Chapter 9

ALGEBRAIC PROBLEMS

9.1 THE GENERAL METHOD

In this chapter we shift our attention away from the problems we've dealt with previously to concentrate on methods for dealing with numbers and polynomials. Though computers have the ability already built-in to manipulate integers and reals, they are not directly equipped to manipulate symbolic mathematical expressions such as polynomials. One must determine a way to represent them and then write procedures that perform the desired operations. A system that allows for the manipulation of mathematical expressions (usually including arbitrary precision integers, polynomials, and rational functions) is called a *mathematical symbol manipulation system*. These systems have been fruitfully used to solve a variety of scientific problems for many years. The techniques we study here have often led to efficient ways to implement the operations offered by these systems.

The first design technique we present is called *algebraic transformation*. Assume we have an input I that is a member of set S_1 and a function $f(I)$ that describes what must be computed. Usually the output $f(I)$ is also a member of S_1. Though a method may exist for computing $f(I)$ using operations on elements in S_1, this method may be inefficient. The algebraic transformation technique suggests that we alter the input into another form to produce a member of set S_2. The set S_2 contains exactly the same elements as S_1 except it assumes a different representation for them. Why would we transform the input into another form? Because it may be easier to compute the function f for elements of S_2 than for elements of S_1. Once the answer in S_2 is computed, an *inverse transformation* is performed to yield the result in set S_1.

Example 9.1 Let S_1 be the set of integers represented using decimal notation, and S_2 the set of integers using binary notation. Given two integers from set S_1, plus any arithmetic operations to carry out on these numbers,

today's computers can transform the numbers into elements of set S_2, perform the operations, and transform the result back into decimal form. The algorithms for transforming the numbers are familiar to most students of computer science. To go from elements of set S_1 to set S_2, repeated division by 2 is used, and from set S_2 to set S_1, repeated multiplication is used. The value of binary representation is the simplification that results in the internal circuitry of a computer. □

Example 9.2 Let S_1 be the set of n-degree polynomials ($n \geq 0$) with integer coefficients represented by a list of their coefficients; e.g.,

$$A(x) = a_n x^n + \cdots + a_1 x + a_0$$

The set S_2 consists of exactly the same set of polynomials but is represented by their values at $2n+1$ points; that is, the $2n+1$ pairs $(x_i,\ A(x_i))$, $1 \leq i \leq 2n+1$, would represent the polynomial A. (At this stage we won't worry about what the values of x_i are, but for now you can consider them consecutive integers.) The function f to be computed is the one that determines the product of two polynomials $A(x)$ and $B(x)$, assuming the set S_1 representation to start with. Rather than forming the product directly using the conventional method (which requires $O(n^2)$ operations, where n is the degree of A and B and any possible growth in the size of the coefficients is ignored), we could transform the two polynomials into elements of the set S_2. We do this by *evaluating* $A(x)$ and $B(x)$ at $2n+1$ points. The product can now be computed simply, by multiplying the corresponding points together. The representation of $A(x)B(x)$ in set S_2 is given by the tuples $(x_i,\ A(x_i)B(x_i))$, $1 \leq i \leq 2n+1$, and requires only $O(n)$ operations to compute. We can determine the product of $A(x)B(x)$ in coefficient form by finding the polynomial that *interpolates* (or satisfies) these $2n+1$ points. It is easy to show that there is a unique polynomial of degree $\leq 2n$ that goes through $2n+1$ points.

Figure 9.1 describes these transformations in a graphical form indicating the two paths one can take to reach the coefficient product domain, either directly by conventional multiplication or indirectly by algebraic transformation. The transformation in one direction is effected by evaluation whereas the inverse transformation is accomplished by interpolation. The value of the scheme rests entirely on whether these transformations can be carried out efficiently.

For instance, if $A(x) = 3x^2 + 4x + 1$ and $B(x) = x^2 + 2x + 5$, these can be represented by the pairs $(0,1), (1,8), (2,21), (3,40)$, and $(4,65)$ and $(0,5), (1,8),\ (2,13), (3,20)$, and $(4,29)$, respectively. Then $A(x)B(x)$ in S_2 takes the form $(0,5),\ (1,64), (2,273), (3,800)$, and $(4,1885)$. □

The world of algebraic algorithms is so broad that we only attempt to cover a few of the interesting topics. In Section 9.2 we discuss the question

