30 Polynomials and the FFT

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***. Signal processing is a rich area for which there are several fine books; the chapter notes reference a few of them.

Polynomials

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

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

We call the values $a_0, a_1, \ldots, a_{n-1}$ the ***coefficients*** of the polynomial. The coefficients are drawn from a field F, typically the set $\mathbf{C}$ of complex numbers. A polynomial $A(x)$ is said to have ***degree*** k if its highest nonzero coefficient is 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.

There are a variety of operations we might wish to define for polynomials. For ***polynomial addition***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, we say that their ***sum*** is a polynomial $C(x)$, 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 $j = 0, 1, \ldots, n-1$. 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, we say that 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 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:

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

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 degree(C) = degree(A) + degree(B), implying

$$\begin{aligned}
\text{degree-bound}(C) &= \text{degree-bound}(A) + \text{degree-bound}(B) - 1 \\
&\le \text{degree-bound}(A) + \text{degree-bound}(B) .
\end{aligned}$$

We shall nevertheless speak of the degree-bound of C as being the sum of the degree-bounds of A and B, since if a polynomial has degree-bound k it also has degree-bound $k+1$.

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 the polynomials are represented in coefficient form, but only $\Theta(n)$ time when they are represented 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 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 the symbol i will be used exclusively to denote $\sqrt{-1}$.

30.1 Representation of 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 they can be combined to allow multiplication of 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 $a = (a_0, a_1, \ldots, a_{n-1})$. 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 $A(x_0)$. Evaluation takes time $\Theta(n)$ using ***Horner's rule***:

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

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

Now, consider the multiplication of 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), polynomial multiplication takes time $\Theta(n^2)$, since each

coefficient in the vector a must be multiplied 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***

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

such that all of the x_k are distinct and

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

for $k = 0, 1, \ldots, n-1$. A polynomial has many different point-value representations, since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be used 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 $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$. With Horner's method, this n-point evaluation takes time $\Theta(n^2)$. We shall see later that if we choose the x_k cleverly, this computation can be accelerated 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 called ***interpolation***. The following theorem shows that interpolation is well defined, assuming that the degree-bound of the interpolating polynomial equals the number of given point-value pairs.

Theorem 30.1 (Uniqueness of an interpolating polynomial)
For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ 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 $k = 0, 1, \ldots, n-1$.

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

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

The matrix on the left is denoted $V(x_0, x_1, \ldots, x_{n-1})$ and is known as a Vandermonde matrix. By Exercise 28.1-11, this matrix has determinant

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

and therefore, by Theorem 28.5, it is invertible (that is, nonsingular) if the x_k are distinct. Thus, the coefficients a_j can be solved for uniquely given the point-value representation:

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

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***:

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

You may wish to verify that the right-hand side of equation (30.5) is a polynomial of degree-bound n that satisfies $A(x_i) = y_i$ for all i. 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.[1] 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,

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

[1] 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.

and for B,

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

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

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

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 the degree-bound of C is the sum of the degree-bounds for A and B. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. Multiplying these together gives us n point-value pairs for C, but since the degree-bound of C is $2n$, we need $2n$ point-value pairs for a point-value representation of C. (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,

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

and a corresponding extended point-value representation for B,

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

then a point-value representation for C is

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

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, there is apparently no approach that is simpler 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 on our ability to convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice-versa (interpolate).

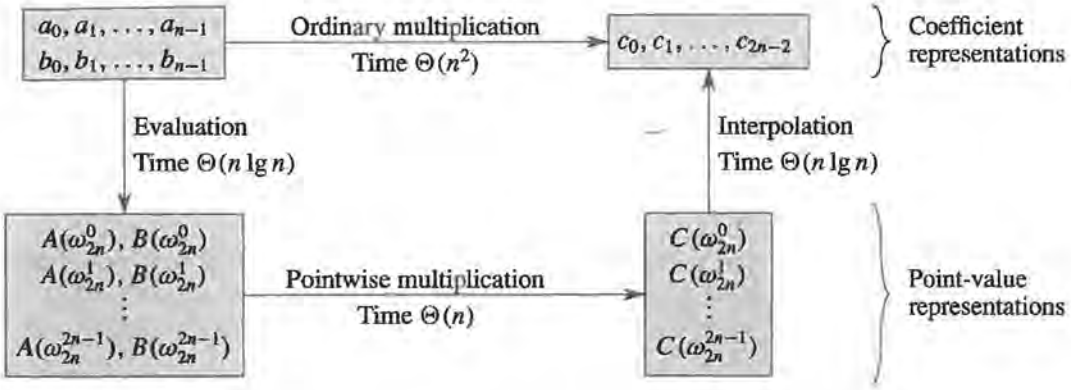

Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots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

