a three-way branching one, with a running time that satisfies

$$T(n) \leq 3T(n/2) + cn$$

for a constant c.

This is the case $q = 3$ of (5.3) that we were aiming for. Using the solution to that recurrence from earlier in the chapter, we have

(5.13) *The running time of* `Recursive-Multiply` *on two n-bit factors is* $O(n^{\log_2 3}) = O(n^{1.59})$.

5.6 Convolutions and the Fast Fourier Transform

As a final topic in this chapter, we show how our basic recurrence from (5.1) is used in the design of the *Fast Fourier Transform*, an algorithm with a wide range of applications.

The Problem

Given two vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, there are a number of common ways of combining them. For example, one can compute the sum, producing the vector $a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$; or one can compute the inner product, producing the real number $a \cdot b = a_0b_0 + a_1b_1 + \cdots + a_{n-1}b_{n-1}$. (For reasons that will emerge shortly, it is useful to write vectors in this section with coordinates that are indexed starting from 0 rather than 1.)

A means of combining vectors that is very important in applications, even if it doesn't always show up in introductory linear algebra courses, is the *convolution* $a * b$. The convolution of two vectors of length n (as a and b are) is a vector with $2n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i,j<n}} a_i b_j.$$

In other words,

$$a * b = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \ldots, a_{n-2}b_{n-1} + a_{n-1}b_{n-2}, a_{n-1}b_{n-1}).$$

This definition is a bit hard to absorb when you first see it. Another way to think about the convolution is to picture an $n \times n$ table whose (i, j) entry is a_ib_j, like this,

$$\begin{array}{ccccc} a_0b_0 & a_0b_1 & \dots & a_0b_{n-2} & a_0b_{n-1} \\ a_1b_0 & a_1b_1 & \dots & a_1b_{n-2} & a_1b_{n-1} \\ a_2b_0 & a_2b_1 & \dots & a_2b_{n-2} & a_2b_{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1}b_0 & a_{n-1}b_1 & \dots & a_{n-1}b_{n-2} & a_{n-1}b_{n-1} \end{array}$$

and then to compute the coordinates in the convolution vector by summing along the diagonals.

It's worth mentioning that, unlike the vector sum and inner product, the convolution can be easily generalized to vectors of different lengths, $a = (a_0, a_1, \dots, a_{m-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$. In this more general case, we define $a * b$ to be a vector with $m + n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i<m, j<n}} a_i b_j.$$

We can picture this using the table of products a_ib_j as before; the table is now rectangular, but we still compute coordinates by summing along the diagonals. (From here on, we'll drop explicit mention of the condition $i < m, j < n$ in the summations for convolutions, since it will be clear from the context that we only compute the sum over terms that are defined.)

It's not just the definition of a convolution that is a bit hard to absorb at first; the motivation for the definition can also initially be a bit elusive. What are the circumstances where you'd want to compute the convolution of two vectors? In fact, the convolution comes up in a surprisingly wide variety of different contexts. To illustrate this, we mention the following examples here.

- A first example (which also proves that the convolution is something that we all saw implicitly in high school) is polynomial multiplication. Any polynomial $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ can be represented just as naturally using its vector of coefficients, $a = (a_0, a_1, \dots, a_{m-1})$. Now, given two polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}$, consider the polynomial $C(x) = A(x)B(x)$ that is equal to their product. In this polynomial $C(x)$, the coefficient on the x^k term is equal to

  $$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  In other words, the coefficient vector c of $C(x)$ is the convolution of the coefficient vectors of $A(x)$ and $B(x)$.
- Arguably the most important application of convolutions in practice is for *signal processing*. This is a topic that could fill an entire course, so

we'll just give a simple example here to suggest one way in which the convolution arises.

Suppose we have a vector $a = (a_0, a_1, \ldots, a_{m-1})$ which represents a sequence of measurements, such as a temperature or a stock price, sampled at m consecutive points in time. Sequences like this are often very noisy due to measurement error or random fluctuations, and so a common operation is to "smooth" the measurements by averaging each value a_i with a weighted sum of its neighbors within k steps to the left and right in the sequence, the weights decaying quickly as one moves away from a_i. For example, in *Gaussian smoothing*, one replaces a_i with

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2},$$

for some "width" parameter k, and with Z chosen simply to normalize the weights in the average to add up to 1. (There are some issues with boundary conditions—what do we do when $i - k < 0$ or $i + k > m$?—but we could deal with these, for example, by discarding the first and last k entries from the smoothed signal, or by scaling them differently to make up for the missing terms.)

