BIBLIOGRAPHIC NOTES

Cooley, Lewis, and Welch [1967] trace the origins of the fast Fourier transform to Runge and König [1924]. The technique has been used by Danielson and Lanczos [1942] and Good [1958, 1960]. The fundamental paper by Cooley and Tukey [1965] made clear the nature of the technique. The development of the FFT as a polynomial division problem, the approach used here, is due to Fiduccia [1972]. Because of its importance in computing, a good deal of attention has been paid to the efficient implementation of the algorithm. For example, see Gentleman and Sande [1966] and many articles from Rabiner and Rader [1972]. The integer-multiplication algorithm is from Schönhage and Strassen [1971]. Exercises 7.18 and 7.19 are from Rader [1968].

INTEGER AND POLYNOMIAL ARITHMETIC

CHAPTER 8

A good reason for treating integer and polynomial arithmetic together is that many of the algorithms for manipulating integers and univariate polynomials are essentially identical. This is true not only for operations like multiplication and division but also for more sophisticated operations. For example, finding the residue of an integer modulo a second integer is equivalent to evaluating a polynomial at a point. Representing an integer by its residues is equivalent to representing a polynomial by its values at several points. Reconstructing an integer from residues ("Chinese remaindering") is equivalent to interpolating a polynomial.

In this chapter, we shall show that certain integer and polynomial operations, such as division and squaring, require the same order of time as multiplication. Other operations, such as the residue operations mentioned above, or the calculation of greatest common divisors, are shown to require at most a factor of $\log n$ more time than multiplication, where n is the length of the binary integer or degree of the polynomial. Our strategy will be to alternate results for integers with the corresponding results for polynomials, usually proving the results for only one and leaving the other as an exercise. As in the other chapters, our emphasis is on algorithms that are asymptotically the most efficient known.

At the end of the chapter we briefly discuss some distinctions between a model for polynomials that assumes most coefficients are nonzero (the dense model) and one that assumes most coefficients are zero (the sparse model). The sparse model is particularly useful when dealing with polynomials involving many variables (a case we do not discuss).

8.1 THE SIMILARITY BETWEEN INTEGERS AND POLYNOMIALS

The most obvious similarity between a nonnegative integer and a polynomial in one variable is that each can be represented as a finite power series $\sum_{i=0}^{n-1} a_i x^i$. In the integer case, the a_i's can be chosen from the set $\{0, 1\}$ with $x = 2$. In the polynomial case, the a_i's can be chosen from some coefficient set† with x the indeterminate.

There is a natural "size" measure, which is essentially the length of the power series representing the integer or polynomial. In the binary integer case the size is the number of bits needed to represent the integer; in the polynomial case, the size is the number of coefficients. Thus we make the following definition.

† In what follows, you may regard the coefficient set as the field (see Section 12.1) of real numbers, although the results will apply to any field of coefficients, with computational complexity measured in terms of the number of operations in the coefficient field. Thus we do not consider the size of coefficients in our model.

Definition. If i is a nonnegative integer, then $\text{SIZE}(i) = \lfloor \log i \rfloor + 1$. If $p(x)$ is a polynomial, then $\text{SIZE}(p) = \text{DEG}(p) + 1$, where $\text{DEG}(p)$ is the degree of p, i.e., the highest power of x with a nonzero coefficient.

With integers and polynomials we can perform approximate division. If a and b are two integers, with $b \neq 0$, then there is a unique pair of integers q and r such that

1. $a = bq + r$, and
2. $r < b$

where q and r are the quotient and remainder when a is divided by b.

Similarly, if a and b are polynomials, with b not equal to a constant, then unique polynomials q and r can be found to satisfy

1. $a = bq + r$, and
2. $\text{DEG}(r) < \text{DEG}(b)$.

Another similarity between integers and polynomials is that they each have surprisingly fast multiplication algorithms. In the previous chapter we showed that two nth-degree polynomials with real coefficients can be multiplied in $O_A(n \log n)$ time by use of the FFT. The arithmetic operation measure of complexity is reasonable, since in practice we would represent polynomials by their coefficients to fixed precision and implement the various polynomial operations by arithmetic operations on the coefficients.

If we use the Schönhage-Strassen algorithm of Section 7.5, we can multiply two n-bit integers in $O_B(n \log n \text{ loglog } n)$ time. We claim that for integers, bit operations are the only measure of interest. There are essentially two situations in which we would not consider multiplication of integers to be primitive. The first is in designing multiplication hardware. The number of bit operations reflects the number of elements needed in a multiplication circuit. The second application is in designing fixed-point arbitrary precision algorithms for fixed-word-length computers. There, the number of bit operations is related to the number of machine instructions needed to do n-precision multiplication.

Thus the results for polynomial and integer arithmetic will appear quite similar when the two different measures of complexity (arithmetic and bit) are used. Moreover, the two measures are analogous in the sense that bit operations are coefficient operations for the power series representing integers, just as arithmetic operations are coefficient operations for polynomials.

8.2 INTEGER MULTIPLICATION AND DIVISION

We shall show that the time, in bit operations, to do integer multiplication is, to within a constant factor, the same as the time to do integer division and is

similarly related to the operations of squaring and taking reciprocals. In this section we shall use four symbols with the following meanings:

Symbol	Meaning
$M(n)$	Time to multiply two integers of size n
$D(n)$	Time to divide an integer of size at most $2n$ by an integer of size n
$S(n)$	Time to square an integer of size n
$R(n)$	Time to compute the reciprocal of an integer of size n

In each case the time is measured in bit operations. We shall also make the reasonable assumption that $M(n)$ satisfies the condition

$$a^2M(n) \geq M(an) \geq aM(n)$$

for $a \geq 1$. We assume the other three functions also share this property.

We first show that the reciprocal of an n-bit integer i can be calculated in essentially the time to multiply two n-bit numbers. Since $1/i$ is not an integer for $i > 1$, what we really mean by the "reciprocal" of $1/i$ is the n-significant-bit approximation to the fraction $1/i$. Since scaling (shifts of the binary point) is assumed to cost nothing, we may equivalently say that the "reciprocal" of i is the quotient of 2^{2n-1} divided by i. For the remainder of this section we use the term reciprocal with this meaning.

First let us consider finding a sequence of approximations to the number $A = 1/P$, where P is an integer. Let A_i be the ith approximation to $1/P$. Then the exact value of A can be expressed as

$$A = A_i + (1/P)(1 - A_iP). \tag{8.1}$$

If we approximate the value of $1/P$ by A_i in (8.1) we obtain

$$A_{i+1} = A_i + A_i(1 - A_iP) = 2A_i - A_i^2P, \tag{8.2}$$

which can be used as an iteration formula for finding the $(i + 1)$st approximation of A in terms of the ith approximation. Note that if $A_iP = 1 - S$, then

$$A_{i+1}P = 2A_iP - A_i^2P^2 = 2(1 - S) - (1 - S)^2 = 1 - S^2.$$

This shows that the iteration formula (8.2) converges quadratically. If $S \leq \frac{1}{2}$, then the number of correct bits is doubled each iteration.

Since P presumably has many bits, it is unnecessary to use all bits of P for the earlier approximations, because only the leading bits of P affect those bits of A_{i+1} which are correct. Moreover, if the first k bits of A_i to the right of the decimal point are correct, then we can obtain A_{i+1} to $2k$ bits using the formula (8.2). That is, we compute $A_{i+1} = 2A_i - A_i^2P$, with $2A_i$ truncated to k places to the right of the decimal point and A_i^2P computed by approximating P to $2k$ places and then truncating the product to $2k$ places to the right of the decimal point. This last approximation may of course affect the convergence, since we previously assumed that P was exact.

We shall use these ideas in the next algorithm. There we actually compute the quotient $\lfloor 2^{2n-1}/P \rfloor$, where $P = p_1p_2 \cdots p_n$ is an n-bit integer with $p_1 = 1$. The method is essentially that of Eq. (8.2), with scaling introduced to allow us to deal exclusively with integers. Thus no maintenance of the decimal point is necessary.

Algorithm 8.1. Integer reciprocals.

Input. An n-bit integer $P = [p_1p_2 \cdots p_n]$, with $p_1 = 1$. For convenience, we assume n is a power of 2 and $[x]$ is the integer denoted by the bit string x (e.g., $[110] = 6$).

Output. The integer $A = [a_0a_1 \cdots a_n]$ such that $A = \lfloor 2^{2n-1}/P \rfloor$.

Method. We call RECIPROCAL($[p_1p_2 \cdots p_n]$), where RECIPROCAL is the recursive procedure in Fig. 8.1. It computes an approximation to $\lfloor 2^{2k-1}/[p_1p_2 \cdots p_k] \rfloor$ for any k which is a power of 2. Observe that the result is normally a k-bit integer except when P is a power of 2, in which case the result is a $(k + 1)$-bit integer.

Given k bits of the reciprocal of $[p_1p_2 \cdots p_k]$, lines 2–4 compute $2k - 3$ bits of the reciprocal of $[p_1p_2 \cdots p_{2k}]$. Lines 5–7 correct the last three bits. In practice, one would skip lines 5–7 and obtain the desired accuracy by an extra application of Eq. (8.2) at the end. We have chosen to include the loop of lines 5–7 to simplify both the understanding of the algorithm and the proof that the algorithm does indeed work. □

Example 8.1. Let us compute $2^{15}/153$. Here,

$$n = 8 \quad \text{and} \quad 153 = [p_1p_2 \cdots p_8] = [10011001].$$

We call

RECIPROCAL([10011001]),

which in turn calls RECIPROCAL recursively with arguments [1001], [10], and [1]. At line 1, we find RECIPROCAL([1]) = [10] and return to RECIPROCAL([10]) at line 2, where we set $[c_0c_1] \leftarrow [10]$. Then, at line 3, we compute $[d_1 \cdots d_4] \leftarrow [10] * 2^3 - [10]^2 * [10] = [1000]$. Next, we set $[a_0a_1a_2]$ to [100] at line 4. No changes occur in the loop of lines 5–7. We return to RECIPROCAL([1001]) with [100] as an approximation to $2^3/[10]$.

Returning to line 2 with $k = 4$, we have $[c_0c_1c_2] = [100]$. Then $[d_1 \cdots d_8] = [01110000]$ and $[a_0 \cdots a_4] = [01110]$. Again, there are no changes in the loop of lines 5–7. We return to RECIPROCAL([10011001])

```
      procedure RECIPROCAL([p_1 p_2 ··· p_k]):
1.    if k = 1 then return [10]
      else
          begin
2.            [c_0 c_1 ··· c_{k/2}] ← RECIPROCAL([p_1 p_2 ··· p_{k/2}]);
3.            [d_1 d_2 ··· d_{2k}] ← [c_0 c_1 ··· c_{k/2}] * 2^{3k/2} −
                [c_0 c_1 ··· c_{k/2}]^2 * [p_1 p_2 ··· p_k];
              comment Although the right-hand side of line 3 appears to
                produce a (2k + 1)-bit number, the leading [(2k + 1)st]
                bit is always zero;
4.            [a_0 a_1 ··· a_k] ← [d_1 d_2 ··· d_{k+1}];
              comment [a_0 a_1 ··· a_k] is a good approximation to
                2^{2k−1}/[p_1 p_2 ··· p_k]. The following loop improves the
                approximation by adding to the last three places if nec-
                essary;
5.            for i ← 2 step −1 until 0 do
6.              if ([a_0 a_1 ··· a_k] + 2^i) * [p_1 p_2 ··· p_k] ≤ 2^{2k−1} then
7.                  [a_0 a_1 ··· a_k] ← [a_0 a_1 ··· a_k] + 2^i;
8.            return [a_0 a_1 ··· a_k]
          end
```

Fig. 8.1. Procedure to compute integer reciprocals.

at line 2 with $[c_0 \cdots c_4] = [01110]$. Then

$$[d_1 \cdots d_{16}] = [0110101011011100].$$

Thus at line 4, $[a_0 \cdots a_8] = [011010101]$. At line 6, we find that $[011010101] * [10011001]$, which is $213 * 153$ in decimal, equals 32589, while $2^{15} = 32768$. Thus in the loop of lines 5–7, 1 is added to 213, yielding answer 214, or [011010110] in binary. □

Theorem 8.1. Algorithm 8.1 finds $[a_0 a_1 \cdots a_k]$ such that

$$[a_0 a_1 \cdots a_k] * [p_1 p_2 \cdots p_k] = 2^{2k-1} - S$$

and $0 \le S < [p_1 p_2 \cdots p_k]$.

