \dots, n-1$ .
- c. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left( \frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

where  $f(j) = a_j \cdot j!$  and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \leq l \leq 0, \\ 0 & \text{if } 1 \leq l \leq n-1. \end{cases}$$

- d. Explain how to evaluate  $A(x_0 + \omega_n^k)$  for  $k = 0, 1, \dots, n-1$  in  $O(n \lg n)$  time. Conclude that we can evaluate all nontrivial derivatives of  $A(x)$  at  $x_0$  in  $O(n \lg n)$  time.

**30-5 Polynomial evaluation at multiple points**

We have seen how to evaluate a polynomial of degree-bound  $n$  at a single point in  $O(n)$  time using Horner's rule. We have also discovered how to evaluate such a polynomial at all  $n$  complex roots of unity in  $O(n \lg n)$  time using the FFT. We shall now show how to evaluate a polynomial of degree-bound  $n$  at  $n$  arbitrary points in  $O(n \lg^2 n)$  time.

To do so, we shall assume that we can compute the polynomial remainder when one such polynomial is divided by another in  $O(n \lg n)$  time, a result that we state without proof. For example, the remainder of  $3x^3 + x^2 - 3x + 1$  when divided by  $x^2 + x + 2$  is

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5.$$

Given the coefficient representation of a polynomial  $A(x) = \sum_{k=0}^{n-1} a_k x^k$  and  $n$  points  $x_0, x_1, \dots, x_{n-1}$ , we wish to compute the  $n$  values  $A(x_0), A(x_1), \dots, A(x_{n-1})$ . For  $0 \leq i \leq j \leq n-1$ , define the polynomials  $P_{ij}(x) = \prod_{k=i}^j (x - x_k)$  and  $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$ . Note that  $Q_{ij}(x)$  has degree at most  $j - i$ .

- a. Prove that  $A(x) \bmod (x - z) = A(z)$  for any point  $z$ .
- b. Prove that  $Q_{kk}(x) = A(x_k)$  and that  $Q_{0,n-1}(x) = A(x)$ .
- c. Prove that for  $i \leq k \leq j$ , we have  $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$  and  $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$ .
- d. Give an  $O(n \lg^2 n)$ -time algorithm to evaluate  $A(x_0), A(x_1), \dots, A(x_{n-1})$ .

**30-6 FFT using modular arithmetic**

As defined, the discrete Fourier transform requires us to compute with complex numbers, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and by using a variant of the FFT based on modular arithmetic, we can guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 gives one approach, using a modulus of length  $\Omega(n)$  bits to handle a DFT on  $n$  points. This problem gives another approach, which uses a modulus of the more reasonable length  $O(\lg n)$ ; it requires that you understand the material of Chapter 31. Let  $n$  be a power of 2.

- a. Suppose that we search for the smallest  $k$  such that  $p = kn + 1$  is prime. Give a simple heuristic argument why we might expect  $k$  to be approximately  $\ln n$ . (The value of  $k$  might be much larger or smaller, but we can reasonably expect to examine  $O(\lg n)$  candidate values of  $k$  on average.) How does the expected length of  $p$  compare to the length of  $n$ ?

Let  $g$  be a generator of  $\mathbb{Z}_p^*$ , and let  $w = g^k \bmod p$ .

- b.* Argue that the DFT and the inverse DFT are well-defined inverse operations modulo  $p$ , where  $w$  is used as a principal  $n$ th root of unity.
- c.* Show how to make the FFT and its inverse work modulo  $p$  in time  $O(n \lg n)$ , where operations on words of  $O(\lg n)$  bits take unit time. Assume that the algorithm is given  $p$  and  $w$ .
- d.* Compute the DFT modulo  $p = 17$  of the vector  $(0, 5, 3, 7, 7, 2, 1, 6)$ . Note that  $g = 3$  is a generator of  $\mathbb{Z}_{17}^*$ .

## Chapter notes

Van Loan's book [343] provides an outstanding treatment of the fast Fourier transform. Press, Teukolsky, Vetterling, and Flannery [283, 284] have a good description of the fast Fourier transform and its applications. For an excellent introduction to signal processing, a popular FFT application area, see the texts by Oppenheim and Schaffer [266] and Oppenheim and Willsky [267]. The Oppenheim and Schaffer book also shows how to handle cases in which  $n$  is not an integer power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in 2 or more dimensions. The books by Gonzalez and Woods [146] and Pratt [281] discuss multidimensional Fourier transforms and their use in image processing, and books by Tolimieri, An, and Lu [338] and Van Loan [343] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [76] are widely credited with devising the FFT in the 1960s. The FFT had in fact been discovered many times previously, but its importance was not fully realized before the advent of modern digital computers. Although Press, Teukolsky, Vetterling, and Flannery attribute the origins of the method to Runge and König in 1924, an article by Heideman, Johnson, and Burrus [163] traces the history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [117] developed a fast and flexible implementation of the FFT, called FFTW ("fastest Fourier transform in the West"). FFTW is designed for situations requiring multiple DFT computations on the same problem size. Before actually computing the DFTs, FFTW executes a "planner," which, by a series of trial runs, determines how best to decompose the FFT computation for the given problem size on the host machine. FFTW adapts to use the hardware cache efficiently, and once subproblems are small enough, FFTW solves them with optimized, straight-line code. Furthermore, FFTW has the unusual advantage of taking  $\Theta(n \lg n)$  time for any problem size  $n$ , even when  $n$  is a large prime.

Although the standard Fourier transform assumes that the input represents points that are uniformly spaced in the time domain, other techniques can approximate the FFT on “nonequispaced” data. The article by Ware [348] provides an overview.