To see the connection with the convolution operation, we picture this smoothing operation as follows. We first define a "mask"

$$w = (w_{-k}, w_{-(k-1)}, \ldots, w_{-1}, w_0, w_1, \ldots, w_{k-1}, w_k)$$

consisting of the weights we want to use for averaging each point with its neighbors. (For example, $w = \frac{1}{Z}(e^{-k^2}, e^{-(k-1)^2}, \ldots, e^{-1}, 1, e^{-1}, \ldots, e^{-(k-1)^2}, e^{-k^2})$ in the Gaussian case above.) We then iteratively position this mask so it is centered at each possible point in the sequence a; and for each positioning, we compute the weighted average. In other words, we replace a_i with $a'_i = \sum_{s=-k}^{k} w_s a_{i+s}$.

This last expression is essentially a convolution; we just have to warp the notation a bit so that this becomes clear. Let's define $b = (b_0, b_1, \ldots, b_{2k})$ by setting $b_\ell = w_{k-\ell}$. Then it's not hard to check that with this definition we have the smoothed value

$$a'_i = \sum_{(j,\ell):j+\ell=i+k} a_j b_\ell.$$

In other words, the smoothed sequence is just the convolution of the original signal and the reverse of the mask (with some meaningless coordinates at the beginning and end).

- We mention one final application: the problem of combining histograms. Suppose we're studying a population of people, and we have the following two histograms: One shows the annual income of all the men in the population, and one shows the annual income of all the women. We'd now like to produce a new histogram, showing for each k the number of *pairs* (M, W) for which man M and woman W have a combined income of k.

  This is precisely a convolution. We can write the first histogram as a vector $a = (a_0, \ldots, a_{m-1})$, to indicate that there are a_i men with annual income equal to i. We can similarly write the second histogram as a vector $b = (b_0, \ldots, b_{n-1})$. Now, let c_k denote the number of pairs (m, w) with combined income k; this is the number of ways of choosing a man with income a_i and a woman with income b_j, for any pair (i, j) where $i + j = k$. In other words,

$$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  so the combined histogram $c = (c_0, \ldots, c_{m+n-2})$ is simply the convolution of a and b.

  (Using terminology from probability that we will develop in Chapter 13, one can view this example as showing how convolution is the underlying means for computing the distribution of the sum of two independent random variables.)

Computing the Convolution Having now motivated the notion of convolution, let's discuss the problem of computing it efficiently. For simplicity, we will consider the case of equal length vectors (i.e., $m = n$), although everything we say carries over directly to the case of vectors of unequal lengths.

Computing the convolution is a more subtle question than it may first appear. The definition of convolution, after all, gives us a perfectly valid way to compute it: for each k, we just calculate the sum

$$\sum_{(i,j):i+j=k} a_i b_j$$

and use this as the value of the k^{th} coordinate. The trouble is that this direct way of computing the convolution involves calculating the product $a_i b_j$ for every pair (i, j) (in the process of distributing over the sums in the different terms) and this is $\Theta(n^2)$ arithmetic operations. Spending $O(n^2)$ time on computing the convolution seems natural, as the definition involves $O(n^2)$ multiplications $a_i b_j$. However, it's not inherently clear that we have to spend quadratic time to compute a convolution, since the input and output both only have size $O(n)$.

Could one design an algorithm that bypasses the quadratic-size definition of convolution and computes it in some smarter way?

In fact, quite surprisingly, this is possible. We now describe a method that computes the convolution of two vectors using only $O(n \log n)$ arithmetic operations. The crux of this method is a powerful technique known as the *Fast Fourier Transform* (FFT). The FFT has a wide range of further applications in analyzing sequences of numerical values; computing convolutions quickly, which we focus on here, is just one of these applications.

Designing and Analyzing the Algorithm

To break through the quadratic time barrier for convolutions, we are going to exploit the connection between the convolution and the multiplication of two polynomials, as illustrated in the first example discussed previously. But rather than use convolution as a primitive in polynomial multiplication, we are going to exploit this connection in the opposite direction.

Suppose we are given the vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$. We will view them as the polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}$, and we'll seek to compute their product $C(x) = A(x)B(x)$ in $O(n \log n)$ time. If $c = (c_0, c_1, \ldots, c_{2n-2})$ is the vector of coefficients of C, then we recall from our earlier discussion that c is exactly the convolution $a * b$, and so we can then read off the desired answer directly from the coefficients of $C(x)$.

Now, rather than multiplying A and B symbolically, we can treat them as functions of the variable x and multiply them as follows.

(i) First we choose $2n$ values $x_1, x_2, \ldots, x_{2n}$ and evaluate $A(x_j)$ and $B(x_j)$ for each of $j = 1, 2, \ldots, 2n$.

(ii) We can now compute $C(x_j)$ for each j very easily: $C(x_j)$ is simply the product of the two numbers $A(x_j)$ and $B(x_j)$.