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. The inverse operation, interpolation, can be performed by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector. Section 30.2 will show how the FFT performs 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 ω_{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; this requirement can always be met 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 $2n$ through two applications of the FFT of order $2n$. 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 $C(x)$ through a single application of an 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
The product of two polynomials of degree-bound n can be computed 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
Evaluating a polynomial $A(x)$ of degree-bound n at a given point x_0 can also be done by dividing $A(x)$ by the polynomial $(x - x_0)$ to obtain 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 $A(x) = \sum_{j=0}^{n-1} a_j x^j$, 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. Each of the n denominators can be computed 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 that the problem can be solved 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 just $\Theta(n \lg n)$ time.

Complex roots of unity

A ***complex nth root of unity*** is a complex number ω such that

$$\omega^n = 1 .$$

There are exactly n complex nth roots of unity: $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$. 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

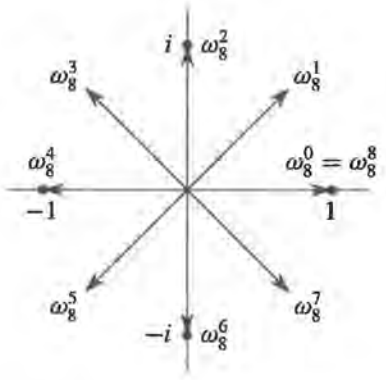

Figure 30.2 The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

is called ***the principal nth root of unity***; all of the other complex nth roots of unity are powers of ω_n.[2]

The n complex nth roots of unity,

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

form a group under multiplication (see Section 31.3). This group has the same structure as the additive group $(\mathbf{Z}_n, +)$ 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, $\omega_n^{-1} = \omega_n^{n-1}$. Essential properties of the complex nth roots of unity are given in the following lemmas.

Lemma 30.3 (Cancellation lemma)
For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k . \tag{30.7}$$

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

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

■

[2]Many other authors define ω_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 ω_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 nth 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 nth roots of unity, then each $(n/2)$th root of unity is obtained 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, ω_n^k and $\omega_n^{k+n/2}$ have the same square. This property can also be proved using Corollary 30.4, 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 nonnegative 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

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

Requiring that k not be divisible by n ensures that the denominator is not 0, since $\omega_n^k = 1$ only when k is divisible by n. ∎

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 $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (that is, at the n complex nth roots of unity).[3] Without loss of generality, we assume that n is a power of 2, since a given degree-bound can always be raised—we can always add new high-order zero coefficients as necessary.[4] We assume that A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, 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 $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***Discrete Fourier Transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$. We also write $y = \mathrm{DFT}_n(a)$.

---

[3] 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.

[4] When using the FFT for signal processing, padding with zero coefficients to arrive at a power-of-2 size is generally ill-advised, as it tends to introduce high-frequency artifacts. One technique used to pad to a power-of-2 size in signal processing is ***mirroring***. Letting n' be the smallest integer power of 2 greater than n, one way to mirror sets $a_{n+j} = a_{n-j-2}$ for $j = 0, 1, \ldots, n' - n - 1$.

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 $\mathrm{DFT}_n(a)$ in time $\Theta(n \lg n)$, as opposed to the $\Theta(n^2)$ time of the straightforward method.