Proof. The proof is by induction on k. The basis, $k = 1$, is trivial by line 1. For the inductive step, let $C = [c_0 c_1 \cdots c_{k/2}]$, $P_1 = [p_1 p_2 \cdots p_{k/2}]$, and $P_2 = [p_{k/2+1} p_{k/2+2} \cdots p_k]$. Then $P = [p_1 p_2 \cdots p_k] = P_1 2^{k/2} + P_2$. By the induction hypothesis

$$CP_1 = 2^{k-1} - S,$$

where $0 \le S < P_1$. By line 3, $D = [d_1 d_2 \cdots d_{2k}]$ is given by

$$D = C2^{3k/2} - C^2(P_1 2^{k/2} + P_2). \tag{8.3}$$

Since $p_1 = 1$, $P_1 \ge 2^{k/2-1}$ and thus $C \le 2^{k/2}$. It follows that $D < 2^{2k}$ and hence the $2k$ bits used to represent D are sufficient.

Consider the product $PD = (P_1 2^{k/2} + P_2)D$, which by (8.3) is:

$$PD = CP_1 2^{2k} + CP_2 2^{3k/2} - (CP_1 2^{k/2} + CP_2)^2. \tag{8.4}$$

By substituting $2^{k-1} - S$ for CP_1 in (8.4) and performing some algebraic simplification, we get:

$$PD = 2^{3k-2} - (S2^{k/2} - CP_2)^2. \tag{8.5}$$

Dividing (8.5) by 2^{k-1}, we have

$$2^{2k-1} = \frac{PD}{2^{k-1}} + T, \tag{8.6}$$

where $T = (S2^{k/2} - CP_2)^2 2^{-(k-1)}$. By the induction hypothesis and the fact that $P_1 < 2^{k/2}$, we have $S < 2^{k/2}$. Since $C \le 2^{k/2}$ and $P_2 < 2^{k/2}$, we have $0 \le T < 2^{k+1}$.

At line 4, $A = [a_0 a_1 \cdots a_k] = \lfloor D/2^{k-1} \rfloor$. Now

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > P\left(\frac{D}{2^{k-1}} - 1\right),$$

so from (8.6) we have

$$2^{2k-1} \ge \frac{PD}{2^{k-1}} \ge P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > \frac{PD}{2^{k-1}} - P = 2^{2k-1} - T - P > 2^{2k-1} - 2^{k+1} - 2^k.$$

Thus

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor = 2^{2k-1} - S',$$

where $0 \le S' < 2^{k+1} + 2^k$. Since $P \ge 2^{k-1}$, it follows that by adding at most 6 to $\lfloor D/2^{k-1} \rfloor$ we obtain the number which satisfies the inductive hypothesis for k. Since this job is done by lines 5–7, the induction step follows. □

Theorem 8.2. There is a constant c such that $R(n) \le cM(n)$.

Proof. It suffices to show that Algorithm 8.1 works in time $O_B(M(n))$. Line 2 requires $R(k/2)$ bit operations. Line 3 consists of a squaring and multiplication, requiring $M(k/2 + 1)$ and $M(k + 2)$ time, respectively, plus a subtraction requiring $O_B(k)$ time. Note that multiplication by powers of 2 does not require any bit operations; the bits of the multiplicand are simply regarded as occupying new positions, i.e., as if they had been shifted. By our assumption on M, $M(k/2 + 1) \le \frac{1}{2}M(k + 2)$. Furthermore, $M(k + 2) - M(k)$ is $O_B(k)$ (see Section 2.6, for example) and thus line 3 is bounded by $\frac{3}{2}M(k) + c'k$ for some constant c'. Line 4 is clearly $O_B(k)$.

It appears that the loop of lines 5–7 requires three multiplications, but the calculation can be done by one multiplication, $[a_0 a_1 \cdots a_k] * [p_1 p_2 \cdots p_k]$,

and some additions and subtractions of at most $2k$-bit integers. Thus lines 5–7 are bounded by $M(k) + c''(k)$ for some constant c''. Putting all costs together, we have

$$R(k) \leq R\left(\frac{k}{2}\right) + \tfrac{5}{2}M(k) + c_1 k \tag{8.7}$$

for some constant c_1.

We claim that there is a constant c such that $R(k) \leq cM(k)$. We can choose c so that $c \geq R(1)/M(1)$ and $c \geq 5 + 2c_1$. We verify the claim by induction on k. The basis, $k = 1$, is immediate. The inductive step follows from (8.7), since

$$R(k) \leq cM\left(\frac{k}{2}\right) + \tfrac{5}{2}M(k) + c_1 k. \tag{8.8}$$

As $M(k/2) \leq \frac{1}{2}M(k)$ follows from our assumption about M, and $k \leq M(k)$ is obvious, we may rewrite (8.8) as

$$R(k) \leq \left(\frac{c}{2} + \frac{5}{2} + c_1\right)M(k). \tag{8.9}$$

Since $c \geq 5 + 2c_1$, (8.9) implies $R(k) \leq cM(k)$. □

It should be evident that Algorithm 8.2 can be used to compute $1/P$ to n significant bits, if P has that number of bits, no matter where the binary point is. For example, if $\frac{1}{2} < P < 1$, and P has n bits, then by scaling in the obvious way, we can produce $1/P$ as 1, followed by $n - 1$ bits in the fractional part.

We next show that $S(n)$, the time needed to square an integer of size n, is of no greater magnitude than $R(n)$, the time to take the reciprocal of an integer of size n. The technique involves the identity

$$P^2 = \frac{1}{\dfrac{1}{P} - \dfrac{1}{P+1}} - P. \tag{8.10}$$

The next algorithm uses (8.10) with proper scaling.

Algorithm 8.2. Squaring by reciprocals.

Input. An n-bit integer P, in binary representation.

Output. The binary representation of P^2.

Method

1. Use Algorithm 8.1 to compute $A = \lfloor 2^{4n-1}/P \rfloor$ by appending $2n$ 0's to P and applying RECIPROCAL† to compute $\lfloor 2^{6n-1}/P2^{2n} \rfloor$ and then shifting.

† RECIPROCAL was defined in Algorithm 8.1 only for integers whose length was a power of 2. The generalization to integers whose length is not a power of 2 should be obvious—add extra 0's and change scale when necessary.

2. Similarly, compute $B = \lfloor 2^{4n-1}/(P+1) \rfloor$.
3. Let $C = A - B$. Note that $C = 2^{4n-1}/(P^2 + P) + T$ where $|T| \leq 1$. The T arises from the fact that the truncation in computing A and B may give rise to an error of up to 1. Since $2^{2n-2} < P^2 + P < 2^{2n}$ we have $2^{2n+1} \geq C \geq 2^{2n-1}$.
4. Compute $D = \lfloor 2^{4n-1}/C \rfloor - P$.
5. Let Q be the last four bits of P. Adjust the last four bits of D up or down as little as possible to cause agreement with the last four bits of Q^2. □

Theorem 8.3. Algorithm 8.2 computes P^2.

Proof. Due to the truncation involved in steps 1 and 2, we can be assured only that

$$C = \frac{2^{4n-1}}{P^2 + P} + T, \qquad \text{where} \quad |T| \leq 1.$$

Since $2^{2n-1} \leq C \leq 2^{2n+1}$ and since the error in C is in the range -1 to 1, the error in $2^{4n-1}/C$ due to the error in C is at most

$$\left|\frac{2^{4n-1}}{C} - \frac{2^{4n-1}}{C-1}\right| = \left|\frac{2^{4n-1}}{C^2 - C}\right|.$$

Since $C^2 - C \geq 2^{4n-3}$, the error is at most 4. The truncation at line 4 can increase the error to 5. Thus $|P^2 - D| \leq 5$. Hence computing the last four bits of P^2 at step 5 insures that D is adjusted to be exactly P^2. □

Example 8.2. Let $n = 4$ and $P = [1101]$. Then

$$A = \lfloor 2^{15}/[1101] \rfloor = [100111011000]$$

and

$$B = \lfloor 2^{15}/[1110] \rfloor = [100100100100].$$

Next,

$$C = A - B = [10110100].$$

Then,

$$D = \lfloor 2^{15}/C \rfloor - P = [10110110] - [1101] = [10101001].$$

Thus D is 169, the square of 13, and no correction is necessary at step 5. □

Theorem 8.4. There exists a constant c such that $S(n) \leq cR(n)$.

Proof. Algorithm 8.2 uses three reciprocal calculations on strings of length at most $3n$. Further, there are subtractions at steps 3 and 4 requiring $O_B(n)$ time, and a fixed amount of work at step 5, independent of n. Hence

$$S(n) \leq 3R(3n) + c_1 n \tag{8.11}$$

for some constant c_1. Thus $S(n) \leq 27R(n) + c_1 n$. Since $R(n) \geq n$, choose $c = 27 + c_1$ to prove the theorem. □

Theorem 8.5. $M(n)$, $R(n)$, $D(n)$, and $S(n)$ are all related by constant factors.

Proof. We have already shown $R(n) \leq c_1 M(n)$ and $S(n) \leq c_2 R(n)$ for constants c_1 and c_2. It is easy to see that $M(n) \leq c_3 S(n)$ by noting that

$$AB = \tfrac{1}{2}[(A+B)^2 - A^2 - B^2].$$

Thus M, R, and S are related by constant factors.

When we discuss division of n-bit numbers, we really mean the division of a number with up to $2n$ bits by one of exactly n bits, producing an answer of at most $n+1$ bits. It is trivial that $R(n) \leq D(n)$, so $M(n) \leq c_2 c_3 D(n)$. Moreover, using the identity $A/B = A * (1/B)$, we may show, taking care for scaling, that for some constant c

$$D(n) \leq M(2n) + R(2n) + cn. \tag{8.12}$$

Since $R(2n) \leq c_1 M(2n)$, and $M(2n) \leq 4M(n)$ is easy to show, we may rewrite (8.12) as

$$D(n) \leq 4(1 + c_1)M(n) + cn. \tag{8.13}$$

Since $M(n) \geq n$, we have from (8.13) that $D(n) \leq c_4 M(n)$, where $c_4 = 4 + 4c_1 + c$. Thus all functions have been shown to lie between $dM(n)$ and $eM(n)$ for some positive constants d and e. □

Corollary. Division of a $2n$-bit integer by an n-bit integer can be done in $O_B(n \log n \operatorname{loglog} n)$ time.