(iii) Finally, we have to recover C from its values on $x_1, x_2, \ldots, x_{2n}$. Here we take advantage of a fundamental fact about polynomials: any polynomial of degree d can be reconstructed from its values on any set of $d + 1$ or more points. This is known as *polynomial interpolation*, and we'll discuss the mechanics of performing interpolation in more detail later. For the moment, we simply observe that since A and B each have degree at most $n - 1$, their product C has degree at most $2n - 2$, and so it can be reconstructed from the values $C(x_1), C(x_2), \ldots, C(x_{2n})$ that we computed in step (ii).

This approach to multiplying polynomials has some promising aspects and some problematic ones. First, the good news: step (ii) requires only

$O(n)$ arithmetic operations, since it simply involves the multiplication of $O(n)$ numbers. But the situation doesn't look as hopeful with steps (i) and (iii). In particular, evaluating the polynomials A and B on a single value takes $\Omega(n)$ operations, and our plan calls for performing $2n$ such evaluations. This seems to bring us back to quadratic time right away.

The key idea that will make this all work is to find a set of $2n$ values $x_1, x_2, \ldots, x_{2n}$ that are intimately related in some way, such that the work in evaluating A and B on all of them can be shared across different evaluations. A set for which this will turn out to work very well is the *complex roots of unity.*

The Complex Roots of Unity At this point, we're going to need to recall a few facts about complex numbers and their role as solutions to polynomial equations.

Recall that complex numbers can be viewed as lying in the "complex plane," with axes representing their real and imaginary parts. We can write a complex number using polar coordinates with respect to this plane as $re^{\theta i}$, where $e^{\pi i} = -1$ (and $e^{2\pi i} = 1$). Now, for a positive integer k, the polynomial equation $x^k = 1$ has k distinct complex roots, and it is easy to identify them. Each of the complex numbers $\omega_{j,k} = e^{2\pi ji/k}$ (for $j = 0, 1, 2, \ldots, k-1$) satisfies the equation, since

$$(e^{2\pi ji/k})^k = e^{2\pi ji} = (e^{2\pi i})^j = 1^j = 1,$$

and each of these numbers is distinct, so these are all the roots. We refer to these numbers as the k^{th} *roots of unity.* We can picture these roots as a set of k equally spaced points lying on the unit circle in the complex plane, as shown in Figure 5.9 for the case $k = 8$.

For our numbers $x_1, \ldots, x_{2n}$ on which to evaluate A and B, we will choose the $(2n)^{\text{th}}$ roots of unity. It's worth mentioning (although it's not necessary for understanding the algorithm) that the use of the complex roots of unity is the basis for the name *Fast Fourier Transform*: the representation of a degree-d

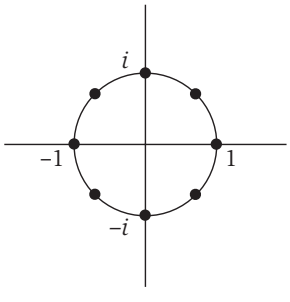

Figure 5.9 The 8^{th} roots of unity in the complex plane.

polynomial P by its values on the $(d+1)^{\text{st}}$ roots of unity is sometimes referred to as the *discrete Fourier transform* of P; and the heart of our procedure is a method for making this computation fast.

A Recursive Procedure for Polynomial Evaluation We want to design an algorithm for evaluating A on each of the $(2n)^{\text{th}}$ roots of unity recursively, so as to take advantage of the familiar recurrence from (5.1)—namely, $T(n) \leq 2T(n/2) + O(n)$ where $T(n)$ in this case denotes the number of operations required to evaluate a polynomial of degree $n-1$ on all the $(2n)^{\text{th}}$ roots of unity. For simplicity in describing this algorithm, we will assume that n is a power of 2.

How does one break the evaluation of a polynomial into two equal-sized subproblems? A useful trick is to define two polynomials, $A_{even}(x)$ and $A_{odd}(x)$, that consist of the even and odd coefficients of A, respectively. That is,

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2},$$

and

$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{(n-1)} x^{(n-2)/2}.$$

Simple algebra shows us that

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2),$$

and so this gives us a way to compute $A(x)$ in a constant number of operations, given the evaluation of the two constituent polynomials that each have half the degree of A.

Now suppose that we evaluate each of A_{even} and A_{odd} on the n^{th} roots of unity. This is exactly a version of the problem we face with A and the $(2n)^{\text{th}}$ roots of unity, except that the input is half as large: the degree is $(n-2)/2$ rather than $n-1$, and we have n roots of unity rather than $2n$. Thus we can perform these evaluations in time $T(n/2)$ for each of A_{even} and A_{odd}, for a total time of $2T(n/2)$.

