7

Divide and Conquer

The *divide-and-conquer* paradigm is one of the most powerful design strategies available in the theory of algorithms. The paradigm can be described in general terms as follows. A problem input (instance) is divided according to some criteria into a set of smaller inputs to the same problem. The problem is then solved for each of these smaller inputs, either recursively by further division into smaller inputs or by invoking an ad hoc or a priori solution. Finally, the solution for the original input is obtained by expressing it in some form as a combination of the solution for these smaller inputs. Ad hoc solutions are often invoked when the input size is smaller than some preassigned *threshold* value. Examples of a priori solutions (solutions known in advance) include sorting single element lists or multiplying single-digit binary numbers.

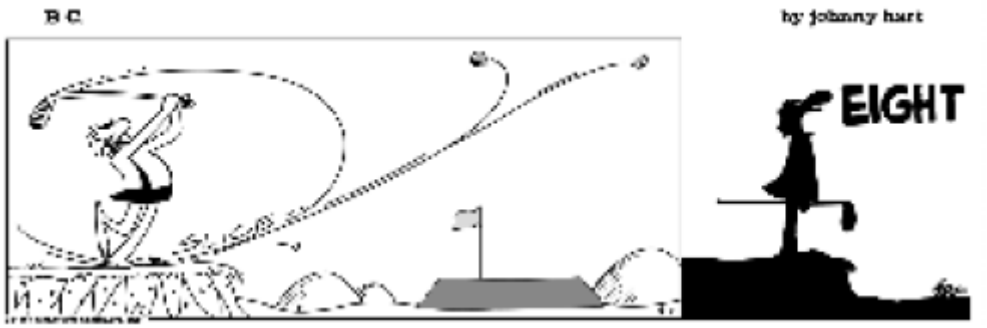
7.1 The Divide-and-Conquer Paradigm

The divide-and-conquer design strategy can be formalized as follows. Let *Known* denote the set of inputs to the problem whose solutions are known a priori or by ad hoc methods. The procedure *Divide_and_Conquer* calls two procedures *Divide* and *Combine*. *Divide* has input parameter I and output parameters $I_1,...,I_m$, where m may depend on I. The inputs $I_1,...,I_m$ must be smaller or simpler inputs to the problem than I, but are not required to always be a *division* of I into subinputs. *Combine* has input parameters $J_1,...,J_m$ and output parameter J. *Combine* is the procedure for obtaining a solution J to the problem with input I by combining the recursively obtained solutions J_1, ... J_m to the problem with inputs $I_1,...,I_m$. For convenience, we formulate *Divide_and_Conquer* as a procedure.

```
procedure Divide_and_Conquer(I,J) recursive
Input: I (an input to the given problem)
Output:   J (a solution to the given problem corresponding to input I)
    if I ∈ Known then
        assign the a priori or ad hoc solution for I to J
    else
        Divide(I,I1,...,Im)        //m may depend on the input I
        for i ← 1 to m do
            Divide_and_Conquer(Ii,Ji)
        endfor
        Combine(J1,...,Jm,J)
    endif
end Divide_and_Conquer
```

Often the work done by an algorithm based on *Divide_and_Conquer* resides solely in one of the two procedures *Divide* or *Combine*, but not both. For example, in the algorithm *QuickSort*, the divide step (call to *Partition*) is the heart of the algorithm and

the combine step requires no work at all. On the other hand, in the algorithm *MergeSort*, the divide step is trivial and the combine step (call to *Merge*) is the heart of the algorithm. Sometimes, both the divide and combine steps are difficult (see Figure 7.1).



An example of a hard divide – hard combine divide-and-conquer algorithm
(Reprinted by permission of Johnny Hart and Creators Syndicate, Inc.)

Figure 7.1

For most divide-and-conquer algorithms, the number m of subproblems is a constant (independent of any particular input I). Divide-and-conquer algorithms where m is a constant equal to one are referred to as *simplifications*. The binary search algorithm is an example of a simplification.

One useful technique to improve the efficiency of a divide-and-conquer algorithm is to employ a known ad hoc algorithm when the input size is smaller than some *threshold*. Of course, the ad hoc algorithm must be more efficient than the given divide-and-conquer algorithm for sufficiently small inputs.

For example, consider the sorting algorithm *MergeSort*, which has $\Theta(n \log n)$ average complexity. The sorting algorithm *InsertionSort*, which has $\Theta(n^2)$ average complexity, is much less efficient than *MergeSort* for large values of n. However, due to the constants involved, *InsertionSort* is more efficient than *MergeSort* for small values of n. Thus, we can improve the performance of *MergeSort* by calling *InsertionSort* for input lists of size not larger than a suitable threshold. Finding the optimal value for a threshold is often done empirically in practice. Empirical studies are needed since the best choice of a threshold depends on the constants associated with the implementation on a particular computer. For example, empirical studies have shown that for *MergeSort*, calling *InsertionSort* with a threshold of around $n = 16$ is usually optimal.

7.2 Symbolic Algebraic Operations on Polynomials

Algebraic manipulation of polynomials is an essential tool in many applications, and therefore the need arises to design efficient algorithms to carry out the basic arithmetic operations on polynomials, such as addition and multiplication, as efficiently as possible. Given a polynomial $P(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0$, it is important to differentiate between the pointwise and the symbolic representation of $P(x)$. The *pointwise representation* of $P(x)$ is simply a function that maps an input point x to the output point

$P(x)$. Thus, given two polynomials $P(x)$ and $Q(x)$, their *pointwise product* is the function *PtWiseMulT*(P,Q) that maps an input point x to the output point $P(x) \times Q(x)$.

The *symbolic representation* of a polynomial $P(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0$ is its coefficient array $[a_0,a_1,...,a_{m-1}]$. Thus given two polynomials $P(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0$ and $Q(x) = b_{n-1}x^{n-1} + ... + b_1x + b_0$, their symbolic product is the coefficient array $[c_0,c_1,...,c_{m+n-2}]$ of the product polynomial $P(x)Q(x)$ given by

$$c_k = \sum_{i+j=k} a_i b_j \ , \ 0 \le i \le m-1, 0 \le j \le n-1, k = 0,\ldots,m+n-2. \qquad (7.2.1)$$

For example, the symbolic product of $P(x) = 3x^2 + 2x - 5$ having coefficient array [–5, 2, 3] and $Q(x) = x^3 - x + 4$ having coefficient array [4, –1, 0, 1] is the polynomial $3x^5 + 2x^4 - 8x^3 + 10x^2 + 13x - 20$ having coefficient array [–20, 13, 10, –8, 2, 3].

The symbolic representation of the product is particularly important for determining various properties of the product polynomial. For example, the function *PtWiseMult* does not lend itself to taking the derivative of the product polynomial, whereas it is a simple matter to compute the symbolic representation of the derivative of a polynomial that is represented symbolically. Similar comments hold for other operations on polynomials such as root finding, integration, and so forth. Throughout the remainder of this chapter, when discussing algebraic operations on polynomials we implicitly assume that we are performing these operations symbolically.

7.2.1 Multiplication of Polynomials of the Same Input Size

An algorithm *DirectPolyMult* based on a straightforward calculation of (7.2.1) has complexity $\Theta(mn)$ (where we choose multiplication of coefficients as our basic operation). We now describe a more efficient algorithm for polynomial multiplication based on the divide-and-conquer paradigm. We first assume that $m = n$. Setting $d = \lceil n/2 \rceil$, we divide the set of coefficients of the polynomials in half, with the higher-order coefficients $a_{n-1},a_{n-2},...,a_d$ in one set and the lower-order coefficients of $a_{d-1},a_{d-2},...,a_0$ in the other. Setting $P_1(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + ... + a_1x + a_0$ and $P_2(x) = a_{n-1}x^{n-d-1} + a_{n-2}x^{n-d-2} + ... + a_{d+1}x + a_d$ we obtain

$$P(x) = x^d P_2(x) + P_1(x).$$

A similar division of the set of coefficients of $Q(x)$ yields polynomials $Q_1(x)$ and $Q_2(x)$ having input size of at most d such that

$$Q(x) = x^d Q_2(x) + Q_1(x).$$

A straightforward application of the distributive law yields

$$P(x)Q(x) = x^{2d}P_2(x)Q_2(x) + x^d(P_1(x)Q_2(x) + P_2(x)Q_1(x)) + P_1(x)Q_1(x). \qquad (7.2.2)$$

Note that polynomials $P_1(x)$ and $Q_1(x)$ both have input size d, and polynomials $P_2(x)$ and $Q_2(x)$ both have input size either d or $d - 1$. In the case when $P_2(x)$ and $Q_2(x)$ both have input size $d - 1$, we add a leading coefficient of zero to each so that they have input

size d. Thus, the problem of multiplying $P(x)$ and $Q(x)$ has been reduced to the problem of taking four products of polynomials of input sizes $d = \lceil n/2 \rceil$ together with two multiplications by powers of x and three additions. It turns out that the resulting divide-and-conquer algorithm based on (7.2.2) still has quadratic complexity. However, there is a clever way of combining the split polynomials that uses only *three* polynomial multiplications instead of four, based on the following simple identity

$$P(x)Q(x) = x^{2d}P_2(x)Q_2(x) + x^d((P_1(x)+P_2(x))(Q_1(x)+Q_2(x)) - P_1(x)Q_1(x) - P_2(x)Q_2(x)) + P_1(x)Q_1(x). \quad (7.2.3)$$

Formula (7.2.3) yields the following divide-and-conquer algorithm *PolyMult1* for polynomial multiplication. *PolyMult1* calls a (split pea) procedure *SpliT*($P(x)$,$P_1(x)$,$P_2(x)$), which inputs a polynomial $P(x)$ (of input size n) and outputs the two polynomials $P_1(x)$ and $P_2(x)$. We write *PolyMult1* as a high-level recursive function whose inputs are the polynomials $P(x)$ and $Q(x)$ and whose output is the polynomial $P(x)Q(x)$. We will not be explicit about how the coefficients of the polynomials are maintained (linked lists, arrays, and so forth). We will also assume that we have well-defined procedures for multiplying a polynomial by x^i. For example, if the polynomial is maintained by an array of its coefficients, then this amounts to shifting indices by i and replacing the first i entries with zeros. We abuse the notation slightly by writing $x^iP(x)$ to mean the result of calling such a procedure. Also, for convenience we simply use the symbol + to denote the addition of two polynomials.

```
function PolyMult1(P,Q,n) recursive
Input: P(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0, Q(x) = b_{n-1}x^{n-1} + ... + b_1x + b_0
                                                                    (polynomials)
          n (a positive integer)
Output:   P(x)Q(x) (the product polynomial)
    if n = 1 then
        return(a_0b_0)
    else
        d ← ⌈n/2⌉
        SpliT(P(x),P_1(x),P_2(x))
        SpliT(Q(x),Q_1(x),Q_2(x))
        R(x) ← PolyMult1(P_2(x),Q_2(x),d)
        S(x) ← PolyMult1(P_1(x) + P_2(x),Q_1(x) + Q_2(x),d)
        T(x) ← PolyMult1(P_1(x),Q_1(x),d)
        return(x^{2d}R(x) + x^d(S(x) – R(x) – T(x)) + T(x))
    endif
end PolyMult1
```

When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation, and thus ignore the two multiplications by powers of x. However, the procedure referred to earlier for multiplying a polynomial by x^i has linear complexity and does not affect the order of complexity of *PolyMult1*. We also assume that n is a power of two, since we can interpolate asymptotic behavior using Θ-scalability (see Section 7.4.3).