Proof. By Theorem 7.8 and Theorem 8.5. □

8.3 POLYNOMIAL MULTIPLICATION AND DIVISION

All the techniques of the previous section carry over to univariate polynomial arithmetic. Let $M(n)$, $D(n)$, $R(n)$, and $S(n)$ in this section stand for the time to multiply, divide, take reciprocals of, and square nth-degree polynomials. We assume, as before, that $a^2 M(n) \geq M(an) \geq aM(n)$ for $a \geq 1$ and similarly for the other functions.

By the "reciprocal" of an nth-degree polynomial $p(x)$, we mean $\lfloor x^{2n}/p(x) \rfloor$.† $D(n)$ is the time to find $\lfloor s(x)/p(x) \rfloor$, where $p(x)$ is of degree n and $s(x)$ is of degree at most $2n$. Note that we can "scale" polynomials by multiplying and dividing by powers of x, just as we scaled integers by powers of 2 in the previous section.

† By analogy with the notation for integers, we use the "floor function" to denote the quotient of polynomials. That is, if $p(x)$ is not a constant, $\lfloor s(x)/p(x) \rfloor$ is the unique $q(x)$ such that $s(x) = p(x)q(x) + r(x)$ and $\mathrm{DEG}(r(x)) < \mathrm{DEG}(p(x))$.

Since the results of this section are so similar to those for integers, we give only one result in detail: the polynomial "reciprocal" algorithm analogous to Algorithm 8.1 for integers. The polynomial algorithms are somewhat easier than the integer ones, due essentially to the fact that carries from place to place in the power series do not occur as they do for integers. Thus the polynomial algorithms require no adjustment of least significant places as was necessary in, for example, lines 5–7 of Algorithm 8.1.

Algorithm 8.3. Polynomial reciprocals.

Input. A polynomial $p(x)$ of degree $n-1$, where n is a power of 2 [i.e., $p(x)$ has 2^t terms for some integer t].

Output. The "reciprocal" $\lfloor x^{2n-2}/p(x) \rfloor$.

Method. In Fig. 8.2 we define a new procedure

$$\text{RECIPROCAL}\left(\sum_{i=0}^{k-1} a_i x^i\right),$$

where k is a power of 2 and $a_{k-1} \neq 0$. The procedure computes

$$\left\lfloor \frac{x^{2k-2}}{\sum_{i=0}^{k-1} a_i x^i} \right\rfloor.$$

Note that if $k = 1$, then the argument is a constant a_0 whose reciprocal is $1/a_0$, another constant. We assume each operation on coefficients can be done in one step, and no call to RECIPROCAL is necessary to compute $1/a_0$.

The algorithm itself is to call RECIPROCAL with argument $p(x)$. □

```
    procedure RECIPROCAL (Σ_{i=0}^{k-1} a_i x^i):
1.  if k = 1 then return 1/a_0
    else
        begin
2.          q(x) ← RECIPROCAL (Σ_{i=k/2}^{k-1} a_i x^{i-k/2});
3.          r(x) ← 2q(x)x^{(3/2)k-2} − (q(x))^2 (Σ_{i=0}^{k-1} a_i x^i);
4.          return ⌊r(x)/x^{k-2}⌋
        end
```

Fig. 8.2. Algorithm to compute polynomial reciprocals.

Example 8.3. Let us compute $\lfloor x^{14}/p(x) \rfloor$, where $p(x) = x^7 - x^6 + x^5 + 2x^4 - x^3 - 3x^2 + x + 4$. In line 2 we compute the reciprocal of $x^3 - x^2 + x + 2$, that is, $q(x) = \lfloor x^6/(x^3 - x^2 + x + 2) \rfloor$. You may verify that $q(x) = x^3 + x^2 - 3$. Since $k = 8$, line 3 computes $r(x) = 2q(x)x^{10} - (q(x))^2 p(x) = x^{13} + x^{12} - 3x^{10} - 4x^9 + 3x^8 + 15x^7 + 12x^6 - 42x^5 - 34x^4 + 39x^3 + 51x^2 - 9x - 36$. Then at line 4, the result is $s(x) = x^7 + x^6 - 3x^4 - 4x^3 + 3x^2 + 15x + 12$. We note that $s(x)p(x)$ is x^{14} plus a polynomial of degree 6. □

Theorem 8.6. Algorithm 8.3 correctly computes the reciprocal of a polynomial.

Proof. We prove by induction on k, for k a power of 2, that if $s(x) =$ RECIPROCAL($p(x)$), and $p(x)$ is of degree $k - 1$, then $s(x)p(x) = x^{2k-2} + t(x)$, where $t(x)$ is of degree less than $k - 1$. The basis, $k = 1$, is trivial, since $p(x) = a_0$, $s(x) = 1/a_0$, and $t(x)$ need not exist.

For the inductive step, let $p(x) = p_1(x)x^{k/2} + p_2(x)$, where $\text{DEG}(p_1) = k/2 - 1$ and $\text{DEG}(p_2) \leq k/2 - 1$. Then by the inductive hypothesis, if $s_1(x) =$ RECIPROCAL($p_1(x)$), then $s_1(x)p_1(x) = x^{k-2} + t_1(x)$, where $\text{DEG}(t_1) < k/2 - 1$. At line 3, we compute

$$r(x) = 2s_1(x)x^{(3/2)k-2} - (s_1(x))^2(p_1(x)x^{k/2} + p_2(x)). \tag{8.14}$$

It suffices to show that $r(x)p(x)$ is x^{3k-4} plus terms of degree less than $2k - 3$. Then division by x^{k-2} at line 4 produces the desired result.

By (8.14) and the fact that $p(x) = p_1(x)x^{k/2} + p_2(x)$, we have

$$r(x)p(x) = 2s_1(x)p_1(x)x^{2k-2} + 2s_1(x)p_2(x)x^{(3/2)k-2} - (s_1(x)p_1(x)x^{k/2} + s_1(x)p_2(x))^2. \tag{8.15}$$

In substituting $x^{k-2} + t_1(x)$ for $s_1(x)p_1(x)$, in (8.15) we obtain

$$r(x)p(x) = x^{3k-4} - (t_1(x)x^{k/2} + s_1(x)p_2(x))^2. \tag{8.16}$$

Since $\text{DEG}(t_1) < k/2 - 1$, and $s_1(x)$ and $p_2(x)$ are of degree at most $k/2 - 1$, the terms other than x^{3k-4} in (8.16) are of degree at most $2k - 4$. □

The running times of Algorithms 8.3 and 8.1 are clearly analogous when one considers the two measures of complexity (arithmetic and bitwise, respectively) being used. In a similar manner we can show that the other time bounds of Section 8.2 apply to polynomials with arithmetic steps substituted for bit ones. Thus we have the following theorem.

Theorem 8.7. Let $M(n)$, $D(n)$, $R(n)$, and $S(n)$ be the arithmetic complexities of univariate polynomial multiplication, division, reciprocal-taking, and squaring respectively. These functions are all related by constant factors.

Proof. Analogous to Theorem 8.5 and the results leading up to that theorem. □

Corollary. Division of a $2n$th-degree polynomial by an nth-degree polynomial can be done in time $O_A(n \log n)$.

Proof. By Corollary 3 to Theorem 7.4 (p. 269) and Theorem 8.7. □

8.4 MODULAR ARITHMETIC

There are certain applications in which it is convenient to do integer arithmetic in a "modular" notation. That is, instead of representing an integer by a fixed-radix notation, we represent the integer by its residues modulo a set of pairwise relatively prime integers. If $p_0, p_1, \ldots, p_{k-1}$ are pairwise relatively prime integers and $p = \prod_{i=0}^{k-1} p_i$, then we can represent any integer u, $0 \leq u < p$, uniquely by the set of residues $u_0, u_1, \ldots, u_{k-1}$ where $u_i = u$ modulo p_i, for $0 \leq i < k$. When $p_0, p_1, \ldots, p_{k-1}$ are known, we write $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

It is quite easy to do addition, subtraction, and multiplication, provided the results continue to lie between 0 and $p - 1$ (or alternatively, these calculations can be regarded as being done modulo p). That is, let

$$u \leftrightarrow (u_0, u_1, \ldots, u_{k-1}) \quad \text{and} \quad v \leftrightarrow (v_0, v_1, \ldots, v_{k-1}).$$

Then

$$u + v \leftrightarrow (w_0, w_1, \ldots, w_{k-1}), \quad \text{where} \quad w_i = (u_i + v_i) \text{ modulo } p_i, \tag{8.17}$$

$$u - v \leftrightarrow (x_0, x_1, \ldots, x_{k-1}), \quad \text{where} \quad x_i = (u_i - v_i) \text{ modulo } p_i, \tag{8.18}$$

$$uv \leftrightarrow (y_0, y_1, \ldots, y_{k-1}), \quad \text{where} \quad y_i = u_i v_i \text{ modulo } p_i. \tag{8.19}$$

Example 8.4. Let $p_0 = 5$, $p_1 = 3$, and $p_2 = 2$. Then $4 \leftrightarrow (4, 1, 0)$, since $4 =$ 4 modulo 5, $1 = 4$ modulo 3, and $0 = 4$ modulo 2. Similarly, $7 \leftrightarrow (2, 1, 1)$ and $28 \leftrightarrow (3, 1, 0)$. We observe that by (8.19) above, $4 \times 7 \leftrightarrow (3, 1, 0)$, which is the representation of 28. That is, the first component of 4×7 is 4×2 modulo 5, which is 3; the second component is 1×1 modulo 3, which is 1; and the last component is 0×1 modulo 2, which is 0. Also, $4 + 7 \leftrightarrow (1, 2, 1)$, which is the representation of 11, and $7 - 4 \leftrightarrow (3, 0, 1)$, which is the representation of 3. □

However, it is not clear how to do division economically using modular arithmetic. Note that u/v is not necessarily an integer and even if it were, we could not necessarily find its modular representation by computing (u_i/v_i) modulo p_i for each i. In fact, if p_i is not a prime, there may be several integers w between 0 and $p_i - 1$ which could be (u_i/v_i) modulo p_i in the sense that $wv_i \equiv u_i$ modulo p_i. For example, if $p_i = 6$, $v_i = 3$, and $u_i = 3$, then w could be 1, 3, or 5, since $1 \times 3 \equiv 3 \times 3 \equiv 5 \times 3 \equiv 3$ modulo 6. Thus (u_i/v_i) modulo p_i may not "make sense."

The advantage of modular representation is chiefly that arithmetic can be implemented with less hardware than is required for conventional arithmetic,

since calculations are done for each modulus independently of the others. No carries are needed as for the usual (radix) number representations. Unfortunately, the problems of doing division and of detecting overflows (telling whether the result is outside the range 0 to $p-1$) efficiently appear insurmountable, and because of this such systems are rarely implemented in general purpose computer hardware.

Nevertheless, the ideas involved do find use, mostly in the polynomial domain. Here, we are likely to find ourselves in a situation where no polynomial division is required. Also, we shall see in the next section that evaluation of polynomials and computation of residues of polynomials (modulo other polynomials) are closely related. We first prove that modular arithmetic for integers "works" as intended.

The first part of the proof is that Eqs. (8.17), (8.18), and (8.19) hold. These relationships are straightforward, and we leave them as exercises. The second part of the proof is to show that the correspondence $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is one-to-one (an isomorphism). Although this result is not hard, we give it as a lemma.