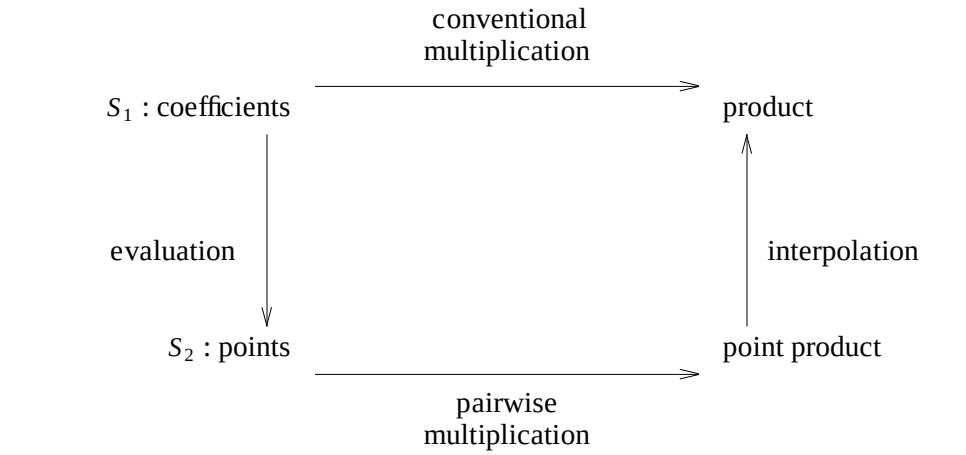

Figure 9.1 Transformation technique for polynomial products

of polynomial evaluation at one or more points and the inverse operation of polynomial interpolation at n points. Then in Section 9.3 we discuss the same problems as in Section 9.2 but this time assuming the n points are nth roots of unity. This is shown to be equivalent to computing the Fourier transform. We also show how the divide-and-conquer strategy leads to the *fast* Fourier transform algorithm. In Section 9.4 we shift our attention to integer problems, in this case the processes of modular arithmetic. Modular arithmetic can be viewed as a transformation scheme that is useful for speeding up large precision integer arithmetic operations. Moreover we see that transformation into and out of modular form is a special case of evaluation and interpolation. Thus there is an algebraic unity to Sections 9.2, 9.3, and 9.4. Finally, in Section 9.5 we present asymptotically efficient algorithms for n-point evaluation and interpolation.

EXERCISES

1. Devise an algorithm that accepts a number in decimal and produces the equivalent number in binary.

2. Devise an algorithm that performs the inverse transformation of Exercise 1.

3. Show the tuples that would result by representing the polynomials

$5x^2+3x+10$ and $7x+4$ at the values $x = 0, 1, 2, 3, 4, 5,$ and 6. What set of tuples is sufficient to represent the product of these two polynomials?

4. If $A(x) = a_n x^n + \cdots + a_1 x + a_0$, then the derivative of $A(x)$, $A'(x) = na_n x^{n-1} + \cdots + a_1$. Devise an algorithm that produces the value of a polynomial and its derivative at a point $x = v$. Determine the number of required arithmetic operations.

9.2 EVALUATION AND INTERPOLATION

In this section we examine the operations on polynomials of evaluation and interpolation. As we search for efficient algorithms, we see examples of another design strategy called *algebraic simplification*. When applied to algebraic problems, algebraic simplification refers to the process of reexpressing computational formulas so that the required number of operations to compute these formulas is minimized. One issue we ignore here is the numerical stability of the resulting algorithms. Though this is often an important consideration, it is too far from our purposes.

A *univariate polynomial* is generally written as

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where x is an indeterminate and the a_i may be integers, floating point numbers, or more generally elements of a commutative ring or a field. If $a_n \neq 0$, then n is called the *degree* of A.

When considering the representation of a polynomial by its coefficients, there are at least two alternatives. The first calls for storing the degree followed by degree + 1 coefficients:

$$(n, a_n, a_{n-1}, \ldots, a_1, a_0)$$

This is termed the *dense* representation because it explicitly stores all coefficients whether or not they are zero. We observe that for a polynomial such as $x^{1000} + 1$ the dense representation is wasteful since it requires 1002 locations although there are only two nonzero terms.

The second representation calls for storing only each *nonzero* coefficient and its corresponding exponent; for example, if all the a_i are nonzero, then the polynomial is stored as

$$(n, a_n, n-1, a_{n-1}, \ldots, 1, a_1,\ 0, a_0).$$

This is termed the *sparse* representation because the storage depends directly on the number of nonzero terms and not on the degree. For a polynomial of degree n, all of whose coefficients are nonzero, this second representation requires roughly twice the storage of the first. However, that is the

```
1   Algorithm StraightEval(A, n, v)
2   {
3       r := 1; s := a_0;
4       for i := 1 to n do
5       {
6           r := r * v;
7           s := s + a_i * r;
8       }
9       return s;
10  }
```

Algorithm 9.1 Straightforward evaluation

worst case. For high-degree polynomials with few nonzero terms, the second representation is many times better than the first.

Secondarily we note that the terms of a polynomial will often be linked together rather than sequentially stored. However, we will avoid this complication in the following algorithms and assume that we can access the ith coefficient by writing a_i.