Since *PolyMult1* invokes itself three times with n replaced by $d = n/2$, the number of coefficient multiplications $T(n)$ performed by *PolyMult1* satisfies the following recurrence relation

$$T(n) = 3T\left(\frac{n}{2}\right), \quad n > 1, \quad \textbf{init.cond.}\ T(1) = 1. \tag{7.2.4}$$

It follows from a simple unwinding of (7.2.4) that $T(n) \in \Theta(n^{\log_2 3})$ (see also the discussion following (3.3.12) in Chapter 3). Since $\log_2 3$ is approximately 1.59, we now have an algorithm for polynomial multiplication whose $\Theta(n^{\log_2 3})$ complexity is a significant improvement over the $\Theta(n^2)$ complexity of *DirectPolyMult*. In Chapter 21 we develop an even faster polynomial multiplication algorithm utilizing the powerful tool known as the Fast Fourier Transform.

7.2.2 Multiplication of Polynomials of Different Input Sizes

In practice, we often encounter the problem of multiplying two polynomials $P(x)$ and $Q(x)$ of different input sizes m and n, respectively. If $m < n$, then we could merely augment $P(x)$ with $n - m$ leading zeros, but this would be quite inefficient if n is significantly larger than m. It is better to partition $Q(x)$ into blocks of size m. For convenience, we assume n is a multiple of m, that is, $n = km$ for some positive integer k. We let $Q_i(x)$ be the polynomial of degree m given by

$$Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}, \quad i \in \{1, \ldots, k\}.$$

Clearly,

$$Q(x) = Q_k(x)x^{m(k-1)} + Q_{k-1}(x)x^{m(k-2)} + \cdots + Q_2(x)x^m + Q_1(x).$$

It follows immediately from the distributive law that

$$P(x)Q(x) = P(x)Q_k(x)x^{m(k-1)} + P(x)Q_{k-1}(x)x^{m(k-2)} + \cdots + P(x)Q_1(x).$$

Applying the above ideas, we obtain the following algorithm *PolyMult2* for multiplying two polynomials $P(x)$ and $Q(x)$. *PolyMult2* is efficient even if the degree $m - 1$ of $P(x)$ is much less than the degree $n - 1$ of $Q(x)$. If n is not a multiple of m, we compute the largest integer k such that $n > km$ and augment $P(x)$ with $n - km$ leading zeros.

function *PolyMult2*($P(x)$,$Q(x)$,m,n)
Input: $P(x) = a_{m-1}x^{m-1} + \ldots + a_1x + a_0$, $Q(x) = b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ (polynomials)
n, m (positive integers) //$n = km$ for some integer k
Output: $P(x)Q(x)$ (the product polynomial)
    ProdPoly(x) ← 0 {initialize all coefficients of *ProdPoly*(x) to be 0}
    for $i \leftarrow 1$ **to** k **do**
        $Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \ldots + B_{(i-1)m}$
    endfor
    for $i \leftarrow 1$ **to** k **do**
        ProdPoly(x) ← *ProdPoly*(x) + $x^{(i-1)m}$*PolyMult1*($P(x)$,$Q_i(x)$,m)

```
    endfor
    return(ProdPoly(x))
end PolyMult2
```

The complexity of *PolyMult1* for multiplying two polynomials of degree $m - 1$ is $\Theta(m^{\log_2 3})$. Since *PolyMult2* invokes *PolyMult1* a total of $k = n/m$ times, each time with input polynomials of degree $m - 1$, it follows that the complexity of *PolyMult2* is

$$\Theta(km^{\log_2 3}) = \Theta\left(\frac{nm^{\log_2 3}}{m}\right) = \Theta\left(nm^{\log_2(3/2)}\right).$$

7.3 Multiplication of Large Integers

Computers typically assign a fixed number of bits for storing integer variables. Arithmetic operations such as addition and multiplication of integers are often carried out by moving the integer operands into *fixed-length* registers and then invoking arithmetic operations built into the hardware. However, there are important applications, such as those that occur in cryptography, when the number of digits is too large to be handled in this way directly by the hardware. In such cases, it is necessary to perform these operations by storing the integers using an appropriate data structure (such as an array or a linked list) and then writing algorithms for carrying out the arithmetic operations. When analyzing the complexity of such algorithms, the size n of an integer A is taken to be the number of digits of A. Any base $b \geq 2$ can be chosen for representing an integer. We denote the ith least significant digit of U (base b) by u_i, $i = 0, 1,...,n - 1$; that is,

$$U = \sum_{i=0}^{n-1} u_i b^i. \tag{7.3.1}$$

The classic grade-school algorithm for multiplying two integers U and V of size n clearly involves n^2 multiplications of digits. Viewing the right-hand side of (7.3.1) as a polynomial in b leads us to a divide-and-conquer strategy for computing UV based on a formula analogous to (7.2.3). Note that although addition and multiplication of large integers are similar to multiplication of polynomials, there is the additional task of handling carry digits. The design details of the resulting $\Theta(n^{\log_2 3})$ algorithm *MultInt* are left to the exercises.

7.4 Multiplication of Matrices

Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, $0 \leq i, j \leq n - 1$, recall that the product AB is defined to be the $n \times n$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}. \tag{7.4.1}$$

The straightforward algorithm based on this definition clearly performs n^3 (scalar) multiplications. Around 1970 Strassen devised a divide-and-conquer algorithm for matrix

multiplication of complexity $O(n^{\log_2 7})$ using certain algebraic identities for multiplying 2×2 matrices.

The classic method of multiplying 2 × 2 matrices performs 8 multiplications as follows:

$$AB = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}. \tag{7.4.2}$$

Strassen discovered a way to carry out the same matrix product *AB* using only the following seven multiplications:

$$\begin{aligned} m_1 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11})b_{00} \\ m_3 &= a_{00}(b_{01} - b_{11}) \\ m_4 &= a_{11}(b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01})(b_{11}) \\ m_6 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned} \tag{7.4.3}$$

The matrix product *AB* is then given by

$$AB = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}. \tag{7.4.4}$$

Consider now the case of two $n \times n$ matrices, where for convenience we assume that $n = 2^k$. We first partition the matrices *A* and *B* into four $(n/2) \times (n/2)$ submatrices, as shown.

$$A = \left[\begin{array}{c:c} A_{00} & A_{01} \\ \hdashline A_{10} & A_{11} \end{array}\right], \quad B = \left[\begin{array}{c:c} B_{00} & B_{01} \\ \hdashline B_{10} & B_{11} \end{array}\right] \tag{7.4.5}$$

The product *AB* can be expressed in terms of eight matrix products as follows.

$$AB = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}. \tag{7.4.6}$$

Thus, in complete analogy with the 2 × 2 case, we can carry out the matrix product *AB* using only the following seven matrix multiplications.

$$\begin{aligned} M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ M_2 &= (A_{10} + A_{11})B_{00} \\ M_3 &= A_{00}(B_{01} - B_{11}) \\ M_4 &= A_{11}(B_{10} - B_{00}) \\ M_5 &= (A_{00} + A_{01})(B_{11}) \\ M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned} \tag{7.4.7}$$

As in the case of 2×2 matrices, the matrix product AB is then given by

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}. \tag{7.4.8}$$

Equations (7.4.7) and (7.4.8) immediately yield a divide-and-conquer algorithm (Strassen) based on expressing the product of two $n \times n$ matrices in terms of seven products of $(n/2) \times (n/2)$ matrices. The complexity of Strassen's algorithm clearly satisfies the recurrence relation

$$T(n) = 7T\left(\frac{n}{2}\right), \quad n > 1, \quad \textbf{init.cond.}\ T(1) = 1. \tag{7.4.9}$$

By unwinding (7.4.9) we see that see also the discussion following (3.3.12 in Chapter 3) $T(n) \in \Theta(n^{\log_2 7})$. Since $\log_2 7$ is approximately 2.81, we now have an algorithm for matrix multiplication whose $\Theta(n^{\log_2 7})$ complexity is a significant improvement over the $\Theta(n^3)$ complexity of the classical algorithm for matrix multiplication.

Note that Strassen's identities (7.4.3) and (7.4.4) involve a total of 18 additions (or subtractions). Winograd discovered the following set of identities, which leads to a method of multiplying 2×2 matrices using only 15 additions or subtractions, but still doing only seven multiplications.

$$\begin{aligned} m_1 &= (a_{10} + a_{11} - a_{00})(b_{11} - b_{01} + b_{00}) \\ m_2 &= a_{00}b_{00} \\ m_3 &= a_{01}b_{10} \\ m_4 &= (a_{00} - a_{10})(b_{11} - b_{01}) \\ m_5 &= (a_{10} + a_{11})(b_{01} - b_{00}) \\ m_6 &= (a_{01} - a_{10} + a_{00} - a_{11})b_{11} \\ m_7 &= a_{11}(b_{00} - b_{11} - b_{01} + b_{10}) \end{aligned} \tag{7.4.10}$$

The matrix product AB is then given by

$$AB = \begin{bmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{bmatrix}. \tag{7.4.11}$$

Although (7.4.10) and (7.4.11) involve a total of 24 additions or subtractions, it can be verified that a total of only 15 distinct additions or subtractions are needed. Thus, we obtain an improvement of three less additions or subtractions over Strassen's identities, but still use only seven multiplications.

7.5 The Discrete Fourier Transform

One of the most celebrated divide-and-conquer algorithms is the Fast Fourier Transform (*FFT*) implementing the Discrete Fourier Transform. The Fast Fourier Transform algorithm has a wide range of important applications to such fields as signal processing and image processing, computerized axial tomography (CAT) scans, weather prediction

and statistics, and coding theory. In fact, the *FFT* has been referred to as the "most important numerical algorithm in our lifetime." In this chapter we describe the *FFT* and show how it can be used to obtain an $O(n\log n)$ algorithm for polynomial multiplication. We begin by introducing the Discrete Fourier Transform, and then show how it can be computed by the $O(n\log n)$ divide-and-conquer algorithm *FFT*. Then we show how *FFT* can be used to multiply two polynomials efficiently by transforming the problem to one of simply multiplying n numbers. We finish the chapter by a discussion of how the *FFT* can be implemented on the PRAM and on the Butterfly Interconnection Network.

7.5.1 The Discrete Fourier Transform

The Discrete Fourier Transform DFT_ω of a polynomial of degree n is defined relative to a *primitive nth root of unity* ω. A primitive n^{th} root of unity (over the field of complex numbers) is a complex number ω such that $\omega^n = 1$ and $\omega^k \neq 1$ for $0 < k < n$ (see Appendix A).