Lemma 8.1. Let $p_0, p_1, \ldots, p_{k-1}$ be a set of integers which are pairwise relatively prime. Let

$$p = \prod_{i=0}^{k-1} p_i$$

and let $u_i = u$ modulo p_i. Then $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is a one-to-one correspondence between integer u, $0 \le u < p$, and

$$(u_0, u_1, \ldots, u_{k-1}), \quad 0 \le u_i < p_i,$$

for $0 \le i < k$.

Proof. Clearly, for each u there exists a corresponding k-tuple. Since there are exactly p values of u in the interval and exactly p k-tuples, it suffices to show that to each k-tuple there corresponds at most one integer u. Assume u and v, where $0 \le u < v < p$, both correspond to the k-tuple $(u_0, u_1, \ldots, u_{k-1})$. Then $v-u$ must be a multiple of each p_i. Since the p_i's are relatively prime, $v-u$ must be a multiple of p. Since $u \ne v$, and $v-u$ is a multiple of p, then u and v must differ by at least p and hence cannot both be in the interval 0 to $p-1$. □

In order to use modular arithmetic, algorithms are needed to convert from radix notation to modular notation and back. One method to convert an integer u from radix notation to modular notation is to divide u by each of the p_i, $0 \le i < k$.

Assume that each p_i requires b bits in binary notation. Then

$$p = \prod_{i=0}^{k-1} p_i$$

requires roughly bk bits, and division of u by each of the p_i's, where $0 \le u < p$, could involve k divisions of a kb-bit integer by integers of b bits. By breaking each division into k divisions of $2b$-bit integers by b-bit integers, we can convert to modular notation in $O_B(k^2 D(b))$ time, where $D(n)$ is the time to do integer division [at most $O_B(n \log n \operatorname{loglog} n)$ by the corollary to Theorem 8.5].

However, we can do the job in considerably less time by a method reminiscent of the technique used to perform polynomial division in Section 7.2. Instead of dividing an integer u by each of k moduli $p_0, p_1, \ldots, p_{k-1}$, we first compute the products $p_0p_1, p_2p_3, \ldots, p_{k-2}p_{k-1}$, then the products $p_0p_1p_2p_3, p_4p_5p_6p_7, \ldots$ and so on. Next we compute the residues by a divide-and-conquer approach. By division we obtain the residues u_1 and u_2 of u modulo $p_0 \cdots p_{k/2-1}$ and u modulo $p_{k/2} \cdots p_{k-1}$. The problem of computing u modulo p_i, $0 \le i < k$, is now reduced to two problems of half size. That is, u modulo $p_i = u_1$ modulo p_i for $0 \le i < k/2$, and u modulo $p_i = u_2$ modulo p_i for $k/2 \le i < k$.

Algorithm 8.4. Computation of residues.

Input. Moduli $p_0, p_1, \ldots, p_{k-1}$, and integer u, where $0 \le u < p = \Pi_{i=0}^{k-1} p_i$.

Output. u_i, $0 \le i < k$, where $u_i = u$ modulo p_i.

Method. Assume k is a power of 2, say $k = 2^t$. (If necessary, add extra moduli, all of which are 1, to the input so k becomes a power of 2.) We begin by computing certain products of the moduli similar to the q_{tm}'s computed in Section 7.2.

For $0 \le j < t$, i a multiple of 2^j, and $0 \le i < k$, let

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m.$$

Thus $q_{i0} = p_i$ and $q_{ij} = q_{i,j-1} \times q_{i+2^{j-1},j-1}$.

We first compute the q_{ij}'s, then find the remainder u_{ij} when u is divided by each of the q_{ij}'s. The desired answers are the u_{i0}'s. The details are in the program of Fig. 8.3. □

Theorem 8.8. Algorithm 8.4 correctly computes the u_i's.

Proof. The proof parallels that of Theorem 7.3, where a polynomial was evaluated at the nth roots of unity. It is easy to show by induction on j that line 4 computes the q_{ij}'s correctly. Then, by backwards induction on the j of line 6, we show that $u_{ij} = u$ modulo q_{ij}. Lines 8 and 9 make this easy. Letting $j = 0$, we have $u_i = u$ modulo p_i. Details are left as an exercise. □

Theorem 8.9. Algorithm 8.4 requires $O_B(M(bk) \log k)$ time if at most b bits are needed to represent each of the p_i's.

Proof. It is easy to see that the loops of lines 3–4 and 7–9 are the

```
begin
1.    for i ← 0 until k − 1 do q_{i0} ← p_i;
2.    for j ← 1 until t − 1 do
3.        for i ← 0 step 2^j until k − 1 do
4.            q_{ij} ← q_{i,j−1} * q_{i+2^{j−1},j−1};
5.    u_{0t} ← u;
6.    for j ← t step −1 until 1 do
7.        for i ← 0 step 2^j until k − 1 do
              begin
8.                u_{i,j−1} ← REMAINDER(u_{ij}/q_{i,j−1});
9.                u_{i+2^{j−1},j−1} ← REMAINDER(u_{ij}/q_{i+2^{j−1},j−1})
              end;
10.   for i ← 0 until k − 1 do u_i ← u_{i0}
end
```

Fig. 8.3. Computation of residues.

most costly. Each requires $O_B(2^{t-j}M(2^{j-1}b))$ time.† Since we assume $M(an) \geq aM(n)$ for $a \geq 1$, we see the cost of these loops is bounded by $O_B(M(2^t b)) = O_B(M(kb))$. Since each of these loops is executed $t = \log k$ times at most, we have our result. □

Corollary. If b bits are required to represent each of the moduli $p_0, p_1, \ldots, p_{k-1}$ then the residues may be computed in at most $O_B(bk \log k \log bk \operatorname{loglog} bk)$ time.

8.5 MODULAR POLYNOMIAL ARITHMETIC AND POLYNOMIAL EVALUATION

Results analogous to those for integers hold for polynomials. Let $p_0, \ldots, p_{k-1}$ be polynomials and $p = \Pi_{i=0}^{k-1} p_i$. Then each polynomial u can be represented by the sequence $u_0, u_1, \ldots, u_{k-1}$ of remainders obtained by dividing u by each p_i. That is, u_i is the unique polynomial with $\mathrm{DEG}(u_i) < \mathrm{DEG}(p_i)$ such that $u = p_i q_i + u_i$ for some polynomial q_i. We write $u_i = u$ modulo p_i in this situation, in complete analogy with integer modular arithmetic.

In analogy with Lemma 8.1, we may show that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is a one-to-one correspondence if the p_i's are pairwise relatively prime and u is restricted to have degree less than that of p, i.e., $\mathrm{SIZE}(u) < \mathrm{SIZE}(p)$. More importantly, Algorithm 8.4 for computing residues works if the p_i's are polynomials instead of integers. Instead of b (the number of bits in the p_i's), we

† Recall that $D(n)$ and $M(n)$ are essentially the same.

must consider the maximum degree of the polynomials p_i. Of course, complexity is in terms of arithmetic steps rather than bitwise ones. With these changes, we have the following analog of Theorem 8.9.

Theorem 8.10. By a conversion of Algorithm 8.4 to the polynomial domain, it is possible to find the residues with respect to polynomials $p_0, p_1, \ldots, p_{k-1}$ of polynomial u in time $O_A(M(dk) \log k)$, where d is an upper bound on the degree of the p_i's and the degree of u is less than that of $\Pi_{i=0}^{k-1} p_i$.

Proof. Analogous to Theorem 8.9 and left for an exercise. □

Corollary 1. Finding residues of a polynomial u with respect to polynomials $p_0, p_1, \ldots, p_{k-1}$ requires time at most $O_A(dk \log k \log dk)$, where d is an upper bound on the degree of the p_i's and the degree of u is less than that of $\Pi_{i=0}^{k-1} p_i$.

Example 8.5. Consider the four polynomial moduli

$$\begin{aligned} p_0 &= x - 3, \\ p_1 &= x^2 + x + 1, \\ p_2 &= x^2 - 4, \\ p_3 &= 2x + 2, \end{aligned}$$

and suppose $u = x^5 + x^4 + x^3 + x^2 + x + 1$. First compute the products

$$\begin{aligned} p_0 p_1 &= x^3 - 2x^2 - 2x - 3, \\ p_2 p_3 &= 2x^3 + 2x^2 - 8x - 8. \end{aligned}$$

Then, compute

$$\begin{aligned} u' &= u \text{ modulo } p_0 p_1 = 28x^2 + 28x + 28, \\ u'' &= u \text{ modulo } p_2 p_3 = 21x + 21. \end{aligned}$$

That is, $u = p_0 p_1 (x^2 + 3x + 9) + 28x^2 + 28x + 28$, and $u = p_2 p_3 (\frac{1}{2}x^2 + \frac{5}{2}) + 21x + 21$.

Next, compute

$$\begin{aligned} u \text{ modulo } p_0 &= u' \text{ modulo } p_0 = 364, \\ u \text{ modulo } p_1 &= u' \text{ modulo } p_1 = 0, \\ u \text{ modulo } p_2 &= u'' \text{ modulo } p_2 = 21x + 21, \\ u \text{ modulo } p_3 &= u'' \text{ modulo } p_3 = 0. \end{aligned}$$ □

Note that the FFT algorithm of Section 7.2 is really an implementation of this algorithm, where the polynomials $p_0, p_1, \ldots, p_{k-1}$ are $x - \omega^0, x - \omega^1, \ldots, x - \omega^{n-1}$. The FFT algorithm took advantage of the fact that $p_i = x - \omega^i$. Because of the ordering of the p_i's, each product had the form of a power of x minus a power of ω and hence division was especially easy.

As we observed in Section 7.2, if p_i is the first-degree polynomial $x - a$, then u modulo p_i is $u(a)$. Therefore, the case in which all the p_i's are of degree 1 is especially important. We have the following corollary to Theorem 8.10.

Corollary 2. An nth-degree polynomial can be evaluated at n points in time $O_A(n \log^2 n)$.

Proof. To evaluate $u(x)$ at the n points $a_0, a_1, \ldots, a_{n-1}$ we compute $u(x)$ modulo$(x - a_i)$ for $0 \le i < n$. This evaluation requires $O_A(n \log^2 n)$ time by Corollary 1, since d there is 1. □

8.6 CHINESE REMAINDERING

We now consider the problem of converting an integer from modular notation to radix notation.† Suppose we are given relatively prime moduli $p_0, p_1, \ldots, p_{k-1}$ and residues $u_0, u_1, \ldots, u_{k-1}$, where $k = 2^t$, and we wish to find the integer u such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$. We may do so by the integer analog of the *Lagrangian interpolation formula* for polynomials.

Lemma 8.2. Let c_i be the product of all the p_j's except p_i (that is, $c_i = p/p_i$ where $p = \prod_{j=0}^{k-1} p_j$). Let d_i be c_i^{-1} modulo p_i (that is, $d_i c_i \equiv 1$ modulo p_i, and $0 \le d_i < p_i$). Then

$$u \equiv \sum_{i=0}^{k-1} c_i d_i u_i \text{ modulo } p. \tag{8.20}$$

Proof. Since the p_j's are relatively prime to one another, we know d_i exists and is unique (Exercise 7.9). Also, c_i is divisible by p_j for $j \ne i$, so $c_i d_i u_i \equiv 0$ modulo p_j if $j \ne i$. Thus

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv c_j d_j u_j \text{ modulo } p_j.$$

Since $c_j d_j \equiv 1$ modulo p_j, we have

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv u_j \text{ modulo } p_j.$$

Since p_j divides p, these relationships hold even if all arithmetic is done modulo p. Thus (8.20) holds. □

Our problem is to compute (8.20) efficiently. To begin, it is hardly clear how to compute the d_i's from the p_i's except by trial and error. We shall later see that this task is not hard, given the Euclidean algorithm of Section 8.8 and the fast implementation in Section 8.10. However, in this section we shall study only the "preconditioned" form of the Chinese remainder problem. If a portion of the input is fixed for a number of problems, then all constants depending on the fixed portion of the input can be precomputed and supplied as part of the input. If any such precomputation of constants is done, the algorithm is said to be *preconditioned.*

† This process is known as *Chinese remaindering*, since an algorithm for the process was known to the Chinese over 2000 years ago.