Suppose we are given the polynomial $A(x) = a_n x^n + \cdots + a_0$ and we wish to evaluate it at a point v, that is, compute $A(v)$. The straightforward or right-to-left method adds $a_1 v$ to a_0 and $a_2 v^2$ to this sum and continues as described in Algorithm 9.1. The analysis of this algorithm is quite simple: $2n$ multiplications, n additions, and $2n+2$ assignments are made (excluding the **for** loop).

An improvement to this procedure was devised by Isaac Newton in 1711. The same improvement was used by W. G. Horner in 1819 to evaluate the coefficients of $A(x + c)$. The method came to be known as Horner's rule. They rewrote the polynomial as

$$A(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

This is our first and perhaps most famous example of algebraic simplification. The function for evaluation that is based on this formula is given in Algorithm 9.2.

Horner's rule requires n multiplications, n additions, and $n + 1$ assignments (excluding the **for** loop). Thus we see that it is an improvement over the straightforward method by a factor of 2. In fact in Chapter 10 we see that Horner's rule yields the optimal way to evaluate an nth-degree polynomial.

```
1   Algorithm Horner(A, n, v)
2   {
3       s := a_n;
4       for i := n − 1 to 0  step −1 do
5       {
6           s := s ∗ v + a_i;
7       }
8       return s;
9   }
```

Algorithm 9.2 Horner's rule

Now suppose we consider the sparse representation of a polynomial, $A(x) = a_m x^{e_m} + \cdots + a_1 x^{e_1}$, where the $a_i \neq 0$ and $e_m > e_{m-1} > \cdots > e_1 \geq 0$. The straightforward algorithm (Algorithm 9.1), when generalized to this sparse case, is given in Algorithm 9.3.

```
1   Algorithm SStraightEval(A, m, v)
2   // Sparse straightforward evaluation.
3   // m is number of nonzero terms.
4   {
5       s := 0;
6       for i := 1 to m do
7       {
8           s := s + a_i ∗ Power(v, e_i);
9       }
10      return s;
11  }
```

Algorithm 9.3 Sparse evaluation

Power(v, e) returns v^e. Assuming that v^e is computed by repeated multiplication with v, this operation requires $e - 1$ multiplications and Algorithm 9.3 requires $e_m + e_{m-1} + \cdots + e_1$ multiplications, m additions, and $m + 1$ assignments. This is horribly inefficient and can easily be improved by an algorithm based on computing

```
1   Algorithm NStraightEval(A, m, v)
2   {
3       s := e_0 := 0; t := 1;
4       for i := 1 to m do
5       {
6           r := Power(v, e_i − e_{i−1});
7           t := t ∗ r;
8           s := s + a_i ∗ t;
9       }
10      return s;
11  }
```

Algorithm 9.4 Evaluating a polynomial represented in coefficient-exponent form

```
1   Algorithm SHorner(A, m, v)
2   {
3       s := e_0 := 0;
4       for i := m to 1  step −1 do
5       {
6           s := (s + a_i) ∗ Power(v, e_i − e_{i−1});
7       }
8       return s;
9   }
```

Algorithm 9.5 Horner's rule for a sparse representation

$$v^{e1}, v^{e2-e1}v^{e1}, v^{e3-e2}v^{e2}, \ldots$$

Algorithm 9.4 requires $e_m + m$ multiplications, $3m + 3$ assignments, m additions, and m subtractions.

A more clever scheme is to generalize Horner's strategy in the revised formula

$$A(x) = (\cdots((a_m x^{e_m - e_{m-1}} + a_{m-1})x^{e_{m-1}-e_{m-2}} + \cdots + a_2)x^{e_2-e_1} + a_1)x^{e_1}$$

The function of Algorithm 9.5 is based on this formula. The number of multiplications required is

$$(e_m - e_{m-1} - 1) + \cdots + (e_1 - e_0 - 1) + m = e_m$$

which is the degree of A. In addition there are m additions, m subtractions, and $m + 2$ assignments. Thus we see that Horner's rule is easily adapted to either the sparse or the dense polynomial model and in both cases the number of operations is bounded and linear in the degree. With a little more work one can find an even better method, assuming a sparse representation, which requires only $m+\log_2 e_m$ multiplications. (See the exercises for a hint.)