Definition 7.5.1

Given a primitive n^{th} root of unity ω, the Discrete Fourier Transform DFT_ω transforms a polynomial $P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ with *real* or *complex* coefficients into the polynomial $b_{n-1}x^{n-1} + \ldots + b_1x^1 + b_0$, where $b_i = P(\omega^i)$, $i = 0,1, \ldots, n-1$. That is,

$$DFT_\omega(P(x)) = P(\omega^{n-1})x^{n-1} + \cdots + P(\omega)x + P(1). \qquad (7.5.1)$$

|

For convenience, we (equivalently) regard DFT_ω as transforming the coefficient array $\mathbf{a} = [a_0, a_1, \ldots a_{n-1}]$ of the polynomial $P(x)$ into the coefficient array $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$ of the polynomial $DFT_\omega(P(x))$. Then **a** and **b** are related by the following matrix equation.

$$\begin{bmatrix} b_0 \\ b_1 \\ \\ \\ \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ 1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \\ \\ \\ a_{n-1} \end{bmatrix} \qquad (7.5.2)$$

We denote the $n \times n$ matrix in (7.5.2) by $F_\omega = (f_{ij})_{n \times n}$, so that

$$f_{ij} = \omega^{(i-1)(j-1)}, \quad i, j = 1, \ldots, n. \qquad (7.5.3)$$

7.5.2 The Fast Fourier Transform

A straightforward way to compute $DFT_\omega(P(x))$ is simply to evaluate $P(x)$ at the points $1, \omega, \ldots, \omega^{n-1}$ using Horner's rule, yielding an $O(n^2)$ algorithm. In fact, for evaluating $P(x)$ at any old set of n points, repeated applications of Horner's rule is the best that we can do. However, the n points, $1, \omega, \ldots, \omega^{n-1}$ are very special beyond the mere fact that they are powers of a given number ω. The special nature of these points can be utilized to design an $O(n \log n)$ algorithm called the *Fast Fourier Transform* (*FFT*) for computing

$DFT_\omega(P(x))$. Indeed, we now motivate the choice of points 1, ω, . . . , ω^{n-1} by showing how a divide-and-conquer algorithm to evaluate $P(x)$ at n points might proceed if the n points have certain simple relationships with one another.

Suppose we choose the n points so that half of the points are the negatives of the other half, say,

$$z_0, z_1, \ldots, z_{n/2-1}, -z_0, -z_1, \ldots, -z_{n/2-1}. \tag{7.5.4}$$

Evaluating a polynomial $P(x) = a_{n-1}x^{n-1} + \ldots + a_1x = a_0$ at such a set of points requires basically half the number of multiplications required for an arbitrary set of n points. To see this, we merely have to write $P(x)$ as the sum of two polynomials of degree $n/2 - 1$ by gathering up the even and odd powers of x.

$$P(x) = Even(x^2) + xOdd(x^2) \tag{7.5.5}$$

where $Even(x) = a_{n-2}x^{n/2-1} + \ldots + a_2x + a_0$ and $Odd(x) = a_{n-1}x^{n/2-1} + \ldots + a_3x + a_1$. Now if we replace x by $-x$ in (7.5.5), we obtain

$$P(-x) = Even(x^2) - xOdd(x^2). \tag{7.5.6}$$

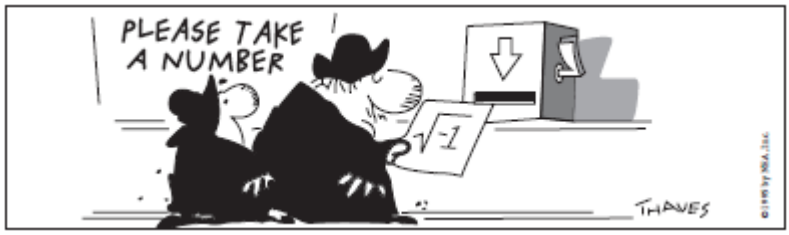
Thus, to evaluate $P(x)$ at the n points given by (7.5.4), we need only evaluate the two polynomials *Even* and *Odd* at the $n/2$ points $(z_0)^2, (z_1)^2, \ldots, (z_{n/2-1})^2$, and then perform the $n/2$ additions, $n/2$ subtractions, and n multiplications as described in (7.5.5) and (7.5.6). Since we have divided our evaluation problem into two problems of size $n/2$, together with a combine step of $O(n)$ complexity, this looks like a promising start to a divide-and-conquer strategy that satisfies the familiar recurrence relation $t(n) = 2t(n/2) + O(n)$, and therefore yields an algorithm having $O(n\log n)$ complexity.

The same strategy can be applied to evaluate *Even* and *Odd* at $(z_0)^2, (z_1)^2, \ldots, (z_{n/2-1})^2$ if we choose $z_0, z_1, \ldots, z_{n/2-1}$ to satisfy the relation

$$(z_{n/4+j})^2 = -(z_j)^2,\ j = 0, \ldots, n/4 - 1. \tag{7.5.7}$$

Now (7.5.7) presents a *real* problem (not an *imaginary* problem, see Figure 7.2). More precisely while we cannot find *real* (nonzero) numbers satisfying (7.5.7), there is no problem finding *complex* numbers that do the job. We merely have to take

$$z_{n/4+j} = iz_j,\ j = 0, \ldots, \frac{n}{4} - 1,\ i = \sqrt{-1}. \tag{7.5.8}$$



Taking a number that leads to numbers satisfying Formula (7.5.7)

Figure 7.2

If we carry the divide-and-conquer strategy to the next step, we are looking at the problem of the evaluation of degree $n/8 - 1$ polynomials at the $n/8$ points $(z_0)^8$, $(z_1)^8$, . . . , $(z_{n/8-1})^8$. As before, the trick allowing us to use formulas (7.5.2) and (7.5.3) works if we choose $z_{n/8+j}$, $j = 0, \ldots, n/8 - 1$, so that

$$(z_{n/8+j})^4 = -(z_j)^4, \ j = 0, \ldots, \frac{n}{8} - 1. \tag{7.5.9}$$

Formula (7.5.9) is, in turn, satisfied if we set

$$z_{n/8+j} = \sqrt{i}(z_j), \ j = 0, \ldots, \frac{n}{8} - 1. \tag{7.5.10}$$

To see precisely where we are headed in the case of a general $n = 2^k$, we consider the case $n = 7$. The three steps just carried out for $n = 8$ determine the specially chosen eight points for rapid evaluation of polynomials of degree seven or less. Figure 7.3 shows a table containing the eight points that have been chosen subject to (7.5.8), (7.5.9), and (7.5.10). For simplicity, we have added the additional condition that $z_0 = 1$.

z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
1	$\sqrt{i}$	i	$i\sqrt{i}$	-1	$-\sqrt{i}$	$-i$	$-i\sqrt{i}$

Table of special points for evaluating polynomials of degree ≤ 7

Figure 7.3

Note that $\sqrt{i}\ (= e^{2\pi i/8} = (1+i)/\sqrt{2})$ is a complex number ω such that $\omega^8 = 1$, whereas $\omega^j \neq 1$ for $0 < j < 7$. Hence, the points $z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7$ in Figure 7.3 are all distinct and are of the form $1, \omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6, \omega^7$, where ω is a primitive eighth root of unity.

Returning to the case of a general $n = 2^k$, if we continue choosing points by generalizing the constraints expressed by (7.5.7) and (7.5.9) in the obvious way, after $\log_2 n$ steps we will arrive at the problem of evaluating constant polynomials at $(z_0)^n$, $\omega(z_0)^n$, where ω is a primitive n^{th} root of unity. Again, for simplicity, we take z_0 to be 1 and ω to be the complex number $e^{2\pi i/n}$. We can verify that the points that we have then generated by this divide-and-conquer strategy are $1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}$. Following our previous discussion, an $O(n\log n)$ recursive algorithm *FFTRec* for evaluating $P(1), P(\omega), \ldots, P(\omega^{n-1})$ is then easily obtained.

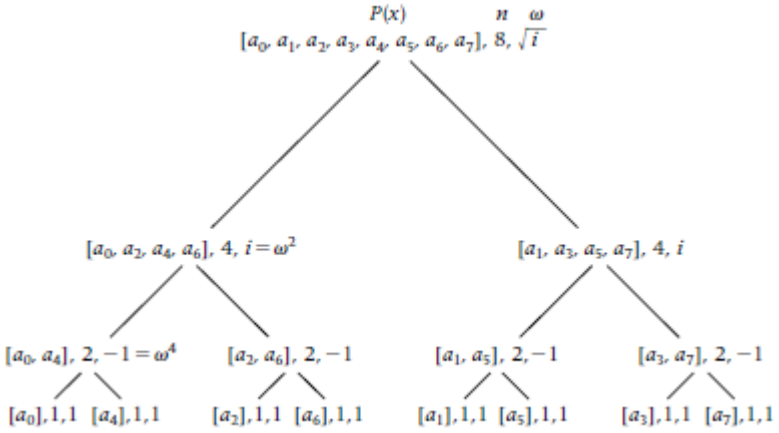
When writing the pseudocode for the algorithm *FFTRec*, it is convenient to use the coefficient array $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ to represent the polynomial $P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. *FFTRec* evaluates $P(1), P(\omega), \ldots, P(\omega^{n-1})$ by first splitting $P(x)$ into even- and odd-degree terms $Even(x) = a_{n-2}x^{n/2-1} + \ldots + a_2x + a_0$ and $Odd(x) = a_{n-1}x^{n/2-1} + \ldots + a_3x + a_1$. Then *FFTRec* recursively evaluates $Even(1), Even(\omega^2), \ldots,$

Even($(\omega^2)^{(n/2)-1}$) and *Odd*(1), *Odd*(ω^2), . . . , *Odd*($(\omega^2)^{(n/2)-1}$). Finally, *FFTRec* uses (7.5.5) and (7.5.6) to evaluate $P(1), P(\omega), \ldots, P(\omega^{n-1})$.

For *FFTRec* to be a valid recursive procedure, we must verify that it is recursively invoked with the proper type of input. It is immediate that when n is a power of two, so is $n/2$. Moreover, it is easy to verify that when ω is a primitive n^{th} root of unity and n is a power of two, then ω^2 is a primitive $(n/2)^{\text{th}}$ root of unity. Thus, $n/2$ and ω^2 are appropriate input parameters for a recursive invocation of *FFTRec*. In our pseudocode for *FFTRec* and other procedures in this chaper, we avoid using i for a loop index, to save possible confusion with $i = \sqrt{-1}$. However, we still find it convenient to use i for an integer when talking, for example, about the i^{th} leaf node in the tree of recursive calls to *FFTRec*.

```
procedure FFTRec(a[0:n – 1], n, ω, b[0:n – 1] recursive
Input: a[0:n – 1] (an array of coefficients of the polynomial P(x) =
                                   a_{n-1}x^{n-1} + . . . + a_1x + a_0)
       n (a power of two)      //n = 2^k
       ω (a primitive nth root of unity)
Output:  b[0:n – 1] (an array of values b[j] = P(ω^j,  j = 0, . . . , n – 1)
   if n = 1 then
      b[0] ← a[0]
   else
      //divide into even-indexed and odd-indexed coefficients
      for j ← 0 to n/2 – 1 do
         Even[j] ← a[2*j]          //[a_0,a_2, . . . , a_{n-2}]
         Odd[j] ← a[2*j + 1]           //[a_1,a_3, . . . , a_{n-1}]
      endfor
      FFTRec(Even[0:n/2 – 1],n/2, ω^2,e[0:n/2 – 1])
      FFTRec(Odd[0:n/2 – 1],n/2, ω^2,d[0:n/2 – 1])
      //now combine according to (7.5.5) and (7.5.6)
      for j ← 0 to n/2 – 1 do
         b[j] ← e[j] + ω^j * d[j]
         b[j + n/2] ← e[j] – ω^j * d[j]
      endfor
   endif
end FFTRec
```

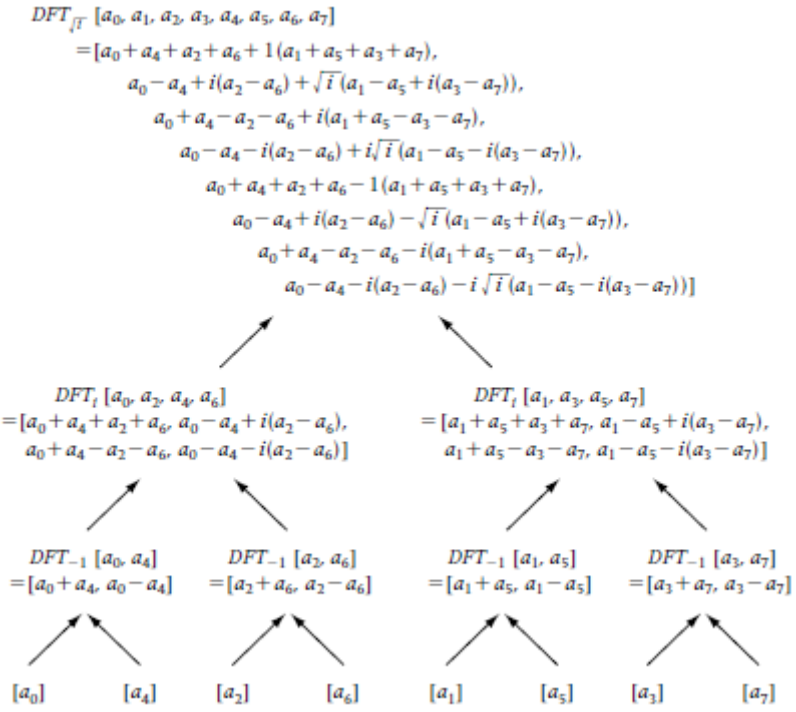
Figure 7.4 illustrates the tree of recursive calls to *FFTRec* resulting from an initial call with $n = 7$. As usual, the left (right) child of a node corresponds to the first (second) recursive call in the code. Each node in the tree lists the relevant **input** parameters for the given invocation. That is, we first list the input polynomial by listing its coefficient array, and we then list the current values of n and associated n^{th} root of unity for the given node.