For the preconditioned Chinese remainder algorithm, the input will be not only the moduli and the p_i's, but also the inverses (the d_i's). This situation is not unrealistic. If we are frequently using the Chinese remainder algorithm to convert numbers represented by a fixed set of moduli, it is reasonable to precompute all functions of these moduli that are used by the algorithm. In the corollary to Theorem 8.21 we shall see that as far as order of magnitude is concerned, it makes only a modest difference whether the algorithm is preconditioned or not.

If we look at (8.20), we notice that the terms $c_i d_i u_i$ have many factors in common as i varies. For example, $c_i d_i u_i$ has $p_0 p_1 \cdots p_{k/2-1}$ as a factor whenever $i \ge k/2$, and it has $p_{k/2} p_{k/2+1} \cdots p_{k-1}$ as a factor if $i < k/2$. Thus we could write (8.20) as

$$u = \left(\sum_{i=0}^{k/2-1} c_i' d_i u_i \right) \times \prod_{i=k/2+1}^{k-1} p_i + \left(\sum_{i=k/2+1}^{k-1} c_i'' d_i u_i \right) \times \prod_{i=0}^{k/2-1} p_i$$

where c_i' is the product $p_0 p_1 \cdots p_{(k/2)-1}$ with p_i missing, and c_i'' is the product $p_{k/2} p_{k/2+1} \cdots p_{k-1}$ with p_i missing. This observation should suggest a divide-and-conquer approach similar to that used for computing residues. We compute the products

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$$

(as in Algorithm 8.4) and then the integers

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

If $j = 0$, then $s_{i0} = d_i u_i$. If $j > 0$, we compute s_{ij} by the formula

$$s_{ij} = s_{i,j-1} q_{i+2^{j-1},j-1} + s_{i+2^{j-1},j-1} q_{i,j-1}.$$

Ultimately we produce $s_{0t} = u$, which evaluates (8.20).

Algorithm 8.5. Preconditioned fast Chinese remainder algorithm.

Input

1. Relatively prime integer moduli $p_0, p_1, \ldots, p_{k-1}$, where $k = 2^t$ for some t.
2. The set of "inverses" $d_0, d_1, \ldots, d_{k-1}$ such that $d_i = (p/p_i)^{-1}$ modulo p_i, where $p = \prod_{i=0}^{k-1} p_i$.
3. A sequence of residues $(u_0, u_1, \ldots, u_{k-1})$.

```
     begin
1.       for i ← 0 until k − 1 do s_i0 ← d_i * u_i;
2.       for j ← 1 until t do
3.           for i ← 0 step 2^j until k − 1 do
4.               s_ij ← s_i,j−1 * q_i+2^(j−1),j−1 + s_i+2^(j−1),j−1 * q_i,j−1;
5.       write s_0t modulo q_0t
     end
```

Fig. 8.4. Program to compute integer from modular representation.

Output. The unique integer u, $0 \le u < p$, such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Method. First compute $q_{ij} = \Pi_{m=i}^{i+2^j-1} p_m$ as in Algorithm 8.4.† Then execute the program in Fig. 8.4, where s_{ij} is intended to be

$$\sum_{m=}^{i+2^j-1} \frac{q_{ij} d_m u_m}{p_m}. \quad \square$$

Example 8.6. Let p_0, p_1, p_2, p_3 be 2, 3, 5, 7, and let (u_0, u_1, u_2, u_3) be (1, 2, 4, 3). Then $q_{i0} = p_i$ for $0 \le i < 4$, $q_{01} = 6$, $q_{21} = 35$, and $q_{02} = p = 210$. We note that

$$\begin{aligned}
d_0 &= (3 * 5 * 7)^{-1} \text{ modulo } 2 = 1, && \text{since } 1 * 105 \equiv 1 \text{ modulo } 2,\\
d_1 &= (2 * 5 * 7)^{-1} \text{ modulo } 3 = 1, && \text{since } 1 * 70 \equiv 1 \text{ modulo } 3,\\
d_2 &= (2 * 3 * 7)^{-1} \text{ modulo } 5 = 3, && \text{since } 3 * 42 \equiv 1 \text{ modulo } 5,\\
d_3 &= (2 * 3 * 5)^{-1} \text{ modulo } 7 = 4, && \text{since } 4 * 30 \equiv 1 \text{ modulo } 7.
\end{aligned}$$

Thus the effect of line 1 is to compute

$$\begin{aligned}
s_{00} &= 1 * 1 = 1, & s_{10} &= 1 * 2 = 2,\\
s_{20} &= 3 * 4 = 12, & s_{30} &= 4 * 3 = 12.
\end{aligned}$$

We then execute the loop of lines 3–4 with $j = 1$. Here, i takes on the values 0 and 2; so we compute

$$\begin{aligned}
s_{01} &= s_{00} * q_{10} + s_{10} * q_{00} = 1 * 3 + 2 * 2 = 7,\\
s_{21} &= s_{20} * q_{30} + s_{30} * q_{20} = 12 * 7 + 12 * 5 = 144.
\end{aligned}$$

Next, we execute the loop of lines 3–4 with $j = 2$, and i takes only the value 0. We compute

$$s_{02} = s_{01} * q_{21} + s_{21} * q_{01} = 7 * 35 + 144 * 6 = 1109.$$

† Note that the q_{ij}'s are functions only of the p_i's. We should rightly include them as inputs rather than calculating them, since we allow preconditioning. However, it is easily shown that the order-of-magnitude running time is not affected by whether or not the q_{ij}'s are precomputed.

The result at line 5 is thus 1109 modulo 210, which is 59. You may check that the residues of 59 modulo 2, 3, 5, and 7 are 1, 2, 4, and 3, respectively. Figure 8.5 graphically portrays the computations. $\square$

Theorem 8.11. Algorithm 8.5 correctly computes the integer u such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Proof. An elementary induction on j proves that s_{ij} is given its intended value, that is

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

The correctness of the algorithm then follows immediately from Lemma 8.2, that is, from the correctness of Eq. (8.20). $\square$

Theorem 8.12. Suppose we are given k relatively prime integer moduli $p_0, p_1, \ldots, p_{k-1}$ and residues $(u_0, u_1, \ldots, u_{k-1})$. If each of the p_i's requires at most b bits, there is a preconditioned algorithm that computes u such that $0 \le u < p = \Pi_{i=0}^{k-1} p_i$ and $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ in time $O_B(M(bk) \log k)$, where $M(n)$ is the time to multiply two n-bit numbers.

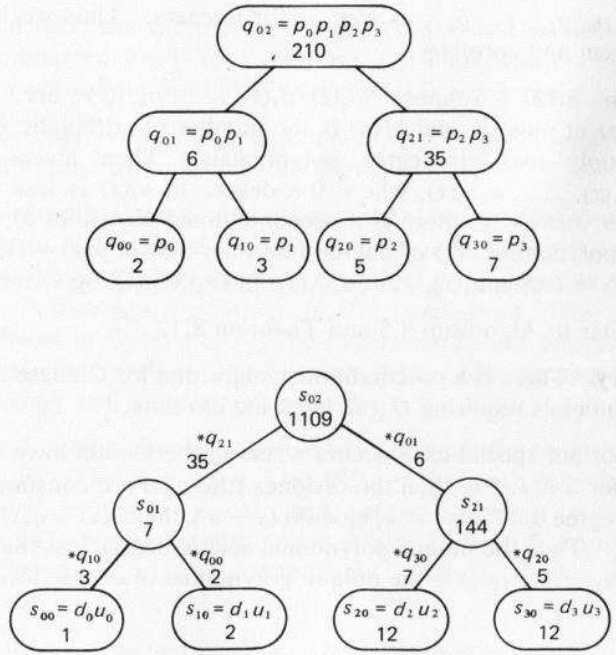

Fig. 8.5 Computations of Example 8.6.

Proof. The computation of the q_{ij}'s requires $O_B(M(bk) \log k)$ time.† For the analysis of the body of the algorithm, note that s_{ij} requires at most $b2^j + b + j$ bits, since it is the sum of 2^j terms, each of which is the product of $2^j + 1$ integers of b or fewer bits. Hence each term requires no more than $b(2^j + 1)$ bits, and the sum of 2^j such terms requires at most $b(2^j + 1) + \log(2^j) = b2^j + b + j$ bits. Thus line 4 takes time $O_B(M(b2^j))$. The loop of lines 3–4 is executed $k/2^j$ times for fixed j, so the total cost of the loop is

$$O_B\left(\frac{k}{2^j} M(b2^j)\right),$$

which, by our usual assumption about the growth of $M(n)$, is bounded above by $O_B(M(bk))$. Since the loop of lines 2–4 is iterated $\log k$ times, the total cost is $O_B(M(bk) \log k)$ time. Line 5 is easily shown to cost less than this. □

Corollary. The preconditioned Chinese remaindering algorithm with k moduli of b bits each requires at most $O_B(bk \log k \log bk \operatorname{loglog} bk)$ steps.

8.7 CHINESE REMAINDERING AND INTERPOLATION OF POLYNOMIALS

It should be clear that all the results of the previous section hold for polynomial moduli $p_0, p_1, \ldots, p_{k-1}$ as well as for integers. Thus we have the following theorem and corollary.

Theorem 8.13. Suppose $p_0(x), p_1(x), \ldots, p_{k-1}(x)$ are polynomials of degree at most d, and $M(n)$ is the number of arithmetic steps needed to multiply two nth-degree polynomials. Then given polynomials $u_0(x), u_1(x), \ldots, u_{k-1}(x)$, where the degree of $u_i(x)$ is less than that of $p_i(x)$, for $0 \le i < k$, there is a preconditioned algorithm to compute the unique polynomial $u(x)$ of degree less than that of $p(x) = \Pi_{i=0}^{k-1} p_i(x)$ such that $u(x) \leftrightarrow (u_0(x), u_1(x), \ldots, u_{k-1}(x))$ in $O_A(M(dk) \log k)$ time.

Proof. Similar to Algorithm 8.5 and Theorem 8.12. □

Corollary. There is a preconditioned algorithm for Chinese remaindering of polynomials requiring $O_A(dk \log k \log dk)$ time.

An important special case occurs when all the moduli have degree 1. If $p_i = x - a_i$ for $0 \le i < k$, then the residues (the u_i's) are constants, i.e., polynomials of degree 0. If $u(x) \equiv u_i$ modulo $(x - a_i)$, then $u(x) = q(x)(x - a_i) + u_i$, so $u(a_i) = u_i$. Thus the unique polynomial $u(x)$ of degree less than k such that $u(x) \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is the unique polynomial of degree less than k such that $u(a_i) = u_i$ for each i, $0 \le i < k$. Put another way, $u(x)$ is the polynomial interpolated through the points (a_i, u_i) for $0 \le i < k$.