Given n points (x_i, y_i), the *interpolation* problem is to find the coefficients of the unique polynomial $A(x)$ of degree $\leq n - 1$ that goes through these n points. Mathematically the answer to this problem was given by Lagrange:

$$A(x) = \sum_{1 \leq i \leq n} \left(\prod_{\substack{i \neq j \\ 1 \leq j \leq n}} \frac{(x - x_j)}{(x_i - x_j)} \right) y_i \tag{9.1}$$

To verify that $A(x)$ does satisfy the n points, we observe that

$$A(x_i) = \left(\prod_{\substack{i \neq j \\ 1 \leq j \leq n}} \frac{(x_i - x_j)}{(x_i - x_j)} \right) y_i = y_i \tag{9.2}$$

since every other term becomes zero. The numerator of each term is a product of $n - 1$ factors and hence the degree of A is $\leq n - 1$.

Example 9.3 Consider the input $(0, 1), (1, 10)$, and $(2, 21)$. Using Equation 9.1, we get

$$\begin{aligned} A(x) &= \tfrac{(x-1)}{(0-1)}\tfrac{(x-2)}{(0-2)}5 + \tfrac{(x-0)}{(1-0)}\tfrac{(x-2)}{(1-2)}10 + \tfrac{(x-0)}{(2-0)}\tfrac{(x-1)}{(2-1)}21 \\ &= \tfrac{5}{2}(x^2 - 3x + 2) - 10(x^2 - 2x) + \tfrac{21}{2}(x^2 - x) \\ &= 3x^2 + 2x + 5 \end{aligned}$$

We can verify that $A(0) = 5$, $A(1) = 10$, and $A(2) = 21$. □

We now give an algorithm (Algorithm 9.6) that produces the coefficients of $A(x)$ using Equation 9.1. We need to perform some addition and multiplication of polynomials. So we assume that the operators $+, *, /$, and $=$ have been overloaded to take polynomials as operands.

```
1   Algorithm Lagrange(X, Y, n, A)
2   // X and Y are one-dimensional arrays containing
3   // n points (x_i, y_i), 1 ≤ i ≤ n. A is a
4   // polynomial that interpolates these points.
5   {
6       // poly is a polynomial.
7       A := 0;
8       for i := 1 to n do
9       {
10          poly := 1; denom := 1;
11          for j := 1 to n do
12              if (i ≠ j) then
13              {
14                  poly := poly * (x - X[j]);
15                      // x - X[j] is a degree one polynomial in x.
16                  denom := denom * (X[i] - X[j]);
17              }
18          A := A + (poly * Y[i]/denom);
19      }
20  }
```

Algorithm 9.6 Lagrange interpolation

An analysis of the computing time of Lagrange is instructive. The **if** statement is executed n^2 times. The time to compute each new value of *denom* is one subtraction and one multiplication, but the execution of $*$ (as applied to polynomials) requires more than constant time per call. Since the degree of $x - X[j]$ is one, the time for one execution of $*$ is proportional to the degree of *poly*, which is at most $j - 1$ on the jth iteration.

Therefore the total cost of the polynomial multiplication step is

$$\begin{aligned}\sum_{1\le i\le n}\sum_{1\le j\le n}(j-1) &= \sum_{1\le i\le n}\left(\frac{n(n+1)}{2}-n\right)\\ &= n^2(n+1)/2 - n^2\end{aligned}$$

$$\text{Thus}\quad \sum_{1\le i\le n}\sum_{1\le j\le n}(j-1) = O(n^3) \tag{9.3}$$

This result is discouraging because it is so high. Perhaps we should search for a better method. Suppose we already have an interpolating polynomial $A(x)$ such that $A(x_i) = y_i$ for $1 \leq i \leq n$ and we want to add just one more point $(x_{n+1},\ y_{n+1})$. How would we compute this new interpolating polynomial given the fact that $A(x)$ was already available? If we could solve this problem efficiently, then we could apply our solution n times to get an n-point interpolating polynomial.

Let $G_{j-1}(x)$ interpolate $j-1$ points $(x_k,\ y_k)$, $1 \leq k < j$, so that $G_{j-1}(x_k) = y_k$. Also let $D_{j-1}(x) = (x - x_1) \cdots (x - x_{j-1})$. Then we can compute $G_j(x)$ by the formula

$$G_j(x) = [y_j - G_{j-1}(x_j)]\frac{D_{j-1}(x)}{D_{j-1}(x_j)} + G_{j-1}(x) \tag{9.4}$$

We observe that

$$G_j(x_k) = [y_j - G_{j-1}(x_j)]\frac{D_{j-1}(x_k)}{D_{j-1}(x_j)} + G_{j-1}(x_k)$$

but $D_{j-1}(x_k) = 0$ for $1 \leq k < j$. So

$$G_j(x_k) = G_{j-1}(x_k) = y_k$$

Also we observe that

$$\begin{aligned} G_j(x_j) &= [y_j - G_{j-1}(x_j)]\frac{D_{j-1}(x_j)}{D_{j-1}(x_j)} + G_{j-1}(x_j) \\ &= y_j - G_{j-1}(x_j) + G_{j-1}(x_j) \\ &= y_j \end{aligned}$$