Tree of recursive calls to *FFTRec* with $n = 8$

Figure 7.4

Resolution of the tree of recursive calls made by *FFTRec* with $n = 8$ is shown in Figure 7.5.



Resolutions of the recursive calls made by *FFTRec* with $n = 8$

Figure 7.5

7.5.3 An Iterative version of *FFT*

An iterative version *FFT* of *FFTRec* can be designed based on the bottom-up resolution of the tree *T* of recursive calls shown in Figure 7.5. Upon examining Figure 7.5, it is not immediately obvious for a general $n = 2^k$ which coefficient a_j corresponds to the i^{th} leaf node, $i = 0, \ldots, n-1$, in the tree of recursive calls for *FFTRec*. Fortunately, there is a simple procedure for computing the permutation $j = \pi_k(i)$ such that a_j is the coefficient corresponding to the *i*th leaf note of *T*.

Proposition 7.5.1

Suppose $n = 2^k$, and π_k: $\{0, \ldots, n-1\} \Rightarrow \{0, \ldots, n-1\}$ is the permutation such that a_j is the coefficient corresponding to the i^{th} leaf node of *T* in the tree of recursive calls of *FFTRec* with input coefficient array $a[0{:}n-1]$. Then the *k*-digit binary representation of $\pi_k(i)$ is obtained from the *k*-digit binary representation of *i* by simply reversing the digits, where leading zeros are included (if necessary).

To illustrate Proposition 7.5.1, suppose $k = 4$ ($n = 16$) and $i = 3$. The 4-digit binary representation of 3 is 0011, so that $\pi_k(3) = 1100$ in binary (in decimal, $\pi_k(3) = 12$).

Proof of Proposition 7.5.1

We prove Proposition 7.5.1 using induction on *k*.

Basis step: $k = 1$. Trivial.

Induction step: Assume that Proposition 7.5.3.1 is true for *k*. For $k + 1 = \log_2 n$, note that the leaves of the left subtree of the tree of recursive calls with input coefficients $a_0, a_1, \ldots, a_{2^{k+1}-1}$, and n^{th} root of unit ω, consists of the coefficients $a_0, a_2, \ldots, a_{2^{k+1}-2}$. Moreover, this left subtree corresponds exactly to the tree of recursive calls of *FFTRec* with input coefficients $\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_{2^k-1}$, and $(n/2)^{\text{th}}$ root of unit ω^2, where

$$\tilde{a}_i = a_{2i}, \quad i = 0, \ldots, 2^k - 1. \tag{7.5.11}$$

It follows from (7.5.11) that

$$\pi_{k+1}(i) = 2\pi_k(i), \quad i = 0, \ldots, 2^k - 1. \tag{7.5.12}$$

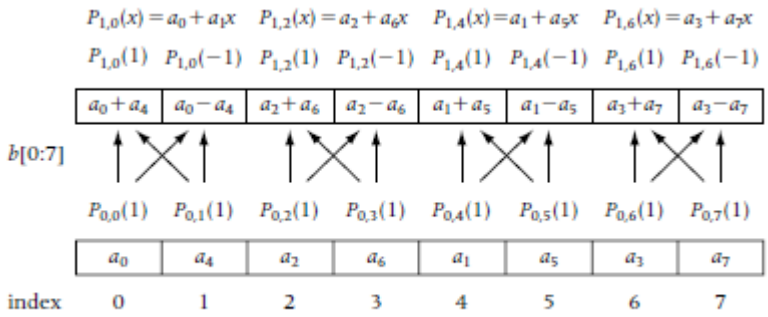
By induction hypothesis, for input coefficients $\tilde{a}_i$, $i = 0, \ldots 2^k - 1$, to *FFTRec*, the *k*-digit binary representation of $\pi_k(i)$ is obtained from the *k*-digit binary representation of *i* by simply reversing the digits. Using (7.5.12), the $(k + 1)$-digit binary representation of $\pi_{k+1}(i)$ is obtained from $\pi_k(i)$ by adding a zero on the right. Since the $(k + 1)$-digit binary representation of *i* is obtained by from the *k*-digit binary representation of *i* by adding a leading zero, we see that Proposition 7.5.1 holds for $\pi_{k+1}(i)$, $i = 0, \ldots, 2^k - 1$. The proof that Proposition 7.5.1 holds for $\pi_{k+1}(i)$, $i = 2^k, \ldots, 2^{k+1} - 1$ is similar (using the right subtree) and is left as an exercise. ■

The pseudocode for a procedure *ReverseBinPerm*($R[0{:}n-1]$), which computes $R[i] = \pi_k(i)$, $i = 0, \ldots, n-1$, is easy to write. An iterative version of *FFT* begins by loading an array $b[0{:}n-1]$ with the values $b[i] = a_{\pi_k(i)} = a[R[i]]$, $i = 0, \ldots, n-1$. The values $b[i]$ correspond to the leaves in the tree T of recursive calls to *FFTRec*, and the iterative version of *FFT* proceeds by resolving these calls level-by-level in a bottom-up fashion. With the aid of Figure 7.5, you might want to proceed directly to the pseudocode for *FFT* given at the end of this subsection. However, it is instructive to describe the polynomial evaluations that take place at each level. The leaves are at level k in T and correspond to evaluating the constant polynomials $P_{0,i}(x) = a_{\pi_k(i)}$, $i = 0, \ldots, n-1$, at $(\omega^n)^0 = 1$. At level $k-j$, $j \in \{1, \ldots, k\}$, the iterative version of *FFT* evaluates $n/2^j$ polynomials $P_{j,s}$, $s = 0, 2^j, \ldots, n-2^j$ of degree $2^j - 1$ at the points $1, \omega^{n/2^j}, \ldots, (\omega^{n/2^j})^{2^{j-1}-1}, -1, -\omega^{n/2^j}, \ldots, -(\omega^{n/2^j})^{2^{j-1}-1}$. In this notation, the splitting of the polynomial $P_{j,s}$ into even and odd powers corresponds precisely to the two polynomials $P_{j-1,s}$ and $P_{j-1,s+2^{j-1}}$, respectively, at the next lower level in T. Hence, (7.5.5) and (7.5.6) become (see Figure 7.6)

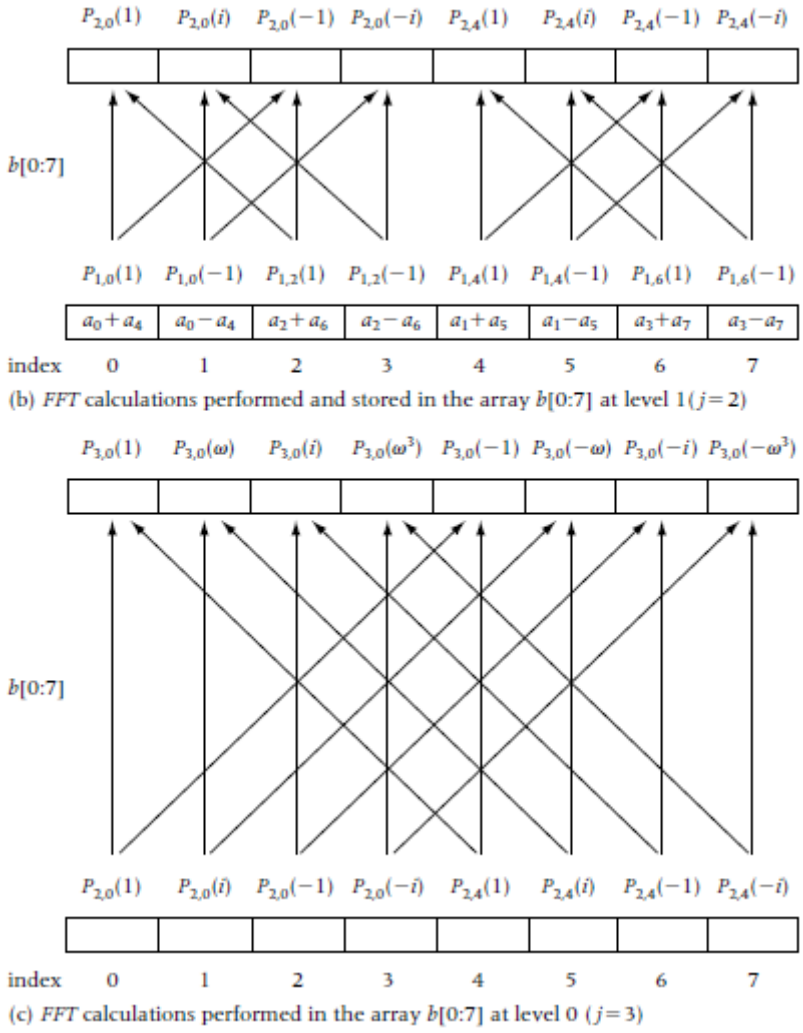
$$P_{j,s}(x) = P_{j-1,s}(x^2) + xP_{j-1,s+2^{j-1}}(x^2), \tag{7.5.13}$$

$$P_{j,s}(-x) = P_{j-1,s}(x^2) - xP_{j-1,s+2^{j-1}}(x^2), \tag{7.5.14}$$

$$j = 1, \ldots, k, \quad s = 0, 2^j, \ldots, n-2^j, \quad x = 1, \omega^{n/2^j}, \ldots, (\omega^{n/2^j})^{2^{j-1}-1}.$$



(a) *FFT* calculations performed and stored in the array $b[0{:}7]$ at level $2(j=1)$



Level-by-level *FFT* computations for $n = 8$

Figure 7.6

The iterative version of *FFT* proceeds by computing (7.5.13) and (7.5.14) with three nested **for-do** loops, the outer loop controlled by $j = 1, \ldots, k$, the middle loop controlled by $s = 0, 2^j, \ldots, n - 2^j$, and the innermost loop controlled by $x = 1, \omega^{n/2^j}, \ldots, (\omega^{n/2^j})^{2^{j-1}-1}$. The calculations at level $k - j$ begin with $b[0{:}n-1]$ containing the previously calculated values.

$$b[s+m] = P_{j-1,s}((\omega^{n/2^{j-1}})^m), \qquad (7.5.15)$$
$$s = 0, 2^{j-1}, \ldots, n-2^{j-1}, \quad m = 0, 1, \ldots, 2^{j-1}-1$$

The values corresponding to level $k - j$ are then calculated by first using an auxiliary array $Temp[0{:}n-1]$ to receive the results of calculations in the right-hand sides of (7.5.13) and (7.5.14).

$$Temp[s+m] = b[s+m] + (\omega^{n/2^j})^m b[s+m+2^{j-1}], \qquad (7.5.16)$$

$$Temp[s+m+2^{j-1}] = b[s+m] - (\omega^{n/2^j})^m b[s+m+2^{j-1}], \qquad (7.5.17)$$
$$j = 1,\ldots,k, \quad s = 0,2^j,\ldots,n-2^j, \quad m = 0,1,\ldots,2^j-1$$

Formula (7.5.17) follows from (7.5.14) and the fact that $-b[s+m] = b[s+m+2^{j-1}]$. Then the results in $Temp[0{:}n-1]$ are copied back to $b[0{:}n-1]$.