Since interpolation is an important polynomial operation, it is pleasant to observe that interpolation through k points can be done in $O_A(k \log^2 k)$ time even if preconditioning is not allowed. This is so because, as we shall see from the next lemma, the coefficients d_i from Eq. (8.20) can be evaluated easily in this special case.†

Lemma 8.3. Let $p_i(x) = x - a_i$, for $0 \le i < k$, where the a_i's are distinct (i.e., the $p_i(x)$'s are relatively prime). Let $p(x) = \Pi_{j=0}^{k-1} p_j(x)$, let $c_i(x) = p(x)/p_i(x)$ and let $d_i(x)$ be the constant polynomial such that $d_i(x)c_i(x) \equiv 1$ modulo $p_i(x)$. Then $d_i(x) = 1/b$, where $b = \dfrac{d}{dx} p(x)\Big|_{x=a_i}$.

Proof. We can write $p(x) = c_i(x)p_i(x)$, so

$$\frac{d}{dx} p(x) = p_i(x) \frac{d}{dx} c_i(x) + c_i(x) \frac{d}{dx} p_i(x). \tag{8.21}$$

Now, $dp_i(x)/dx = 1$, and $p_i(a_i) = 0$. Thus

$$\left.\frac{dp(x)}{dx}\right|_{x=a_i} = c_i(a_i). \tag{8.22}$$

Note that $d_i(x)$ has the property that $d_i(x)c_i(x) \equiv 1$ modulo $(x - a_i)$, so $d_i(x)c_i(x) = q_i(x)(x - a_i) + 1$ for some $q_i(x)$. Thus $d_i(a_i) = 1/c_i(a_i)$. The lemma is now immediate from (8.22) since $d_i(x)$ is a constant. □

Theorem 8.14. We may interpolate a polynomial through k points in $O_A(k \log^2 k)$ time without preconditioning.

Proof. By Lemma 8.3, the computation of the d_i's is equivalent to evaluating the derivative of a $(k-1)$st-degree polynomial at k points. The polynomial $p(x) = \Pi_{i=0}^{k-1} p_i(x)$ can be obtained in $O_A(k \log^2 k)$ time by first computing $p_0p_1, p_2p_3, \ldots$, then $p_0p_1p_2p_3, p_4p_5p_6p_7, \ldots$, and so on. The derivative of $p(x)$ can be taken in $O_A(k)$ steps. The evaluation of the derivative requires $O_A(k \log^2 k)$ time by Corollary 2 to Theorem 8.10. The theorem then follows from the corollary to Theorem 8.13 with $d = 1$. □

Example 8.7. Let us interpolate a polynomial through the points (1, 2), (2, 7), (3, 4), and (4, 8). That is, $a_i = i + 1$ for $0 \le i < 4$, $u_0 = 2$, $u_1 = 7$, $u_2 = 4$, and $u_3 = 8$. Then $p_i(x) = x - i - 1$, and $p(x) = \Pi_{i=0}^{3} p_i(x)$ is $x^4 - 10x^3 + 35x^2 - 50x + 24$. Next, $dp(x)/dx = 4x^3 - 30x^2 + 70x - 50$, and its values at 1, 2, 3, 4 are $-6, +2, -2, +6$, respectively. Thus d_0, d_1, d_2, and d_3 are $-\frac{1}{6}$,

† Since $D(n)$ and $M(n)$ are essentially the same functions, we use $M(n)$ in preference throughout.

† As was alluded to in Section 8.6, the task is really not hard in the general case. However, in the general case, we need the machinery of the next section.

$+\frac{1}{2}$, $-\frac{1}{2}$, and $+\frac{1}{6}$, respectively. Using the fast Chinese remainder algorithm redone for polynomials (Algorithm 8.5) we compute:

$$s_{01} = d_0u_0p_1 + d_1u_1p_0 = (-\tfrac{1}{6})(2)(x-2) + (\tfrac{1}{2})(7)(x-1) = \tfrac{19}{6}x - \tfrac{17}{6},$$
$$s_{21} = d_2u_2p_3 + d_3u_3p_2 = (-\tfrac{1}{2})(4)(x-4) + (\tfrac{1}{6})(8)(x-3) = -\tfrac{2}{3}x + 4.$$

Then

$$s_{02} = u(x) = s_{01}q_{21} + s_{21}q_{01}$$
$$= (\tfrac{19}{6}x - \tfrac{17}{6})(x^2 - 7x + 12) + (-\tfrac{2}{3}x + 4)(x^2 - 3x + 2),$$
$$u(x) = \tfrac{5}{2}x^3 - 19x^2 + \tfrac{89}{2}x - 26. \square$$

As we mentioned in Chapter 7, we can do polynomial arithmetic, such as addition, subtraction, and multiplication, by evaluating polynomials at n points, performing the arithmetic on the values at the points, and then interpolating a polynomial through the resulting values. If the answer is a polynomial of degree $n-1$ or less, this technique will yield the correct answer.

The FFT is a method of doing just this, where the points selected are $\omega^0, \omega^1, \ldots, \omega^{n-1}$. In this case, the evaluation and interpolation algorithms were made especially easy because of properties of the powers of ω and the particular order of these powers that we chose. However, it is worth noting that we could use any collection of points to substitute for the powers of ω. Then we would have a "transform" that required $O_A(n \log^2 n)$, rather than $O_A(n \log n)$ time, to compute and invert.

8.8 GREATEST COMMON DIVISORS AND EUCLID'S ALGORITHM

Definition: Let a_0 and a_1 be positive integers. A positive integer g is called a *greatest common divisor* of a_0 and a_1, often denoted $\text{GCD}(a_0, a_1)$, if

1. g divides both a_0 and a_1, and
2. every divisor of both a_0 and a_1 divides g.

It is easy to show that if a_0 and a_1 are positive integers, then g is unique. For example, GCD(57, 33) is 3.

Euclid's algorithm for computing $\text{GCD}(a_0, a_1)$ is to compute the *remainder sequence* $a_0, a_1, \ldots, a_k$, where a_i, for $i \geq 2$, is the nonzero remainder resulting from the division of a_{i-2} by a_{i-1}, and where a_k divides a_{k-1} exactly (i.e., $a_{k+1} = 0$). Then $\text{GCD}(a_0, a_1) = a_k$.

Example 8.8. In the example above, $a_0 = 57$, $a_1 = 33$, $a_2 = 24$, $a_3 = 9$, $a_4 = 6$, and $a_5 = 3$. Thus $k = 5$ and GCD(57, 33) = 3. $\square$

Theorem 8.15. Euclid's algorithm correctly computes $\text{GCD}(a_0, a_1)$.

Proof. The algorithm computes $a_{i+1} = a_{i-1} - q_i a_i$ for $1 \leq i < k$, where $q_i = \lfloor a_{i-1}/a_i \rfloor$. Since $a_{i+1} < a_i$, the algorithm will clearly terminate. Moreover,


```
     begin
1.       x0 ← 1; y0 ← 0; x1 ← 0; y1 ← 1; i ← 1;
2.       while ai does not divide ai−1 do
             begin
3.               q ← ⌊ai−1/ai⌋;
4.               ai+1 ← ai−1 − q * ai;
5.               xi+1 ← xi−1 − q * xi;
6.               yi+1 ← yi−1 − q * yi;
7.               i ← i + 1
             end
8.       write ai; write xi; write yi
     end
```

Fig. 8.6. Extended Euclidean algorithm.

any divisor of both a_{i-1} and a_i is a divisor of a_{i+1}, and any divisor of a_i and a_{i+1} is also a divisor of a_{i-1}. Hence $\text{GCD}(a_0, a_1) = \text{GCD}(a_1, a_2) = \cdots = \text{GCD}(a_{k-1}, a_k)$. Since $\text{GCD}(a_{k-1}, a_k)$ is clearly a_k, we have our result. $\square$

The Euclidean algorithm can be extended to find not only the greatest common divisor of a_0 and a_1, but also to find integers x and y such that $a_0x + a_1y = \text{GCD}(a_0, a_1)$. The algorithm is as follows.

Algorithm 8.6. Extended Euclidean algorithm.

Input. Positive integers a_0 and a_1.

Output. $\text{GCD}(a_0, a_1)$ and integers x and y such that $a_0x + a_1y = \text{GCD}(a_0, a_1)$.

Method. We execute the program in Fig. 8.6. $\square$

Example 8.9. If $a_0 = 57$ and $a_1 = 33$, we obtain the following values for the a_i's, x_i's, and y_i's.

i	a_i	x_i	y_i
0	57	1	0
1	33	0	1
2	24	1	−1
3	9	−1	2
4	6	3	−5
5	3	−4	7

Note that $57 \times (-4) + 33 \times 7 = 3$. $\square$

It should be clear that Algorithm 8.6 correctly computes $\text{GCD}(a_0, a_1)$, since the a_i's clearly form the remainder sequence. An important property of the x_i's and y_i's computed by Algorithm 8.6 is the subject of the next lemma.

Lemma 8.4. In Algorithm 8.6, for $i \geq 0$

$$a_0 x_i + a_1 y_i = a_i. \tag{8.23}$$

Proof. Equation (8.23) holds for $i = 0$ and $i = 1$ by line 1 of Algorithm 8.6. Assume (8.23) holds for $i - 1$ and i. Then $x_{i+1} = x_{i-1} - qx_i$ by line 5 and $y_{i+1} = y_{i-1} - qy_i$ by line 6. Thus

$$a_0 x_{i+1} + a_1 y_{i+1} = a_0 x_{i-1} + a_1 y_{i-1} - q(a_0 x_i + a_1 y_i). \tag{8.24}$$

By the inductive hypothesis and (8.24), we have

$$a_0 x_{i+1} + a_1 y_{i+1} = a_{i-1} - q a_i.$$

Since $a_{i-1} - q a_i = a_{i+1}$ by line 4, we have our result. □

Note that Lemma 8.4 does not even depend on the way q is computed at line 3, although line 3 is essential to guarantee that Algorithm 8.6 does compute $\mathrm{GCD}(a_0, a_1)$. We may put these observations together and prove the following theorem.

Theorem 8.16. Algorithm 8.6 computes $\mathrm{GCD}(a_0, a_1)$ and numbers x and y such that $a_0 x + a_1 y = \mathrm{GCD}(a_0, a_1)$.

Proof. Elementary exercise from Lemma 8.4. □

We now introduce some notation that will be useful in developments of the next section.

Definition. Let a_0 and a_1 be integers with remainder sequence $a_0, a_1, \ldots, a_k$. Let $q_i = \lfloor a_{i-1}/a_i \rfloor$, for $1 \leq i \leq k$. We define 2×2 matrices $R_{ij}^{(a_0, a_1)}$, or R_{ij} where a_0 and a_1 are understood for $0 \leq i \leq j \leq k$, by:

1. $R_{ii} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, for $i \geq 0$.

2. If $j > i$, then $R_{ij} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -q_{j-1} \end{bmatrix} * \cdots * \begin{bmatrix} 0 & 1 \\ 1 & -q_{i+1} \end{bmatrix}$.

Example 8.10. Let $a_0 = 57$ and $a_1 = 33$, with remainder sequence 57, 33, 24, 9, 6, 3 and quotients q_i, for $1 \leq i \leq 4$, given by 1, 1, 2, 1. Then

$$R_{04}^{(57,33)} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 3 & -5 \\ -4 & 7 \end{bmatrix}. \quad \square$$

Two interesting properties of these matrices are given in the next lemma.