Example 9.4 Consider again the input $(0, 5), (1, 10)$, and $(2, 21)$. Here $G_1(x) = 5$ and $D_1(x) = (x - x_1) = x$.

$$G_2(x) = [y_2 - G_1(x_2)]\frac{D_1(x)}{D_1(x_2)} + G_1(x) = (10 - 5)\frac{x}{1} + 5 = 5x + 5$$

Also, $D_2(x) = (x - x_1)(x - x_2) = (x - 0)(x - 1) = x^2 - x$. Finally,

$$\begin{aligned} G_3(x) &= [y_3 - G_2(x_3)]\frac{D_2(x)}{D_2(x_3)} + G_2(x) \\ &= [21 - 15]\frac{x^2 - x}{2} + (5x + 5) = 3x^2 + 2x + 5 \quad \square \end{aligned}$$

Having verified that this formula is correct, we present an algorithm (Algorithm 9.7) for computing the interpolating polynomial that is based on Equation 9.4. Notice that from the equation, two applications of Horner's rule are required, one for evaluating $G_{j-1}(x)$ at x_j and the other for evaluating $D_{j-1}(x)$ at x_j.

```
1   Algorithm Interpolate(X, Y, n, G)
2   // Assume n ≥ 2. X[1 : n] and Y[1 : n] are the
3   // n pairs of points. The unique interpolating
4   // polynomial of degree < n is returned in G.
5   {
6       // D is a polynomial.
7       G := Y[1]; // G begins as a constant.
8       D := x − X[1]; // D is a linear polynomial.
9       for j := 2 to n do
10      {
11          denom := Horner(D, j − 1, X[j]); // Evaluate D at X[j].
12          num := Horner(G, j − 2, X[j]); // Evaluate G at X[j].
13              G := G + (D ∗ (Y[j] − num)/denom);
14          D := D ∗ (x − X[j]);
15      }
16  }
```

Algorithm 9.7 Newtonian interpolation

On the jth iteration D has degree $j-1$ and G has degree $j-2$. Therefore the invocations of Horner require

$$\sum_{1 \le j \le n-1} (j-1+j-2) = n(n-1) - 3(n-1) = n^2 - 4n + 3 \tag{9.5}$$

multiplications in total. The term $(Y[j] - num)/denom$ in Algorithm 9.7 is a constant. Multiplying this constant by D requires j multiplications and multiplying D by $x - X[j]$ requires j multiplications. The addition with G requires zero multiplications. Thus the remaining steps require

$$\sum_{1 \le j \le n-1} (2j) = n(n-1) \tag{9.6}$$

operations, so the entire algorithm Interpolate requires $O(n^2)$ operations.

In conclusion we observe that for a dense polynomial of degree n, evaluation can be accomplished using $O(n)$ operations or, for a sparse polynomial with m nonzero terms and degree n, evaluation can be done using at most $O(m+n) = O(n)$ operations. Also, given n points, we can produce the interpolating polynomial in $O(n^2)$ time. In Chapter 10 we discuss the question of the optimality of Horner's rule for evaluation. Section 9.5 presents an even faster way to perform the interpolation of n points as well as the evaluation of a polynomial at n points.

EXERCISES

1. Devise a divide-and-conquer algorithm to evaluate a polynomial at a point. Analyze carefully the time for your algorithm. How does it compare to Horner's rule?

2. Present algorithms for overloading the operators + and $*$ in the case of polynomials.

3. Assume that polynomials such as $A(x) = a_n x^n + \cdots + a_0$ are represented using the dense form. Present an algorithm that overloads the operators + and = to perform the instruction $r = s + t$**;**, where r, s, and t are arbitrary polynomials.

4. Using the same assumptions as for Exercise 3, write an algorithm to perform $r = s * t$**;**.

5. Let $A(x) = a_n x^n + \cdots + a_0$, $p = n/2$ and $q = \lceil n/2 \rceil$. Then a variation of Horner's rule states that

$$\begin{aligned} A(x) &= (\cdots(a_{2p}x^2 + a_{2p-2})x^2 + \cdots)x^2 + a_0 \\ &\quad +((\cdots(a_{2q-1}x^2 + a_{2q-3})x^2 + \cdots)x^2 + a_1)x \end{aligned}$$

   Show how to use this formula to evaluate $A(x)$ at $x = v$ and $x = -v$.

6. Given the polynomial $A(x)$ in Exercise 5 devise an algorithm that computes the coefficients of polynomial $A(x + c)$ for some constant c.

7. Suppose the polynomial $A(x)$ has real coefficients but we wish to evaluate A at the complex number $x = u + iv$, u and v being real. Develop an algorithm to do this.