The level-by-level bottom-up calculations are illustrated in Figure 7.6, where we show using arrows that the new value of $b[s+m]$ is determined from the previously calculated values of $b[s+m]$ and $b[s+m+2^{j-1}]$. Due to space limitations, we only show the initial values of $b[0{:}7]$ and its values after one iteration. However, all the values at each level can be determined by examining Figure 7.5.

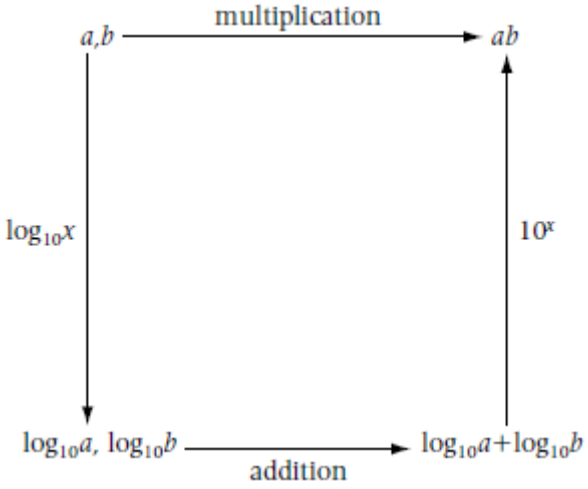
The pseudocode for the iterative version *FFT* is based on the bottom-up strategy described above.

```
procedure FFT(a[0:n – 1], n, ω, b[0:n – 1])
Input: a[0:n – 1] (an array of coefficients of the polynomial P(x) =
                                        a_{n-1}x^{n-1} + . . . + a_1x + a_0)
       n (a positive integer)        //n = 2^k
       ω (a primitive n^th root of unity)
Output:  b[0:n – 1] (an array of values b[j] = P(ω^j), j = 0, . . . , n – 1)
   call ReverseBinPerm(R[0:n – 1])
   for j ← 0 to n – 1
      b[j] ← a[R[j]]
   endfor
   for j ← 1 to k do
      for s ← 0 to n – 2^j by 2^j do
         for m ← 0 to 2^{j-1} – 1
            Temp[s + m] ← b[s + m] + (ω^{n/2^j})^m * b[s + m + 2^{j-1}]
            Temp[s + m + 2^{j-1}] ← b[s + m] – (ω^{n/2^j})^m * b[s + m + 2^{j-1}]
         endfor
      endfor
      for j ← 0 to n – 1 do
         b[j] ← Temp[j]
      endfor
   endfor
end FFT
```

7.5.4 Transforming the Problem Domain

In secondary school mathematics, you were introduced to the idea of transforming problems into equivalent problems that are simpler or easier to solve. For example, it is often convenient to deal with large numbers by transforming them using logarithms to a certain base, say, base 10. The problem of multiplying two large numbers *a* and *b* is then transformed into the simpler but equivalent problem of adding their logarithms and then using the inverse transformation (exponentiation) to compute the given product. The idea

behind this transformation can be nicely captured by the commutative diagram shown in Figure 7.7.



Commutative diagram for transforming multiplication via logarithms

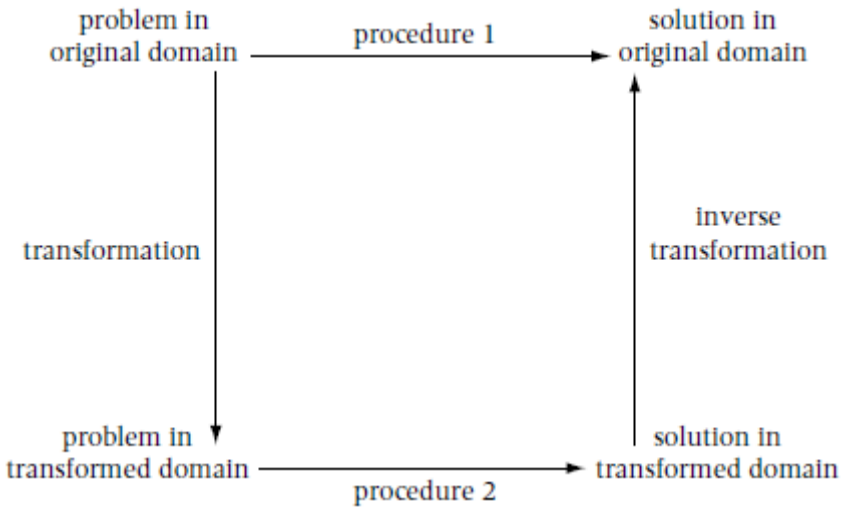
Figure 7.7

The diagram given in Figure 7.7 is called commutative since the product *ab* can be computed either directly by following the top arrow or indirectly by following the three arrows along the sides and bottom of the diagram. The mathematical expression of this commutativity is simply the equation

$$ab = 10^{\log_{10} a + \log_{10} b}$$

The utility of this transformation resides in the fact that addition can be carried out more quickly than multiplication. It is precisely to make the above transformations useful in practice that extensive tables of logarithms and antilogs have been generated.

The general idea behind transforming a given problem can be captured by the generic commutative diagram shown in Figure 7.8. In order for this transformation to be worthwhile, the transformation to the new problem domain and its inverse must both be done more efficiently than solving the problem in the original domain. Also, the problem in the transformed domain must be easier to solve than the problem in the original domain.



Generic commutative diagram for transforming problems

Figure 7.8

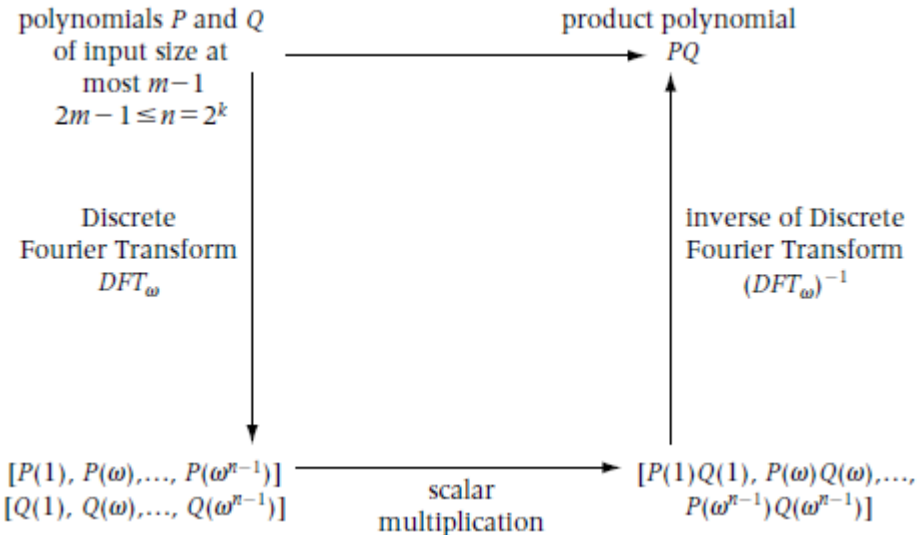
There is a host of important applications, such as signal and image processing, where the Discrete Fourier Transform (*DFT*) can be used to transform the problem domain into an equivalent, but (algorithmically) simpler, problem domain. However, in order for this transformation to be effective, the *DFT* and its inverse need to be computed efficiently. As we will see, the inverse of a *DFT* is actually (except for a multiplicative factor) just another *DFT*! Fortunately, the *FFT* provides an efficient algorithm to compute *DFT*s, and therefore efficiently transform the problem to the new domain and transform back to the original domain. The problem that we will use to illustrate *DFT* transformation of the domain is (symbolic) polynomial multiplication. This problem will be transformed into the simpler problem on multiplying n numbers.

7.5.5 The Inverse Fourier Transform and Fast Polynomial Multiplication

We have mentioned that the *FFT* can be used to design an $O(n \log n)$ algorithm for polynomial multiplication. Of course, *FFT* worked on polynomials whose input size is a power of two. The product of two polynomials both of input size m is a polynomial of input size $n = 2m - 1$. Thus, for convenience in discussing polynomial multiplication, we assume that n is a power of two and that we are multiplying two polynomials of input size m, where $n = 2m - 1$. In practice, when multiplying any two polynomials we can arrange for these conditions by adding leading terms with zero coefficients, if necessary.

Figure 7.9 shows how the Fourier Transform can be utilized to obtain the product polynomial $P(x)Q(x)$. We first apply the Fourier Transform to find the coefficient arrays $[P(1), P(\omega), \ldots, P(\omega^{n-1})]$ and $[Q(1), Q(\omega), \ldots, Q(\omega^{n-1})]$ of both $P(x)$ and $Q(x)$. Then we simply compute the coefficient array of the product polynomial $P(x)Q(x)$ by forming the n scalar products $P(1)Q(1), P(\omega)Q(\omega), \ldots, P(\omega^{n-1})Q(\omega^{n-1})$. Thus by following the left side and bottom of the commutative diagram in Figure 7.9 we have actually computed the Fourier Transform of the product polynomial $P(x)Q(x)$. Hence, to recover $P(x)Q(x)$ we need to be able to perform the inverse to the Fourier Transform. The inverse

certainly exists, since it amounts to finding the polynomial interpolating the n points $(1, P(1)Q(1)), (\omega, P(\omega)Q(\omega)), \ldots, (\omega^{n-1}, P(\omega^{n-1})Q(\omega^{n-1}))$.



Commutative diagram for computing product of two polynomials using DFT_ω

Figure 7.9

The left side of the diagram in Figure 7.9 can be computed using *FFT* with $O(n\log n)$ complexity. The bottom of the diagram can be computed with $O(n)$ complexity. However, the most straightforward interpolation algorithms have complexity $O(n^2)$. Hence, to arrive at an $O(n\log n)$ algorithm for polynomial multiplication we need to look for an $O(n\log n)$ algorithm to compute the inverse Discrete Fourier Transform DFT_ω^{-1}. Fortunately, we have the following useful key fact about the inverse Discrete Fourier Transform.

Key Fact

To compute the inverse of a Discrete Fourier Transform we simply need to compute another Discrete Fourier Transform.