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

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

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

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

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

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

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \tag{30.10}$$

   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, the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated 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 DFT_n computation into two $n/2$-element $\mathrm{DFT}_{n/2}$ computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2.

```
RECURSIVE-FFT(a)
 1  n ← length[a]                ▷ n is a power of 2.
 2  if n = 1
 3     then return a
 4  ω_n ← e^{2πi/n}
 5  ω ← 1
 6  a^[0] ← (a_0, a_2, ..., a_{n-2})
 7  a^[1] ← (a_1, a_3, ..., a_{n-1})
 8  y^[0] ← RECURSIVE-FFT(a^[0])
 9  y^[1] ← RECURSIVE-FFT(a^[1])
10  for k ← 0 to n/2 − 1
11        do y_k ← y_k^[0] + ω y_k^[1]
12           y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
13           ω ← ω ω_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 ω is updated properly so that whenever lines 11–12 are executed, $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves time over computing ω_n^k from scratch each time through the **for** loop.) Lines 8–9 perform the recursive $\mathrm{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \ldots, n/2 - 1$,

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

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 $\mathrm{DFT}_{n/2}$ calculations. For $y_0, y_1, \ldots, y_{n/2-1}$, line 11 yields

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

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

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

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

Within the **for** loop of lines 10–13, each value $y_k^{[1]}$ is multiplied by ω_n^k, for $k = 0, 1, \ldots, n/2 - 1$. The product is both added to and subtracted from $y_k^{[0]}$. Because each factor ω_n^k is used in both its positive and negative forms, the factors ω_n^k are known as ***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 nth 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 ω_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 ω_n^{kj}, for $j, k = 0, 1, \ldots, n - 1$, and the exponents of the entries of V_n form a multiplication table.

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

Theorem 30.7
For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_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 $-(n-1) < j' - j < n-1$, 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 $\mathrm{DFT}_n^{-1}(y)$ is given by

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

for $j = 0, 1, \ldots, n-1$. 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 ω_n by ω_n^{-1}, and divide each element of the result by n, we compute the inverse DFT (see Exercise 30.2-4). Thus, DFT_n^{-1} can be computed in $\Theta(n \lg n)$ time as well.

Thus, 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,

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

where the vectors a and b are padded with 0's 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 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 $\mathbf{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. Use $\omega = 2^t$ instead of ω_n as a principal nth 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 $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree-bound $n + 1$ that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (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 $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, 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 $z = \omega_n$. Prove that the chirp transform can be evaluated 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 has a lower constant hidden in the Θ-notation than the recursive implementation in Section 30.2. 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, this value is known as 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
    do t ← ω y_k^[1]
       y_k ← y_k^[0] + t
       y_{k+(n/2)} ← y_k^[0] − t
       ω ← ω ω_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 by the corresponding input vector. Each RECURSIVE-FFT invocation makes two recursive calls, unless it has received a 1-element vector. We make the first call the left child and the second call 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 mimic the execution of the RECURSIVE-FFT procedure as follows. 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 DFT's. Next, we take these $n/2$ DFT's in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFT's with one 4-element DFT. The vector then holds $n/4$ 4-element DFT's. We

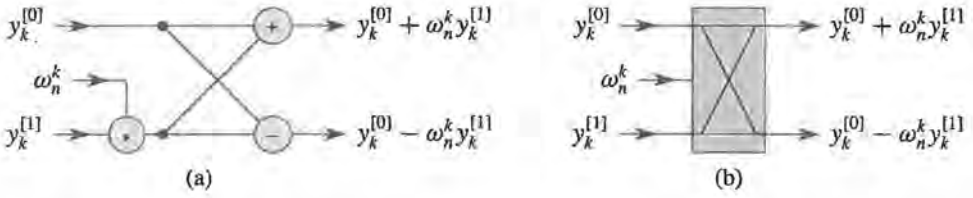

Figure 30.3 A butterfly operation. (a) The two input values enter from the left, the twiddle factor ω_n^k is multiplied by $y_k^{[1]}$, and the sum and difference are output on the right. (b) A simplified drawing of a butterfly operation. We will use this representation in a parallel FFT circuit.

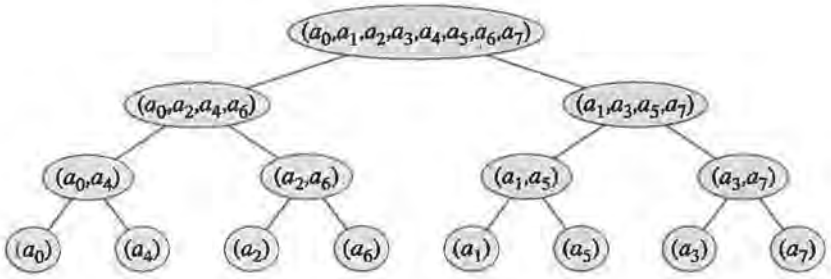

Figure 30.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for $n = 8$.

continue in this manner until the vector holds two $(n/2)$-element DFT's, which we can combine using $n/2$ butterfly operations into the final n-element DFT.

To turn this observation 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 the combining has to be done 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 DFT's) to $\lg n$ (at the top, when we are combining two $(n/2)$-element DFT's to produce the final result). The algorithm therefore has the following structure:

```
1  for s ← 1 to lg n
2      do for k ← 0 to n − 1 by 2^s
3             do combine the two 2^(s−1)-element DFT's in
                    A[k..k + 2^(s−1) − 1] and A[k + 2^(s−1)..k + 2^s − 1]
                    into one 2^s-element DFT in A[k..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 $A[k\,..\,k + 2^{s-1} - 1]$ and $y^{[1]}$ with $A[k + 2^{s-1}\,..\,k + 2^s - 1]$. The twiddle factor used in each butterfly operation depends on the value of s; it is a power of ω_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 ← length[a]          ▷ n is a power of 2.
 3  for s ← 1 to lg n
 4      do m ← 2^s
 5         ω_m ← e^{2πi/m}
 6         for k ← 0 to n − 1 by m
 7             do ω ← 1
 8                for j ← 0 to m/2 − 1
 9                    do t ← ω 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                       ω ← ω ω_m
```

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 $\mathrm{rev}(k)$ 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 $A[\mathrm{rev}(k)]$. 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 are placed in the left subtree and indices whose low-order bit is 1 are placed in 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 the function $\mathrm{rev}(k)$ is easily computed, the BIT-REVERSE-COPY procedure can be written as follows.

```
BIT-REVERSE-COPY(a, A)
1  n ← length[a]
2  for k ← 0 to n − 1
3       do A[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, we usually know the initial value of n in advance, so 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) is executed, 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,

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

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 that looks much like the comparison networks of Chapter 27. Instead of comparators, the FFT circuit uses butterfly operations, as drawn in Figure 30.3(b). The notion of depth that we developed in Chapter 27 applies here as well. The circuit PARALLEL-FFT that computes the FFT on n inputs is shown in Figure 30.5 for $n = 8$. It 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 is therefore $\Theta(\lg n)$.

The leftmost part of the circuit PARALLEL-FFT 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. Within stage s, for $s = 1, 2, \ldots, \lg n$, there are $n/2^s$ groups of butterflies (corresponding to each value of k in ITERATIVE-FFT), with 2^{s-1} butterflies per group

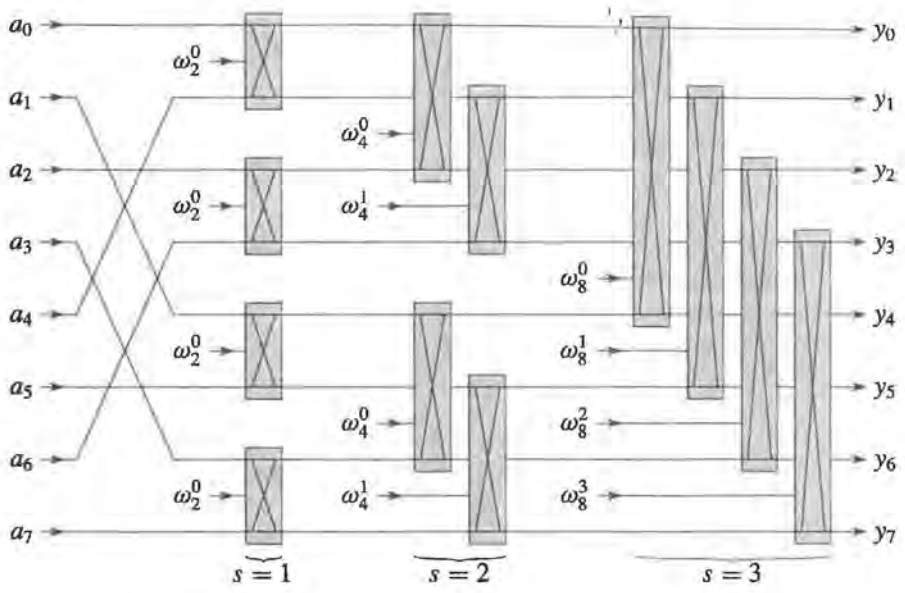

Figure 30.5 A circuit PARALLEL-FFT that computes the FFT, here shown on $n = 8$ inputs. Each butterfly operation takes as input the values on two wires, along with a twiddle factor, and it produces as outputs the values on two wires. The stages of butterflies are labeled to correspond to iterations of 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). An FFT on n inputs can be computed in $\Theta(\lg n)$ depth with $\Theta(n \lg n)$ butterfly operations.