8. Suppose the polynomial $A(x) = a_m x^{e_m} + \cdots + a_1 x^{e_1}$, where $a_i \neq 0$ and $e_m > e_{m-1} > \cdots > e_1 \geq 0$, is represented using the sparse form. Write a function $\mathsf{PAdd}(r, s, t)$ that computes the sum of two such polynomials r and s and stores the result in t.

9. Using the same assumptions as in Exercise 8, write a function $\mathsf{PMult}(r, s, t)$ that computes the product of the polynomials r and s and places the result in t. What is the computing time of your function?

10. Determine the polynomial of smallest degree that interpolates the points (0, 1), (1, 2), and (2, 3).

11. Given n points $(x_i,\ y_i)$, $1 \leq i \leq n$, devise an algorithm that computes both the interpolating polynomial $A(x)$ and its derivative at the same time. How efficient is your algorithm?

12. Prove that the polynomial of degree $\leq n$ that interpolates $n+1$ points is unique.

13. The binary method for exponentiation uses the binary expansion of the exponent n to determine when to square the temporary result and when to multiply it by x. Since there are $\lfloor \log n \rfloor + 1$ bits in n, the algorithm requires $O(\log n)$ operations; this algorithm is an order of magnitude faster than iteration. The method appears as Algorithm 1.20. Show how to use the binary method to evaluate a sparse polynomial in time $m + \log e_m$.

14. Suppose you are given the real and imaginary parts of two complex numbers. Show that the real and imaginary parts of their product can be computed using only three multiplications.

15. (a) Show that the polynomials $ax + b$ and $cx + d$ can be multiplied using only three scalar multiplications.

    (b) Employ the above algorithm to devise a divide-and-conquer algorithm to multiply two given nth degree polynomials in time $\Theta(n^{\log_2 3})$.

16. The Fibonacci sequence is defined as $f_0 = 0,\ f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$. Give an $O(\log n)$ algorithm to compute f_n. (*Hint*:

$$\begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_{n-2} \\ f_{n-1} \end{bmatrix}.)$$

9.3 THE FAST FOURIER TRANSFORM

If one is able to devise an algorithm that is an order of magnitude faster than any previous method, that is a worthy accomplishment. When the improvement is for a process that has many applications, then that accomplishment has a significant impact on researchers and practitioners. This is the case

of the fast Fourier transform. No algorithm improvement has had a greater impact in the recent past than this one. The Fourier transform is used by electrical engineers in a variety of ways including speech transmission, coding theory, and image processing. But before this fast algorithm was developed, the use of this transform was considered impractical.

The Fourier transform of a continuous function $a(t)$ is given by

$$A(f) = \int_{-\infty}^{\infty} a(t)e^{2\pi ift}\ dt \tag{9.7}$$

whereas the inverse transform of $A(f)$ is

$$a(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} A(f)e^{-2\pi ift}\ df \tag{9.8}$$

The i in the above two equations stands for the square root of -1. The constant e is the base of the natural logarithm. The variable t is often regarded as time, and f is taken to mean frequency. Then the Fourier transform is interpreted as taking a function of time into a function of frequency.

Corresponding to this continuous Fourier transform is the *discrete* Fourier transform which handles sample points of $a(t)$, namely, $a_0, a_1, \ldots, a_{N-1}$. The discrete Fourier transform is defined by

$$A_j = \sum_{0\le k\le N-1} a_k e^{2\pi ijk/N}, \quad 0 \le j \le N-1 \tag{9.9}$$

and the inverse is

$$a_k = \frac{1}{N}\sum_{0\le j\le N-1} A_j e^{-2\pi ijk/N}, \quad 0 \le k \le N-1 \tag{9.10}$$

In the discrete case a set of N sample points is given and a resulting set of N points is produced. An important fact to observe is the close connection between the discrete Fourier transform and polynomial evaluation. If we imagine the polynomial

$$a(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + \cdots + a_1x + a_0$$

then the Fourier element A_j is the value of $a(x)$ at $x = w^j$, where $w = e^{2\pi i/N}$. Similarly for the inverse Fourier transform if we imagine the polynomial with the Fourier coefficients

$$A(x) = A_{N-1}x^{N-1} + A_{N-2}x^{N-2} + \cdots + A_1x + A_0$$

then each a_k is the value of $A(x)/N$ at $x = (w^{-1})^k$, where $w = e^{2\pi i/N}$. Thus, the *discrete Fourier transform* corresponds exactly to the evaluation of a polynomial at N points: $w^0, w^1, \ldots, w^{N-1}$.