The key fact results from the following elegant formula for the inverse namely $DFT_{\omega^{-1}}$:

$$DFT_\omega^{-1} = \left(\frac{1}{n}\right) DFT_{\omega^{-1}}. \tag{7.5.18}$$

To verify (7.5.18), it is useful to utilize the array and matrix notation discussed earlier for the Discrete Fourier Transform DFT_ω. Then (7.5.18) is equivalent to verifying that the inverse of the matrix A_ω defined by (7.5.19) is given by the formula

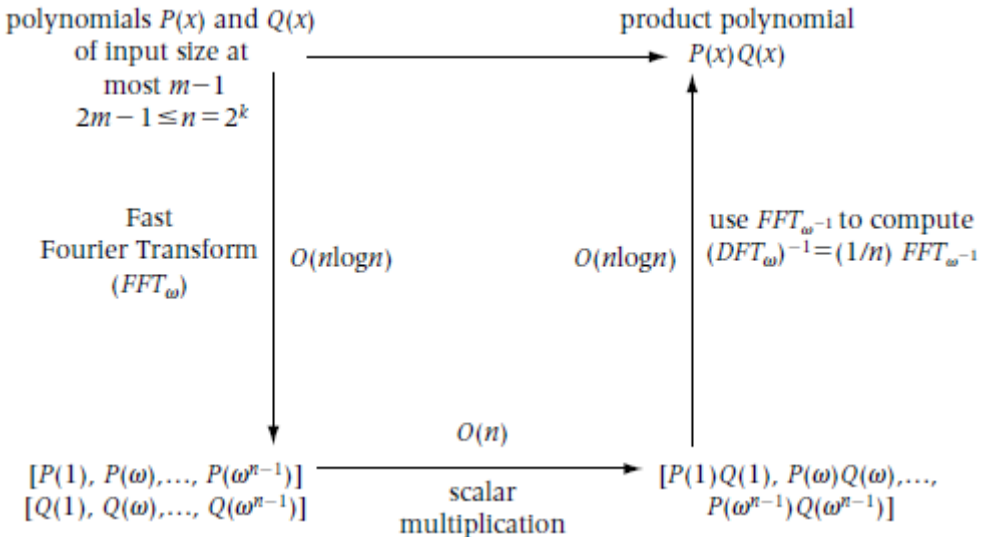
$$F_\omega^{-1} = \left(\frac{1}{n}\right) f_{\omega^{-1}}. \tag{7.5.19}$$

Equivalently, we must verify that

$$F_{\omega}^{-1} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ 1 & \omega^{-(n-1)} & (\omega^{-2})^{n-1} & \dots & (\omega^{-(n-1)})^{n-1} \end{bmatrix} \tag{7.5.20}$$

We leave the proof of (7.5.10) to the exercises.

The commutative diagram in Figure 7.10 summarizes the application of *FFT* for $O(n\log n)$ polynomial multiplication. In Figure 7.10, we denote the application of *FFT* with input parameter ω by FFT_{ω}.



Commutative diagram summarizing $O(n\log n)$ polynomial multiplication using FFT_{ω}

Figure 7.10

We illustrate the actions of the commutative diagram in Figure 7.10 to compute the product of the polynomials $P(x) = x^3 - x + 2$ and $Q(x) = 2x^2 - 1$. The product polynomial has degree 5, so we taken $n = 7$. Then, $\omega = \sqrt{i} = (1+i)/\sqrt{2}$. Using *FFT* yields the evaluations

$$[P(1),P(\omega),\dots,P(\omega^7)] = [2,2-\omega+\omega^3,2-2i,2+\omega-\omega^3,2,2+\omega-\omega^3,$$
$$2+2i,2-\omega+\omega^3]$$
$$[Q(1),Q(\omega),\dots,Q(\omega^7)] = [1,2i-1,-3,-2i-1,1,2i-1,-3,-2i-1].$$

Next we form the pointwise products $P(i)Q(i)$, $i = 1, \omega, \ldots, \omega^7$ and acquire the coefficient array for $DFT_\omega(P(x)Q(x))$ given by

$$[2, 4i - 3\omega^3 - \omega - 2, 6i - 6, -4i - \omega^3 - 3\omega - 2, 2, 4i + 3 + \omega - 2, -6i - 6, -4i + \omega^3 + 3\omega - 2]. \quad (7.5.21)$$

To complete the illustration of computing the product polynomial $P(x)Q(x)$ using the commutative diagram, we use *FFT* to compute $(DFT_\omega)^{-1} = (1/8)\, DFT_{\omega^{-1}}$ for the polynomial (coefficient array) given by (7.5.21). This yields the coefficient array [–2, 1, 4, –3, 0, 2, 0, 0] of the product polynomial

$$P(x)Q(x) = 2x^5 - 3x^3 + 4x^2 + x - 2.$$

7.6 Two Classical Problems in Computational Geometry

We now use the divide-and-conquer technique to solve two fundamental problems in computational geometry, the *closest-pair* problem, and the *convex hull* problem. While both problems can be posed for points in Euclidean d-dimensional space for any $d \geq 1$, we limit our discussion of the solution to these problems to the line and the plane ($d = 1$ and 2, respectively).

7.6.1 The Closest-Pair Problem

In the closest-pair problem, we are given n points $P_1,\ldots,P_n$ in Euclidean d-space, and we wish to determine a pair of points P_i and P_j such that the distance between P_i and P_j is minimized over all pairs of points drawn from $P_1,\ldots,P_n$. Of course, a brute-force quadratic complexity algorithm solving this problem is obtained by simply computing the distance between each of the $n(n-1)/2$ pairs and recording the minimum value so computed. Actually to avoid round-off problems associated with taking square roots, it would be best to work with the square of the distance, which we assume throughout the discussion (and which, by abuse of language and notation, we still refer to as simply the distance d).

Using divide-and-conquer, we now design a $O(n\log n)$ algorithm for the closest-pair problem. This turns out to be order optimal, since there is a $\Omega(n\log n)$ lower bound for the problem. We describe the algorithm informally, since the actual pseudocode, while straightforward, is somewhat messy to fully describe.

To motivate the solution to the problem in the plane, we first consider the problem for points on the real line. Of course, we can solve the problem on the line by sorting the points using an $O(n\log n)$ algorithm, and then simply making a linear scan of these sorted points. However, this method does not generalize to the plane. To find a method that does generalize, let m be the median value of the n points. Using the algorithm *Select2* described in Section 7.3, m can be computed in linear time. Divide the points $x_1,\ldots,x_n$ into two subsets X_1, X_2 of equal size, those less than or equal to the median, and the remaining points (a linear scan determines the division). Let d_1 and d_2 be the minimum distances between pairs of points in X_1 and X_2, respectively. Now either d

$= \min\{d_1,d_2\}$ is also the smallest distance between any pair drawn from $x_1,\ldots,x_n$, or there is a pair $x_i \in X_1$ and $x_j \in X_2$ having a strictly smaller distance than d. However, it is easy to check (exercise) that the interval $(m - d\ ,\ m]$ contains at most one point of X_1, and the interval $(m,\ m + d]$ contains at most one point of X_2. Hence, in linear time it can be determined if the only possible pair have closer distance than d actually exists. Thus, for $n = 2^k$ the complexity $W(n)$ of the algorithm satisfies the recurrence

$$W(n) = 2W(n/2) + n, \qquad \textbf{init. cond. } W(1) = 0,$$

which unwinds to yield $W(n) \in O(n\log n)$.

The above divide-and-conquer solution for points on the real line generalizes naturally to a divide-and-conquer solution in the plane by dividing the n points $(x_1,y_1),\ldots,(x_n,y_n)$ into sets X_1 , X_2 on either side of line $x = m$, where m is the median of the x-coordinates of the points. The sets X_1 , X_2 can be determined with $O(n\log n)$ complexity by sorting the points by their x-coordinates. We then recursively find the minimum distances d_1 and d_2 between pairs of points in the sets X_1 and X_2 , respectively. Now again, either $d = \min\{d_1,d_2\}$ is also the smallest distance between any pair drawn from $(x_1,y_1),\ldots,(x_n,y_n)$, or there is a pair $(x_i,y_i) \in X_1$ and $(x_j,y_j) \in X_2$ having a strictly smaller distance than d. Also, if there is such a pair, then (x_i,y_i) lies in the strip S_1 determined by the lines $x = m - d$ and $x = d$, whereas (x_j,y_j) lies in the strip S_2 determined by the lines $x = d$ and $x = m + d$. However, unlike the case for the line, we can no longer be sure that there is at most a single pair of points to examine. Indeed, *all* the points $(x_1,y_1),\ldots,(x_n,y_n)$ might be in the strip $S = S_1 \cup S_2$ between the lines $x = m - d$ and $x = m + d$, so that we would have to examine a quadratic number of pairs, which is no better than the brute force solution! However, the following proposition, whose proof we leave as an exercise, comes to our rescue.

Proposition 7.6.1 Consider a rectangle R in the plane of width d and height $2d$. There can only be at most six points in R such that the distance between each pair of these points is at least d. □

The following Key Fact follows from Proposition 7.6.1 and the definition of d.

Key Fact

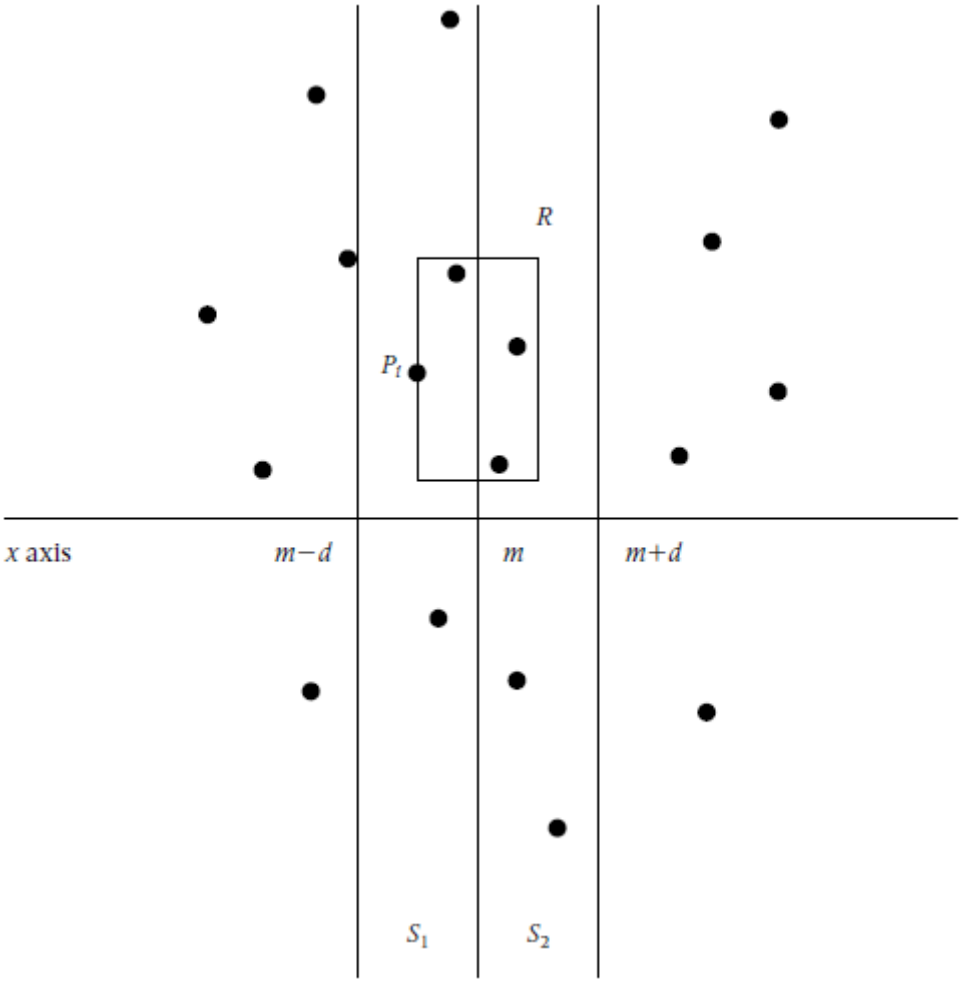
For each point $(x,y) \in X_1 \cap S_1$ there are at most five points in $X_2 \cap S_2 \cap R$, where R is the rectangle with corner points $(x,y - d)$, $(x,y + d)$, $(x + d,\ y - d)$, $(x + d,\ y + d)$ (see Figure 7.11).