(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 $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, 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

a. Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (*Hint:* One of the multiplications is $(a + b) \cdot (c + d)$.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n that run in time $\Theta(n^{\lg 3})$. 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 that two n-bit integers can be multiplied 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 $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b. Describe how to represent a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added 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. Our input is a d-dimensional array $A = (a_{i_1,i_2,\ldots,i_d})$ whose dimensions are $n_1, n_2, \ldots, n_d$, where $n_1 n_2 \cdots n_d = n$. We define the d-dimensional Discrete Fourier Transform by the equation

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

for $0 \le k_1 < n_1, 0 \le k_2 < n_2, \ldots, 0 \le k_d < n_d$.

a. Show that we can compute a d-dimensional DFT by computing 1-dimensional DFT's on each dimension in turn. That is, first compute n/n_1 separate 1-dimensional DFT's along dimension 1. Then, using the result of the DFT's along dimension 1 as the input, compute n/n_2 separate 1-dimensional DFT's along dimension 2. Using this result as the input, compute n/n_3 separate 1-dimensional DFT's 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 DFT's 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, its tth derivative is defined by

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

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

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

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

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

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

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

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

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

30-5 Polynomial evaluation at multiple points

We have observed that the problem of evaluating a polynomial of degree-bound $n-1$ at a single point can be solved in $O(n)$ time using Horner's rule. We have also discovered that such a polynomial can be evaluated 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 use the fact 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 assume 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 $x_0, x_1, \ldots, x_{n-1}$, we wish to compute the n values $A(x_0), A(x_1), \ldots, A(x_{n-1})$. For $0 \le i \le j \le n-1$, 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 $i \le k \le j$, 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 $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

30-6 FFT using modular arithmetic

As defined, the Discrete Fourier Transform requires the use of 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 it is desirable to utilize a variant of the FFT based on modular arithmetic in order to 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 that 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 $\lg n$. (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 $\mathbf{Z}_p^*$, 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 nth root of unity.

c. Argue that the FFT and its inverse can be made to 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 $\mathbf{Z}_{17}^*$.

Chapter notes

VanLoan's book [303] is an outstanding treatment of the Fast Fourier Transform. Press, Flannery, Teukolsky, and Vetterling [248, 249] 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 Schafer [232] and Oppenheim and Willsky [233]. The Oppenheim and Schafer 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 [127] and Pratt [246] discuss multidimensional Fourier Transforms and their use in image processing, and books by Tolimieri, An, and Lu [300] and Van Loan [303] discuss the mathematics of multidimensional Fast Fourier Transforms.

Cooley and Tukey [68] are widely credited with devising the FFT in the 1960's. 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, Flannery, Teukolsky, and Vetterling attribute the origins of the method to Runge and König in 1924, an article by Heideman, Johnson, and Burrus [141] traces the history of the FFT as far back as C. F. Gauss in 1805.

31 Number-Theoretic Algorithms

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in part to the invention of cryptographic schemes based on large prime numbers. The feasibility of these schemes rests on our ability to find large primes easily, while their security rests on our inability to factor the product of large primes. This chapter presents some of the number theory and associated algorithms that underlie such applications.

Section 31.1 introduces basic concepts of number theory, such as divisibility, modular equivalence, and unique factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers. Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then studies the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax \equiv b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6 considers powers of a given number a, modulo n, and presents a repeated-squaring algorithm for efficiently computing $a^b \bmod n$, given a, b, and n. This operation is at the heart of efficient primality testing and of much modern cryptography. Section 31.7 then describes the RSA public-key cryptosystem. Section 31.8 examines a randomized primality test that can be used to find large primes efficiently, an essential task in creating keys for the RSA cryptosystem. Finally, Section 31.9 reviews a simple but effective heuristic for factoring small integers. It is a curious fact that factoring is one problem people may wish to be intractable, since the security of RSA depends on the difficulty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a "large input" typically means an input containing "large integers" rather than an input containing "many integers" (as for sorting). Thus,