From the preceding section we know that we can evaluate an Nth-degree polynomial at N points using $O(N^2)$ operations. We apply Horner's rule once for each point. The *fast Fourier transform* (abbreviated FFT) is an algorithm for computing these N values using only $O(N \log N)$ operations. This algorithm was popularized by J. M. Cooley and J. W. Tukey in 1965, and the long history of this method was traced by J. M. Cooley, P. A. Lewis and P. D. Welch.

A hint that the Fourier transform can be computed faster than by Horner's rule comes from observing that the evaluation points are not arbitrary but are in fact very special. They are the N powers w^j for $0 \leq j \leq N-1$, where $w = e^{2\pi i/N}$. The point w is a primitive Nth root of unity in the complex plane.

Definition 9.1 An element w in a commutative ring is called a *primitive* Nth *root of unity* if

1. $w \neq 1$
2. $w^N = 1$
3. $\sum_{0 \leq p \leq N-1} w^{jp} = 0,\ 1 \leq j \leq N-1$ □

Example 9.5 Let $N = 4$. Then, $w = e^{\pi i/2} = cos(\pi/2) + i\ sin(\pi/2) = i$. Thus, $w \neq 1$, and $w^4 = i^4 = 1$. Also, $\sum_{0 \leq p \leq 3} w^{jp} = 1 + i^j + i^{2j} + i^{3j} = 0$. □

We now present two simple properties of Nth roots from which we can see how the FFT algorithm can easily be understood.

Theorem 9.1 Let $N = 2n$ and suppose w is a primitive Nth root of unity. Then $-w^j = w^{j+n}$.

Proof: Here $(w^{j+n})^2 = (w^j)^2(w^n)^2 = (w^j)^2(w^{2n}) = (w^j)^2$ since $w^n = 1$. Since the w^j are distinct, we know that $w^j \neq w^{j+n}$, so we can conclude that $w^{j+n} = -w^j$. □

Theorem 9.2 Let $N = 2n$ and w a primitive Nth root of unity. Then w^2 is a primitive nth root of unity.

Proof: Since $w^N = w^{2n} = 1$, $(w^2)^n = 1$; this implies w^2 is an nth root of unity. In addition we observe that $(w^2)^j \neq 1$ for $1 \leq j \leq n-1$ since otherwise we would have $w^k = 1$ for $1 \leq k < 2n = N$ which would contradict the fact

that w is a primitive Nth root of unity. Therefore w^2 is a primitive nth root of unity. □

From this theorem we can conclude that if w^j, $0 < j \leq N-1$, are the primitive Nth roots of unity and $N = 2n$, then w^{2j}, $0 < j \leq n-1$, are primitive nth roots of unity. Using these two theorems, we are ready to show how to derive a divide-and-conquer algorithm for the Fourier transform. The complexity of the algorithm is $O(N \log N)$, an order of magnitude faster than the $O(N^2)$ of the conventional algorithm which uses polynomial evaluation.

Again let $a_{N-1}, \ldots, a_0$ be the coefficients to be transformed and let $a(x) = a_{N-1}x^{N-1} + \cdots + a_1 x + a_0$. We break $a(x)$ into two parts, one of which contains even-numbered exponents and the other odd-numbered exponents.

$$\begin{aligned} a(x) &= (a_{N-1}x^{N-1} + a_{N-3}x^{N-3} + \cdots + a_1 x) \\ &\quad + (a_{N-2}x^{N-2} + \cdots + a_2 x^2 + a_0) \end{aligned}$$

Letting $y = x^2$, we can rewrite $a(x)$ as a sum of two polynomials.

$$\begin{aligned} a(x) &= (a_{N-1}y^{n-1} + a_{N-3}y^{n-2} + \cdots + a_1)x \\ &\quad + (a_{N-2}y^{n-1} + a_{N-4}y^{n-2} + \cdots + a_0) \\ &= c(y)x + b(y) \end{aligned}$$

Recall that the values of the Fourier transform are $a(w^j)$, $0 \leq j \leq N-1$. Therefore the values of $a(x)$ at the points w^j, $0 \leq j \leq n-1$, are now expressible as

$$\begin{aligned} a(w^j) &= c(w^{2j})w^j + b(w^{2j}) \\ a(w^{j+n}) &= -c(w^{2j})w^j + b(w^{2j}) \end{aligned}$$

These two formulas are computationally valuable in that they reveal how to take a problem of size N and transform it into two identical problems of size $n = N/2$. These subproblems are the evaluation of $b(y)$ and $c(y)$, each of degree $n-1$, at the points $(w^2)^j$, $0 \leq j \leq n-1$, and these points are primitive nth roots. This is an example of divide-and-conquer, and we can apply the divide-and-conquer strategy again as long as the number of points remains even. This leads us to always choose N as a power of 2, $N = 2^m$, for then we can continue to carry out the splitting procedure until a trivial problem is reached, namely, evaluating a constant polynomial.