We're now very close to having a recursive algorithm that obeys (5.1) and gives us the running time we want; we just have to produce the evaluations of A on the $(2n)^{\text{th}}$ roots of unity using $O(n)$ additional operations. But this is easy, given the results from the recursive calls on A_{even} and A_{odd}. Consider one of these roots of unity $\omega_{j,2n} = e^{2\pi ji/2n}$. The quantity $\omega_{j,2n}^2$ is equal to $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$, and hence $\omega_{j,2n}^2$ is an n^{th} root of unity. So when we go to compute

$$A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n} A_{odd}(\omega_{j,2n}^2),$$

we discover that both of the evaluations on the right-hand side have been performed in the recursive step, and so we can determine $A(\omega_{j,2n})$ using a

constant number of operations. Doing this for all $2n$ roots of unity is therefore $O(n)$ additional operations after the two recursive calls, and so the bound $T(n)$ on the number of operations indeed satisfies $T(n) \leq 2T(n/2) + O(n)$. We run the same procedure to evaluate the polynomial B on the $(2n)^{\text{th}}$ roots of unity as well, and this gives us the desired $O(n \log n)$ bound for step (i) of our algorithm outline.

Polynomial Interpolation We've now seen how to evaluate A and B on the set of all $(2n)^{\text{th}}$ roots of unity using $O(n \log n)$ operations and, as noted above, we can clearly compute the products $C(\omega_{j,n}) = A(\omega_{j,2n})B(\omega_{j,2n})$ in $O(n)$ more operations. Thus, to conclude the algorithm for multiplying A and B, we need to execute step (iii) in our earlier outline using $O(n \log n)$ operations, reconstructing C from its values on the $(2n)^{\text{th}}$ roots of unity.

In describing this part of the algorithm, it's worth keeping track of the following top-level point: it turns out that the reconstruction of C can be achieved simply by defining an appropriate polynomial (the polynomial D below) and evaluating it at the $(2n)^{\text{th}}$ roots of unity. This is exactly what we've just seen how to do using $O(n \log n)$ operations, so we do it again here, spending an additional $O(n \log n)$ operations and concluding the algorithms.

Consider a polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ that we want to reconstruct from its values $C(\omega_{s,2n})$ at the $(2n)^{\text{th}}$ roots of unity. Define a new polynomial $D(x) = \sum_{s=0}^{2n-1} d_s x^s$, where $d_s = C(\omega_{s,2n})$. We now consider the values of $D(x)$ at the $(2n)^{\text{th}}$ roots of unity.

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n})\omega_{j,2n}^s \\ &= \sum_{s=0}^{2n-1} \Big(\sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t\Big)\omega_{j,2n}^s \\ &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} \omega_{s,2n}^t \omega_{j,2n}^s\Big), \end{aligned}$$

by definition. Now recall that $\omega_{s,2n} = (e^{2\pi i/2n})^s$. Using this fact and extending the notation to $\omega_{s,2n} = (e^{2\pi i/2n})^s$ even when $s \geq 2n$, we get that

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} e^{(2\pi i)(st+js)/2n}\Big) \\ &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s\Big). \end{aligned}$$

To analyze the last line, we use the fact that for any $(2n)^{\text{th}}$ root of unity $\omega \neq 1$, we have $\sum_{s=0}^{2n-1} \omega^s = 0$. This is simply because ω is by definition a root of $x^{2n} - 1 = 0$; since $x^{2n} - 1 = (x-1)(\sum_{t=0}^{2n-1} x^t)$ and $\omega \neq 1$, it follows that ω is also a root of $(\sum_{t=0}^{2n-1} x^t)$.

Thus the only term of the last line's outer sum that is not equal to 0 is for c_t such that $\omega_{t+j,2n} = 1$; and this happens if $t+j$ is a multiple of $2n$, that is, if $t = 2n - j$. For this value, $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. So we get that $D(\omega_{j,2n}) = 2nc_{2n-j}$. Evaluating the polynomial $D(x)$ at the $(2n)^{\text{th}}$ roots of unity thus gives us the coeffients of the polynomial $C(x)$ in reverse order (multiplied by $2n$ each). We sum this up as follows.

(5.14) *For any polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$, and corresponding polynomial $D(x) = \sum_{s=0}^{2n-1} C(\omega_{s,2n}) x^s$, we have that $c_s = \frac{1}{2n} D(\omega_{2n-s,2n})$.*

We can do all the evaluations of the values $D(\omega_{2n-s,2n})$ in $O(n \log n)$ operations using the divide-and-conquer approach developed for step (i).

And this wraps everything up: we reconstruct the polynomial C from its values on the $(2n)^{\text{th}}$ roots of unity, and then the coefficients of C are the coordinates in the convolution vector $c = a * b$ that we were originally seeking.

In summary, we have shown the following.

(5.15) *Using the Fast Fourier Transform to determine the product polynomial $C(x)$, we can compute the convolution of the original vectors a and b in $O(n \log n)$ time.*

Solved Exercises

Solved Exercise 1

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is *unimodal*: For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.)

You'd like to find the "peak entry" p without having to read the entire array—in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A.