Lemma 8.5

a) $\begin{bmatrix} a_j \\ a_{j+1} \end{bmatrix} = R_{ij} \begin{bmatrix} a_i \\ a_{i+1} \end{bmatrix}$ for $i < j < k$,

b) $R_{0j} = \begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix}$, for $0 \leq j < k$,

where the a_i's, x_i's, and y_i's are as defined in the extended Euclidean algorithm.

Proof. Elementary inductive exercises. □

We should observe that all the developments of this section go through with univariate polynomials instead of integers, with the following modification. While it is easy to show that the greatest common divisor of two integers is uniquely defined, for polynomials over a field the greatest common divisor is unique only up to multiplication by a constant. That is, if $g(x)$ divides polynomials $a_0(x)$ and $a_1(x)$, and any other divisor of these two polynomials also divides $g(x)$, then $cg(x)$ also has this property for any constant $c \neq 0$. We shall be satisfied with any polynomial that divides $a_0(x)$ and $a_1(x)$ and that is divisible by any divisor of these. To insure uniqueness we could (but don't) insist that the greatest common divisor be *monic*, i.e., that its highest-degree term have coefficient 1.

8.9 AN ASYMPTOTICALLY FAST ALGORITHM FOR POLYNOMIAL GCD'S

For a discussion of greatest common divisor algorithms we reverse our pattern and discuss polynomials first, since there are several extra details which must be handled when we adapt the algorithm to integers. Let $a_0(x)$ and $a_1(x)$ be the two polynomials whose greatest common divisor we wish to compute. We assume $\mathrm{DEG}(a_1(x)) < \mathrm{DEG}(a_0(x))$. This condition can be easily enforced as follows. If $\mathrm{DEG}(a_0) = \mathrm{DEG}(a_1)$, replace the polynomials a_0 and a_1 by a_1 and a_0 modulo a_1, i.e., the second and third terms of the remainder sequence, and proceed from there.

We shall break the problem into two parts. The first is to design an algorithm that obtains the last term in the remainder sequence whose degree is more than half that of a_0. Formally, let $l(i)$ be the unique integer such that $\mathrm{DEG}(a_{l(i)}) > i$ and $\mathrm{DEG}(a_{l(i)+1}) \leq i$. Note that if a_0 is of degree n, then $l(i) \leq n - i - 1$ on the assumption $\mathrm{DEG}(a_1) < \mathrm{DEG}(a_0)$, since $\mathrm{DEG}(a_i) \leq \mathrm{DEG}(a_{i-1}) - 1$ for all $i \geq 1$.

We now introduce a recursive procedure HGCD (half GCD) which takes polynomials a_0 and a_1, with $n = \mathrm{DEG}(a_0) > \mathrm{DEG}(a_1)$, and produces the matrix R_{0j} (see Section 8.8), where $j = l(n/2)$, that is, a_j is the last term of the remainder sequence whose degree exceeds half that of a_0.

The principle behind the HGCD algorithm is that quotients of polynomials of degrees d_1 and d_2, with $d_1 > d_2$, depend only on the leading $2(d_1 - d_2) + 1$ terms of the dividend and the leading $d_1 - d_2 + 1$ terms of the divisor. HGCD is defined in Fig. 8.7.

Example 8.11. Let

$$p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$$

```
      procedure HGCD(a_0, a_1):
1.    if DEG(a_1) ≤ DEG(a_0)/2 then return [1 0; 0 1]
      else
          begin
2.            let a_0 = b_0 x^m + c_0, where m = ⌊DEG(a_0)/2⌋ and
                  DEG(c_0) < m;
3.            let a_1 = b_1 x^m + c_1, where DEG(c_1) < m;
              comment b_0 and b_1 are the leading terms of a_0 and a_1.
                We have DEG(b_0) = ⌈DEG(a_0)/2⌉ and DEG(b_0) −
                DEG(b_1) = DEG(a_0) − DEG(a_1);
4.            R ← HGCD(b_0, b_1);
5.            [d; e] ← R [a_0; a_1];
6.            f ← d modulo e;
              comment e and f are successive terms in the remainder
                sequence, of degree at most ⌈3m/2⌉, that is, 3/4
                the degree of a_0;
7.            let e = g_0 x^⌊m/2⌋ + h_0, where DEG(h_0) < ⌊m/2⌋;
8.            let f = g_1 x^⌊m/2⌋ + h_1, where DEG(h_1) < ⌊m/2⌋;
              comment g_0 and g_1 are each of degree m + 1, at most;
9.            S ← HGCD(g_0, g_1);
10.           q ← ⌊d/e⌋;
11.           return S * [0 1; 1 −q] * R
          end
```

Fig. 8.7. The procedure HGCD.

and

$$p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7.$$

Suppose we attempt to compute HGCD(p_1, p_2). If $a_0 = p_1$ and $a_1 = p_2$, then at lines 2 and 3 we have $m = 2$, and

$$b_0 = x^3 + x^2 + x + 1, \qquad c_0 = x + 1,$$
$$b_1 = x^2 - 2x + 3, \qquad c_1 = -x - 7.$$

Then we call HGCD(b_0, b_1) at line 4. We may check that R is given the value

$$\begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix}$$

at that step. Next, at lines 5 and 6 we compute

$$d = x^4 - 2x^3 + 3x^2 - x - 7,$$
$$e = 4x^3 - 7x^2 + 11x + 22,$$
$$f = -\tfrac{3}{16}x^2 - \tfrac{93}{16}x - \tfrac{45}{8}.$$

We find $\lfloor m/2 \rfloor = 1$, so at lines 7 and 8 we obtain

$$g_0 = 4x^2 - 7x + 11, \qquad h_0 = 22,$$
$$g_1 = -\tfrac{3}{16}x - \tfrac{93}{16}, \qquad h_1 = -\tfrac{45}{8}.$$

Thus at line 9, we find

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

At line 10, $q(x)$, the quotient $\lfloor d(x)/e(x) \rfloor$, is found to be† $\frac{1}{4}x - \frac{1}{16}$. So at line 11 we have the result

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -(\frac{1}{4}x - \frac{1}{16}) \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix} = \begin{bmatrix} 1 & -(x+3) \\ -(\frac{1}{4}x - \frac{1}{16}) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Note that

$$T\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix},$$

which is correct since in the remainder sequence for p_1 and p_2, e is the last polynomial whose degree exceeds half that of p_1. □

Let us consider the matrix R computed at line 4 of HGCD. Presumably, R is

$$\prod_{j=1}^{l(\lceil m/2 \rceil)} \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix},$$

where $q_j(x)$ is the jth quotient of the remainder sequence for b_0 and b_1. That is, $R = R^{(b_0,b_1)}_{0,l(\lceil m/2 \rceil)}$. Yet on line 5, we used R as if it were the matrix $R^{(a_0,a_1)}_{0,l(\lceil 3m/2 \rceil)}$ to obtain d and e, where d is the last term in the remainder sequence of degree greater than $3m/2$. We must show that both these interpretations of R are correct. That is, $R^{(a_0,a_1)}_{0,l(\lceil 3m/2 \rceil)} = R^{(b_0,b_1)}_{0,l(\lceil m/2 \rceil)}$. Similarly, we must show that S, computed on line 9, may play the role assigned to it. That is,

$$S = R^{(g_0,g_1)}_{0,l(\lceil m/2 \rceil)} = R^{(e,f)}_{0,l(m)}.$$

These results are implied by the next lemmas.

Lemma 8.6. Let

$$f(x) = f_1(x)x^k + f_2(x), \tag{8.25}$$

† This computation could be done right after line 5, of course.

where $\mathrm{DEG}(f_2) < k$, and let

$$g(x) = g_1(x)x^k + g_2(x), \tag{8.26}$$

where $\mathrm{DEG}(g_2) < k$. Let q and q_1 be the quotients $\lfloor f(x)/g(x) \rfloor$ and $\lfloor f_1(x)/g_1(x) \rfloor$, and let r and r_1 be the remainders $f(x) - q(x)g(x)$ and $f_1(x) - q_1(x)g_1(x)$, respectively. If $\mathrm{DEG}(f) > \mathrm{DEG}(g)$ and $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ [i.e., $\mathrm{DEG}(g_1) \geq \frac{1}{2}\mathrm{DEG}(f_1)$], then

a) $q(x) = q_1(x)$ and
b) $r(x)$ and $r_1(x)x^k$ agree in all terms of degree $k + \mathrm{DEG}(f) - \mathrm{DEG}(g)$ or higher.

Proof. Consider dividing $f(x)$ by $g(x)$ using the ordinary division algorithm which divides the first term of $f(x)$ by the first term of $g(x)$ to get the first term of the quotient. The first term of the quotient is multiplied by $g(x)$ and subtracted from $f(x)$ and so on. The first $\mathrm{DEG}(g) - k$ terms produced do not depend on $g_2(x)$. But the quotient has only terms of degree $\mathrm{DEG}(f) - \mathrm{DEG}(g)$. Thus if $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(g) - k$, that is, $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$, the quotient does not depend on $g_2(x)$. If $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(f) - k$, then the quotient does not depend on $f_2(x)$. But $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(f) - k$ follows from $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ and $\mathrm{DEG}(f) > \mathrm{DEG}(g)$. Thus part (a) follows. For part (b), similar reasoning shows that the remainder terms of degree $\mathrm{DEG}(f) - (\mathrm{DEG}(g) - k)$ or greater do not depend on $g_2(x)$. Similarly, terms of the remainder of degree k or greater do not depend on $f_2(x)$. But $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k > k$. Thus $r(x)$ and $r_1(x)x^k$ agree in all terms of degree $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k$ or higher. □

Lemma 8.7. Let $f(x) = f_1(x)x^k + f_2(x)$ and $g(x) = g_1(x)x^k + g_2(x)$, where $\mathrm{DEG}(f_2) < k$ and $\mathrm{DEG}(g_2) < k$. Let $\mathrm{DEG}(f) = n$, and $\mathrm{DEG}(g) < \mathrm{DEG}(f)$. Then

$$R^{(f,g)}_{0,I(\lfloor (n+k)/2 \rfloor)} = R^{(f_1,g_1)}_{0,I(\lfloor (n-k)/2 \rfloor)}.$$

That is, the quotients of the remainder sequences for (f, g) and (f_1, g_1) agree at least until the latter reaches a remainder whose degree is no more than half that of f_1.

Proof. Lemma 8.6 assures that the quotients agree, and that in the corresponding remainders of the two remainder sequences a sufficient number of the high-order terms agree. □

Theorem 8.17. Let $a_0(x)$ and $a_1(x)$ be polynomials, with $\mathrm{DEG}(a_0) = n$ and $\mathrm{DEG}(a_1) < n$. Then $\mathrm{HGCD}(a_0, a_1) = R_{0,I(n/2)}$.

Proof. The result is a straightforward induction on n, using Lemma 8.7 to insure that R at line 4 is $R^{(a_0,a_1)}_{0,I(\lceil 3m/2 \rceil)}$ and that S at line 9 is $R^{(a_0,a_1)}_{I(\lceil 3m/2 \rceil)+1,I(m)}$. □

Theorem 8.18. HGCD requires $O_A(M(n) \log n)$ time if its arguments are of degree at most n, where $M(n)$ is the time required to multiply two polynomials of degree n.