The key fact allows us to check less than $5n$ pairs to determine whether or not there is a pair $(x_i,y_i) \in X_1 \cap S_1$ and $(x_j,y_j) \in X_2 \cap S_2$ having distance less than d. To see this, note that we can require our recursive calls determining X_1,d_1, and X_2,d_2 , to return X_1 and X_2 sorted by their y-coordinates, which then can be merged using no more that $n - 1$ comparisons by the procedure *Merge* discussed in Chapter 2. Thus, as we scan through the points in $X_1 \cap S_1$ in increasing order of their y-coordinates, a corresponding pointer can also scan the points in $X_2 \cap S_2$ in (slightly oscillatory) increasing order of their y-

coordinates checking at most 5n pairs. More precisely, suppose $P_1,\ldots,P_m$ (respectively, $Q_1,\ldots,Q_k$) are the points in $X_1 \cap S_1$ (respectively, in $X_2 \cap S_2$). We first scan $X_2 \cap S_2$ until we find a point (if any) such that d added to its y-coordinate is at least as large as P_1's y-coordinate. If we find such a point Q_i, then we leave a pointer Q at Q_i, and using Proposition 7.6.1 we need only check Q_i and the next four points $X_2 \cap S_2$ in order to see if d needs updating. We then move to point P_2, and resume the scan of $X_2 \cap S_2$ starting at Q_i, this time looking for a point whose y-coordinate is at least as large as P_2's y-coordinate. We repeat this process until all of $X_1 \cap S_1$ is scanned, and less than 5n pairs of points will be examined for updates to the current smallest distance. Since we make most $n - 1$ comparisons when merging X_1 and X_2, we see that the combine step in our divide-and-conquer algorithm performs less than 6n comparisons altogether. Hence, the worst case $W(n)$ of the algorithm satisfies

$$W(n) < 2W(n/2) + 6n, \qquad \textbf{init. cond. } W(1) = 0,$$

which shows that $W(n) \in O(n\log n)$.



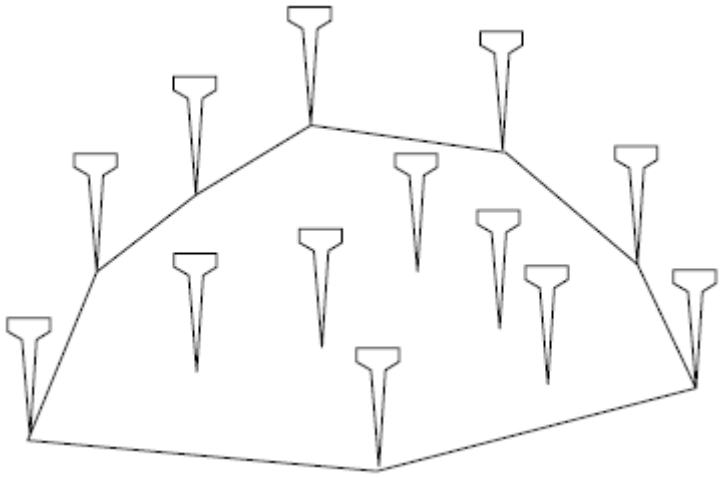
For $P_i \in X_1 \cap S_1$, rectangle R of width d and height 2d is shown where distances from P_i to points in $X_2 \cap S_2 \cap R$ need to be checked

Figure 7.11

When implementing the above algorithm, there are various degenerate cases that have to be handled. For example, it might happen that some (or even all) of the points are on the line $x = m$. We leave the implementation details to the exercises.

7.6.2 The Convex Hull Problem

Given any subset S of the Euclidean plane, S is called convex if for each pair of points P_1, $P_2 \in S$, the line segment joining P_1 and P_2 lies entirely within the set S. Give a set of n points $(x_1,y_1), \ldots, (x_n,y_n)$ in the plane, the *convex hull* of these points is the smallest convex set containing them. In other words, the convex hull of the points is contained in any convex set containing the points. There is a nice physical interpretation of the convex hull. Consider placing pegs (or golf tees) at each of the n points $(x_1,y_1), \ldots, (x_n,y_n)$. Stretching a small rubber band about the entire set of points and releasing the band determines the boundary of the convex hull (see Figure 7.12).



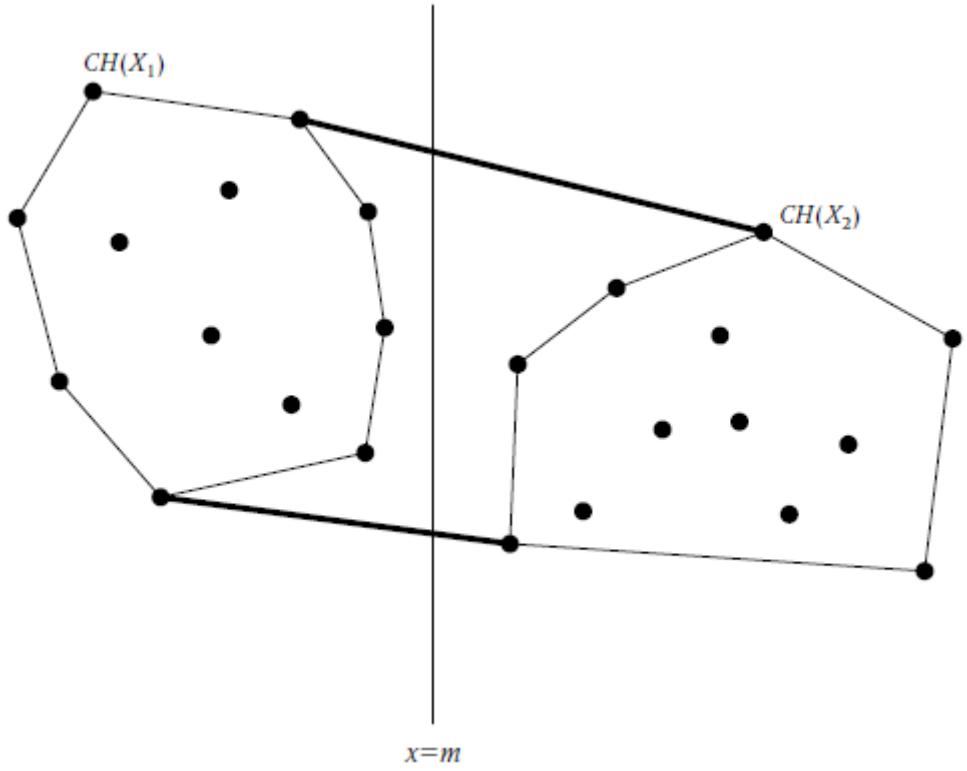
A small rubber band stretched and released around pegs at points in the plane determines convex hull of these points

Figure 7.12

Given the points $(x_1,y_1),\ldots,(x_n,y_n)$ in the plane, the convex hull problem is to determine a subset of these points $P_1,\ldots,P_k,P_{k+1} = P_1$ such that the boundary of the convex hull of the points $(x_1,y_1),\ldots,(x_n,y_n)$ consists of the line segments joining P_i and P_{i+1}, $i = 1,\ldots,k$. We assume that no three consecutive points in the (circular) list $P_1,\ldots,P_k,P_{k+1} = P_1$ are collinear.

The divide-and-conquer algorithm for computing the convex hull that we now describe begins in the same way as the closest pair algorithm; namely, we divide the n points $(x_1,y_1),\ldots,(x_n,y_n)$ into sets X_1,X_2 on either side of line $x = m$, where m is the median of the x-coordinates of the points. We then recursively determine the convex hulls $CH(X_1)$ and $CH(X_2)$, of X_1 and X_2, respectively. The initial condition for the recursion is

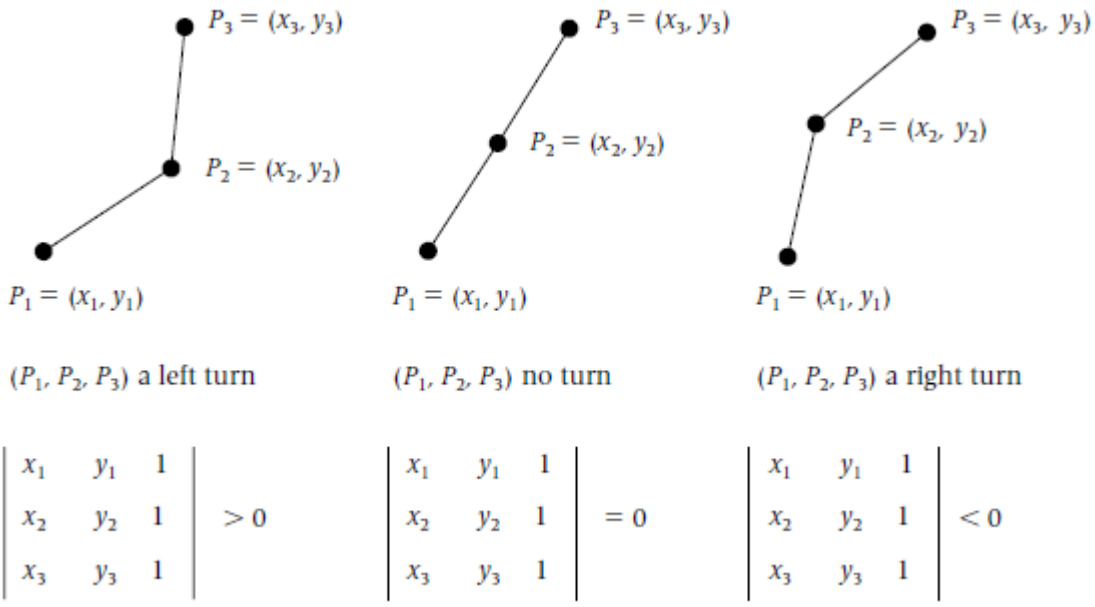
a set of one, two, or three points, since the convex hull in these cases is a point, a line segment connecting the two points, and a triangle connecting the three points (unless they are collinear), respectively. It then becomes a question of how to merge the convex hulls $CH(X_1)$, $CH(X_2)$ into the convex hull $CH(X_1 \cup X_2)$. The answer is to determine the upper and lower support line segments of $CH(X_1) \cup CH(X_2)$, as illustrated in Figure 7.13.



Upper and lower support line segments of $CH(X_1) \cup CH(X_2)$

Figure 7.13

In order to determine the support line segments, we introduce the notion of a turn determined by an ordered triple of points (P_1,P_2,P_3) in the plane. We have a left turn, no turn, or a right turn, respectively, as determined by the determinant shown in Figure 7.14. This determinant corresponds to twice the "signed area" determined by the triple of points (P_1,P_2,P_3).



The indicated determinant determines whether the triple (P_1,P_2,P_3) is a left turn, no turn, or a right turn; that is, the turn depends on whether this determinant is positive, zero, or negative, respectively.