FFT (Algorithm 9.8) combines all these ideas into a recursive version of the fast Fourier transform algorithm. Dense representation for polynomials

```
1   Algorithm FFT(N, a(x), w, A)
2   // N = 2^m, a(x) = a_{N-1}x^{N-1} + ··· + a_0, and w is a
3   // primitive Nth root of unity. A[0 : N − 1] is set to
4   // the values a(w^j), 0 ≤ j ≤ N − 1.
5   {
6       // b and c are polynomials.
7       // B, C, and wp are complex arrays.
8       if N = 1 then A[0] := a_0;
9       else
10      {
11          n := N/2;
12          b(x) := a_{N-2}x^{n-1} + ··· + a_2x + a_0;
13          c(x) := a_{N-1}x^{n-1} + ··· + a_3x + a_1;
14          FFT(n, b(x), w^2, B);
15          FFT(n, c(x), w^2, C);
16          wp[−1] := 1/w;
17          for j := 0 to n − 1 do
18          {
19              wp[j] := w * wp[j − 1];
20              A[j] := B[j] + wp[j] * C[j];
21              A[j + n] := B[j] − wp[j] * C[j];
22          }
23      }
24  }
```

Algorithm 9.8 Recursive fast Fourier transform

is assumed. We overload the operators $+$, $-$, $*$, and $=$ with regard to complex numbers.

Now let us derive the computing time of FFT. Let $T(N)$ be the time for the algorithm applied to N inputs. Then we have

$$T(N) = 2T(N/2) + DN$$

where D is a constant and DN is a bound on the time needed to form $b(x)$, $c(x)$, and A. Since $T(1) = d$, where d is another constant, we can repeatedly simplify this recurrence relation to get

$$T(2^m) = 2T(2^{m-1}) + D2^m$$

$$
\begin{aligned}
&= \\
&\vdots \\
&= Dm2^m + T(1)2^m \\
&= DN\log_2 N + dN \\
&= O(N\log_2 N)
\end{aligned}
$$

Suppose we return briefly to the problem considered at the beginning of this chapter, the multiplication of polynomials. The transformation technique calls for evaluating $A(x)$ and $B(x)$ at $2N+1$ points (where N is the degree of A and B), computing the $2N+1$ products $A(x_i)B(x_i)$, and then finding the product $A(x)B(x)$ in coefficient form by computing the interpolating polynomial that satisfies these points. In Section 9.2 we saw that N-point evaluation and interpolation required $O(N^2)$ operations, so that no asymptotic improvement is gained by using this transformation over the conventional multiplication algorithm. However, in this section we have seen that if the points are chosen to be the $N = 2^m$ distinct powers of a primitive Nth root of unity, then evaluation and interpolation can be done using at most $O(N \log N)$ operations. Therefore by using the fast Fourier transform algorithm, we can multiply two N-degree polynomials in $O(N \log N)$ operations.

The divide-and-conquer strategy plus some simple properties of primitive Nth roots of unity leads to a very nice conceptual framework for understanding the FFT. The above analysis shows that asymptotically it is better than the direct method by an order of magnitude. However the version we have produced uses auxiliary space for b, c, B, and C. We need to study this algorithm more closely to eliminate this overhead.

Example 9.6 Consider the case in which $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. Let us walk through the execution of Algorithm 9.8 on this input. Here $N = 4, n = 2$, and $w = i$. The polynomials b and c are constructed as $b(x) = a_2x + a_0$ and $c(x) = a_3x + a_1$. Function FFT is invoked on $b(x)$ and $c(x)$ to get $B[0] = a_0 + a_2$, $B[1] = a_0 + a_2w^2$, $C[0] = a_1 + a_3$, and $C[1] = a_1 + a_3w^2$.

In the **for** loop, the array $A[\ ]$ is modified. When $j = 0$, $wp[0] = 1$. Thus, $A[0] = B[0] + C[0] = a_0 + a_1 + a_2 + a_3$ and $A[2] = B[0] - C[0] = a_0 + a_2 - a_1 - a_3 = a_0 + a_1w^2 + a_2w^4 + a_3w^6$ (since $w^2 = -1, w^4 = 1$, and $w^6 = -1$). When $j = 1$, $wp[1] = w$. Then $A[1] = B[1] + wC[1]$ $= a_0+a_2w^2+w(a_1+a_3w^2) = a_0+a_1w+a_2w^2+a_3w^3$ and $A[3] = B[1]-wC[1]$ $= a_0+a_2w^2-w(a_1+a_3w^2) = a_0-a_1w+a_2w^2-a_3w^3 = a_0+a_1w^3+a_2w^6+a_3w^9$ (since $w^2 = -1, w^4 = 1$, and $w^6 = -1$). □