Figure 7.14

We now use the notion of turns to determine the upper support line segment. Suppose $CH(X_1)$ is given by $P_1,\ldots,P_k$, $P_{k+1} = P_1$and $CH(X_2)$ is given by $Q_1,\ldots,Q_m,Q_{m+1} = Q_1$ where both sequences are in clockwise order (that is, you make right turns as you pass through the sequences). Let P_i be the point in X_1 with largest x-coordinate (if there are two such points, take the point with the smaller y-coordinate). Consider the sequence of turns determined by (P_i,Q_j,Q_{j+1}), $j = 1,..,m$, where we set $Q_{m+1} = Q_1$ for convenience. Then the right-hand endpoint of the upper support line segment is the point Q_r where the turns go from left turns or no turn to a right turn; that is, (P_i,Q_{r-1},Q_r) is a left turn or no turn, and (P_i,Q_r,Q_{r+1}) is a right turn. Now consider the sequence of turns (Q_r,P_j,P_{j-1}), $j = 1,\ldots,k$, where we set $P_0 = P_m$ for convenience (that is, we go around $CH(X_1)$ in counter-clockwise order). Then the left-hand endpoint of the upper support line segment is the point P_s where the turns go from right turns or no turn to a left turn. This determines the upper support segment. The lower support segment is obtained similarly.

Given the two support line segments, it is a simple matter to eliminate the points in $CH(X_1) \cup CH(X_2)$ that are not in $CH(X_1 \cup X_2)$, and to return the sequence of points in $CH(X_1 \cup X_2)$ in clockwise order. It is also easy to verify that the algorithm has $O(n\log n)$ complexity. As with the closest pair problem, when implementing the algorithm, various degenerate cases have to be handled. We leave the implementation details to the exercises.

There are other $O(n\log n)$ algorithms for determining the convex hull that are not based on divide-and-conquer. Two of the most commonly used of these are the Graham scan and the Jarvis march. We refer you to the references for a discussion of these and other convex hull algorithms. We point out there is a $\Omega(n\log n)$ lower bound for the

convex hull problem in the plane, since sorting can be reduced to this problem (see Exercise 7.32). Hence, the above divide-and-conquer algorithm, and the algorithms of Graham and Javis, all have optimal order of complexity.

7.7 Closing Remarks

Since Strassen's result for matrix multiplication appeared in 1970, researchers have been trying to improve the basic method. Identities are sought that perform fewer multiplications when multiplying matrices of a certain fixed size m. Then this reduction is used as the basis of a divide-and-conquer algorithm using decomposition into m^2 blocks of size n/m, where we assume for convenience that $n = m^k$ for some positive integer k. It has been shown that for 2×2 matrices, at least seven multiplications are *required*. Thus, divide-and-conquer algorithms, which use as their starting point a method of multiplying two $m \times m$ matrices using less than the number of multiplications used by Strassen's method, require that m be larger than 2.

Nearly ten years after Strassen discovered his identities, Pan found a way to multiply two 70×70 matrices that involves only 143,640 multiplications (compared to over 150,000 multiplications used by Strassen's method), yielding an algorithm that performs $O(n^{2.795})$ multiplications. Improvements over Pan's algorithm have been discovered, and the best result currently known (due to Coppersmith and Winograd) can multiply two $n \times n$ matrices using $O(n^{2.376})$ multiplications. However, all of these methods require n to be quite large before improvements over Strassen's method are significant. Moreover, they are very complicated due to the large number of identities required to achieve the savings in the number of multiplications. Hence, the currently known order of complexity improvements over Strassen's algorithm are mostly of theoretical rather than of practical interest.

There are a number of other transformations related to the Discrete Fourier Transform that are also often used in practice. For example, an important transform known as the Discrete Cosine Transformation (DCT) uses only cosines, instead of both sines and cosines used by the Fourier transform. For example, the DCT is used in JPEG compression of graphics files in combination with Huffman coding.

The subject of computational geometry has a vast literature, and is an active area of research. We refer you to the references at the end of this chapter for further reading on this important topic.

Exercises

Section 7.1 The Divide-and-Conquer Paradigm

7.1 Design and analyze a divide-and-conquer algorithm for finding the maximum element in a list $L[0{:}n-1]$.

7.2 Design and analyze a divide-and-conquer algorithm for finding the maximum and minimum elements in a list $L[0{:}n-1]$.

7.3 Suppose we have a list $L[0{:}n-1]$ representing the results of an election, so that $L[i]$ is the candidate voted for by person i, $i = 0,...,n-1$. Design a linear algorithm

to determine whether a candidate got a majority of the votes, that is, whether there exists a list element that occurs more than $n/2$ times in the list.

7.4 Design a version of *QuickSort* that uses a threshold. Do some empirical testing to determine a good threshold value.

Section 7.2 Symbolic Algebraic Operations on Polynomials

7.5 Give pseudocode and analyze the procedure *DirectPolyMult*, which is based directly on (7.2.1).

7.6 Repeat Exercise 7.5 when the polynomials are implemented by storing the coefficients in

a) an array

b) a linked list

c) consider the sparse case

7.7 Give pseudocode for a version of *PolyMult1* when the polynomials are implemented by storing the coefficients and the associated powers of the polynomials in a linked list.

7.8 When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation. This ignores the two multiplications by powers of x. Show how the procedure for multiplying a polynomial by x^i can be implemented with linear complexity. Taking this operation into account, verify that the order of complexity of *PolyMult1* remains unchanged.

7.9 Show that the divide-and-conquer algorithm for multiplying two polynomials based on the recurrence relation (7.2.2) has quadratic complexity.

Section 7.3 Multiplication of Large Integers

7.10 Design and analyze an algorithm *MultInt* for multiplying large integers.

7.11 a. Design an algorithm for adding two large integers implemented using arrays.

b. Repeat part (a) for linked list implementations.

7.12 Give pseudocode for the algorithm *MultInt* you designed in Exercise 7.10 for the following two implementations of large integers.

a. arrays

b. linked lists

Section 7.4 Multiplication of Matrices

7.13 Verify formula (7.4.4).

7.14 Verify formula (7.4.11).

7.15 Verify that the evaluation of formulas (7.4.10) and (7.4.11) requires 15 distinct additions or subtractions, which is three less than what is performed when using Strassen's identities.

7.16 Give pseudocode for the procedure *Strassen*, which implements Strassen's matrix multiplication algorithm. Assume that the input matrices A and B are both $n \times n$ matrices, where n is a power of 2.

7.17 Demonstrate the action of the procedure *Strassen* in Exercise 7.16 for the following input matrices.

$$A = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 4 \\ 3 & -4 & 5 & 7 \\ 0 & 1 & 2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 3 & 0 & 5 & 0 \\ 6 & 1 & 4 & 0 \end{bmatrix}$$

Section 7.5 The Discrete Fourier Transform

7.18 Verify formula (7.5.2).

7.19 Show that when ω is a primitive *n*th root of unity and *n* is a power of two, then ω^2 is a primitive $(n/2)^{th}$ root of unity.

7.20 Give the tree of recursive calls to *FFTRec* for $n = 16$.

7.21 Write a program implementing *FFTRec*, and run for various inputs.

7.22 For $i = 1, \ldots, n = 2^k$, consider the permutation $\pi_k(i) = j$ such that a_j is the coefficient corresponding to the i^{th} leaf node of the tree of recursive calls to *FFTRec*. Complete the inductive proof that the *k*-digit binary representation of $\pi_k(i)$ is obtained from the *k*-digit binary representation of *i* by simply reversing the digits, where leading zeros are included (if necessary).

7.23 Design and analyze the procedure *ReverseBinPerm*(*R*[0:*n* – 1]), which computes $R[i] = \pi_k(i)$, $i = 0, \ldots, n-1$.

7.24 Write a program implementing the iterative version of *FFT*, and run for various inputs.

Section 7.5.5 **The Inverse Fourier Transformation and Fast Polynomial Multiplication**

7.25 Using x.x verify that DFT^{-1} =

7.5.8 Given the polynomials $P(x) = x^3 - x + 2$, $Q(x) = 2x^2 - 1$,

a. Show that going along the left-hand side of the commutative diagram in Figure 7.10 and using *FFT* yields the evaluations:

$$[P(1), P(\omega), \ldots, P(\omega^7)] = [2, 2-\omega+\omega^3, 2-2i, 2+\omega-\omega^3, 2, 2+\omega-\omega^3, 2+2i, 2-\omega+\omega^3]$$

$$[Q(1), Q(\omega), \ldots, Q(\omega^7)] = [1, 2i-1, -3, -2i-1, 1, 2i-1, -3, -2i-1].$$

b. Verify that the polynomial $P(x)Q(x)$ results from going along the right-hand side of the commutative diagram in Figure 7.10 by using *FFT* to compute $(DFT_\omega)^{-1}) = (1/8)DFT_{\omega^{-1}}$.

Additional Exercises from Appendix A

7.54 Prove Proposition A.1 from Appendix A.

7.55 Prove Proposition A.2 from Appendix A.

7.56 Prove Proposition A.3 from Appendix A.

7.57 Give a complex number $z = x + iy$, its *complex conjugate* $\bar{z}$ is defined by $\bar{z} = x - iy$.

a. Give a geometric interpretation of $\bar{z}$.

b. Show that the complex conjugate has the following properties:

i. $\overline{-z} = -\bar{z}$,

ii. $\overline{z_1 + z_2} = \bar{z_1} + \bar{z_2}$,

iii. $\overline{z_1 z_2} = \bar{z_1}\,\bar{z_2}$.

c. Use part (b) to show that the complex roots of a polynomial $P(x)$ with real coefficients occur in conjugate pairs; that is, if $P(z) = 0$, then $P(\bar{z}) = 0$.

7.58 Verify that the n^{th} roots of z defined by (A.31) from Appendix A are all distinct.

Section 7.6 Two Classical Problems in Computational Geometry

8.27 Prove that a rectangle of width d and height $2d$ can contain at most 6 points whose pair-wise distances are at least d.

8.28 Write a computer program implementing the closest pair algorithm.

8.29 Verify that the divide-and-conquer algorithm for the convex hull problem discussed in Section 7.6. has $O(n\log n)$ complexity.

8.30 Show that the convex hull of a set of points P_1, P_2, …, P_n in the plane consists of the set of all points $\lambda_1 P_1 + \lambda_2 P_2 + \ldots + \lambda_n P_n$, where $\lambda_1, \lambda_2, \ldots, \lambda_n$ are all non-negative real numbers such that $\lambda_1 + \lambda_2 + \ldots + \lambda_n = 1$.

8.31 Write a computer program implementing the divide-and-conquer algorithm for the convex hull problem discussed in Section 7.6.

8.32 Show that a lower bound in $\Omega(n\log n)$ exists for the convex hull problem in the plane. Hint: Let $x_1, \ldots, x_n$ be n real numbers. Consider the convex hull $P_1, \ldots, P_m$ of the n points $(x_1, (x_1)^2), \ldots, (x_n, (x_n)^2)$, listed in counter-clockwise order, and where P1 has minimum x-coordinate. Show that $m = n$ and the x-coordinates of $P_1, \ldots, P_m$ (in that order) is a sorting of $x_1, \ldots, x_n$ in increasing order.

Section 7.7 Closing Remarks

7.33 Pan's divide-and-conquer matrix multiplication algorithm is based on a partitioning scheme that assumes n is a power of 70. The complexity $T(n)$ of Pan's divide-and-conquer algorithm satisfies the recurrence relation

$$T(n) = 143{,}640\,T(n/70), \quad n = 70^i, \quad i \geq 1, \quad \textbf{init.cond}.\ T(1) = 1.$$

Show that this implies that $T(n)$ is approximately $n^{2.795}$.