Proof. We show the result for n a power of 4. Since the time for HGCD is clearly a nondecreasing function, the theorem then follows for all n. If $\mathrm{DEG}(a_0)$ is a power of 4, then

$$\mathrm{DEG}(b_0) = \tfrac{1}{2}\mathrm{DEG}(a_0) \quad \text{and} \quad \mathrm{DEG}(g_0) \leq \tfrac{1}{2}\mathrm{DEG}(a_0).$$

Thus $T(n)$, the time for HGCD with input of degree n, is bounded by

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cM(n) \tag{8.27}$$

for some constant c. That is, the body of HGCD involves two calls to itself with half-sized arguments and a constant number of other operations which are either $O_A(n)$ or $O_A(M(n))$ in time complexity. The solution to (8.27) should be familiar; it is bounded from above by $c_1 M(n) \log n$ for some constant c_1. □

Now we proceed to the complete algorithm for greatest common divisors. It uses HGCD to calculate $R_{0,n/2}$, then $R_{0,3n/4}$, then $R_{0,7n/8}$, and so on, where n is the degree of the input.

Algorithm 8.7. GCD algorithm.

Input. Polynomials $p_1(x)$ and $p_2(x)$, where $\mathrm{DEG}(p_2) < \mathrm{DEG}(p_1)$.

Output. $\mathrm{GCD}(p_1, p_2)$, the greatest common divisor of p_1 and p_2.

Method. We call the procedure $\mathrm{GCD}(p_1, p_2)$, where GCD is the recursive procedure of Fig. 8.8. □

Example 8.12. Let us continue Example 8.11 (p. 303). There, $p_1(x) = x^5 + x^4 + x^3 + x^2 + 1$ and $p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7$. We already found

$$\mathrm{HGCD}(p_1, p_2) = \begin{bmatrix} 1 & -(x+3) \\ -(\frac{1}{4}x - \frac{1}{16}) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Thus we compute $b_0 = 4x^3 - 7x^2 + 11x + 22$ and $b_1 = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}$ at line 3. We find that b_1 does not divide b_0. At line 5, we find

$$b_0 \text{ modulo } b_1 = 3952x + 3952.$$

Since the latter divides $-\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}$, the call to GCD at line 6 terminates at line 1 and produces $3952x + 3952$ as an answer. Of course, $x + 1$ is also a greatest common divisor of p_1 and p_2. □

```
      procedure GCD(a_0, a_1):
1.    if a_1 divides a_0 then return a_1
      else
          begin
2.            R ← HGCD(a_0, a_1);
3.            [b_0; b_1] ← R * [a_0; a_1];
4.            if b_1 divides b_0 then return b_1
              else
                  begin
5.                    c ← b_0 modulo b_1;
6.                    return GCD(b_1, c)
                  end
          end
```

Fig. 8.8. Procedure GCD.

The correctness of Algorithm 8.7 is trivial if we can show that it terminates. Thus the correctness of the algorithm is implied by its timing analysis, the result of the next theorem.

Theorem 8.19. If $\mathrm{DEG}(p_1) = n$, then Algorithm 8.7 requires $O_A(M(n) \log n)$ time, where $M(n)$ is the time needed to multiply two polynomials of degree n.

Proof. The inequality

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1 M(n) + c_2 M(n) \log n, \tag{8.28}$$

where c_1 and c_2 are constants, describes the running time of Algorithm 8.7. That is, the degree of b_1 is less than half that of a_0, so the first term of (8.28) accounts for the recursive call on line 6. The term $c_1 M(n)$ accounts for the divisions and multiplications on lines 1, 3, 4, and 5, and the last term accounts for the call to HGCD on line 2. The solution to (8.28) is easily seen to be bounded from above by $kM(n) \log n$ for a constant k. □

Corollary. The GCD of two polynomials of degree at most n can be computed in $O_A(n \log^2 n)$ time.

8.10 INTEGER GCD'S

We shall now briefly discuss the modifications to the procedures HGCD and GCD to make them work for integers. To understand where problems arise, let us look at Lemma 8.6, which showed that when taking quotients of

polynomials of degrees n and $n - d$, we have no need for terms of degree less than $n - 2d$ in either.

In analogy with Lemma 8.6, we may consider two integers f and g, with $f > g$, and write $f = f_1 2^k + f_2$ and $g = g_1 2^k + g_2$, where $f_2 < 2^k$ and $g_2 < 2^k$. In place of the condition $\mathrm{DEG}(g_1(x)) \geq \frac{1}{2}\mathrm{DEG}(f_1(x))$, we may assume $f_1 \leq (g_1)^2$. Then, we may let $f = qg + r$ and $f_1 = q_1 g_1 + r_1$. Combining these formulas, we obtain

$$g_1 2^k (q - q_1) = f_2 + r_1 2^k - q g_2 - r. \tag{8.29}$$

Since $r_1 < g_1$, $f_2 < 2^k$, and all integers are nonnegative, we may easily show from (8.29) that $q - q_1 \leq 0$. Let $q = q_1 - m$, for some $m \geq 0$. Then from (8.29) we may conclude

$$m g_1 2^k \leq q g_2 + r = (q_1 - m) g_2 + r.$$

Hence

$$mg = m g_1 2^k + m g_2 \leq q_1 g_2 + r. \tag{8.30}$$

Since $f_1 \leq (g_1)^2$, we have $q_1 \leq g_1$. Also, $g_2 < 2^k$ and $r < g$ are known, so (8.30) implies:

$$mg < g_1 2^k + g \leq 2g. \tag{8.31}$$

Now $m < 2$ follows immediately from (8.31). We conclude that either $q = q_1$ or $q = q_1 - 1$.

In the former case, there is no problem. If $q = q_1 - 1$, on the other hand, we cannot expect HGCD to work properly. Fortunately, we can show that $q_1 \neq q$ only if the quotient is the last one in the remainder sequence whose matrix is produced by HGCD. That is, if we substitute $q - q_1 = -1$ into (8.29), we have

$$r = f_2 + r_1 2^k - q g_2 + g_1 2^k. \tag{8.32}$$

Since $r < g = g_1 2^k + g_2$, we conclude from (8.32) that $r_1 2^k + f_2 < g_2(1 + q)$, or surely $r_1 < 1 + q$, that is, $r_1 < q_1$.

Thus r_1, which is the term in the remainder sequence following f_1 and g_1, must be less than f_1/g_1. Since $g_1 \geq \sqrt{f_1}$, we have $r_1 < \sqrt{f_1}$, meaning that HGCD would return a matrix involving quotients up to f_1/g_1 but no further. If this matrix is used in line 5 of HGCD (p. 304), it is possible that f computed on line 6 will not be less than $a_0^{3/4}$. However, since there was an error of only one in the last quotient, it is possible to show that extending the remainder sequence a limited amount (independent of the size of a_0) after line 6 is sufficient to bring the sequence below $a_0^{3/4}$. A similar "fixup" is needed for GCD after line 5 of that procedure.

Another source of problems concerns Lemma 8.6. With polynomials, we were able to show that the remainder sequence formed by considering

leading terms of the polynomials agreed in certain high-order terms with the sequence formed from the complete polynomials. The analogous results hold approximately for integers, provided we have $q = q_1$. However, we can only limit the difference between r and $r_1 2^k$ to within $2^k(q + 1)$; we cannot be sure that any particular bits of r and $r_1 2^k$ will agree. Nevertheless, this source of "rounding error" will cause only a limited number of additional terms of the remainder sequence to be needed after line 6 of HGCD and line 5 of GCD.

Since the additional cost of extending the remainder sequence a bounded amount is $O_B(M(n))$ if $M(n)$ is the time to multiply n-bit numbers, the timing analyses of HGCD and GCD are essentially unaffected. We therefore have the following theorem.

Theorem 8.20. If $M(n)$ is the time needed to multiply two n-bit numbers, then there is an algorithm to find $\mathrm{GCD}(a_0, a_1)$ for integers a_0 and a_1 in $O_B(M(n) \log n)$ time.

Proof. Exercise based on the above suggested modification to procedures HGCD and GCD. □

Corollary. We can find integer GCD's in $O_B(n \log^2 n \operatorname{loglog} n)$ time. □

8.11 CHINESE REMAINDERING REVISITED

As promised, we shall now see how the GCD algorithm can be used to develop an asymptotically fast algorithm for the integer case of Chinese remaindering without preconditioning. Recall that the preconditioned Algorithm 8.5 requires $O_B(M(bk) \log k)$ time for reconstructing u from k moduli of b bits each. The problem is to compute $d_i = (p/p_i)^{-1}$ modulo p_i, where $p_0, p_1, \ldots, p_{k-1}$ are the moduli and p their product.

Using the obvious divide-and-conquer algorithm, we can compute p itself by first computing products of pairs of p_i's, then products of four p_i's, etc., in $O_B(M(bk) \log k)$ time. The techniques of Algorithm 8.5 enable us to compute $e_i = (p/p_i)$ modulo p_i in $O_B(M(bk) \log k)$ steps, for $0 \le i < k$, without preconditioning. It remains to determine the time necessary to calculate $d_i = e_i^{-1}$ modulo p_i.

Since p/p_i is the product of the moduli other than p_i, it must be relatively prime to p_i. If we express p/p_i as $qp_i + e_i$ for some integer q, it follows that e_i and p_i are relatively prime, that is, $\mathrm{GCD}(e_i, p_i) = 1$. Thus, given x and y such that $e_i x + p_i y = 1$, we have $e_i x \equiv 1$ modulo p_i. It follows that $x \equiv e_i^{-1} = d_i$ modulo p_i. But the extended Euclidean algorithm computes such an x and y.

While procedure GCD was designed to produce only $\mathrm{GCD}(p_1, p_2)$, we designed HGCD to produce the matrix $R_{0,I(n/2)}$. Thus a simple modification to the GCD algorithm could allow it to produce $R_{0,I(0)}$. It is then possible to obtain x, since it will be the upper left element of this matrix. It is now possible to state the timing result for Chinese remaindering without preconditioning.

Theorem 8.21. Given k moduli of b bits each, in the integer case Chinese remaindering is $O_B(M(bk) \log k) + O_B(kM(b) \log b)$.

Proof. By the above analysis, the first term accounts for computing the e_i's and executing Algorithm 8.5. The second term accounts for computing the d_i's, since the computation of x and y may be done modulo p_i, permitting b-bit arithmetic throughout. □

Corollary. Chinese remaindering without preconditioning requires time at most $O_B[bk \log^2 bk \operatorname{loglog} bk]$.†

8.12 SPARSE POLYNOMIALS

We have been using a representation of univariate polynomials which assumes that the polynomial $\sum_{i=0}^{n-1} a_i x^i$ is *dense*, that is, almost all of the coefficients are nonzero. For many applications, it is useful to assume that the polynomial is *sparse*, that is, the number of nonzero coefficients is much less than the largest degree. In this situation, the logical representation of a polynomial is the list of pairs (a_i, j_i) consisting of a nonzero coefficient and its corresponding power of x.

While we cannot delve into all the techniques known for doing arithmetic on sparse polynomials, we shall mention two interesting aspects of the theory. First, it is unreasonable to use Fourier transforms to do multiplication; thus we shall give one reasonable algorithm to multiply sparse polynomials. Second, we shall exhibit a surprising difference between the way arithmetic should be done on dense and on sparse polynomials by considering the computation of $[p(x)]^4$.

The most reasonable known strategy for handling sparse polynomials when multiplications are being done is to represent a polynomial $\sum_{i=1}^{n} a_i x^{j_i}$ as the list of pairs $(a_1, j_1), (a_2, j_2), \ldots, (a_n, j_n)$, where we assume the j's are distinct and in decreasing order, i.e., $j_i > j_{i+1}$ for $1 \le i < n$. To multiply two polynomials represented in this way, we compute products of pairs and sort the resulting terms by their exponents (second components of the pairs), combining any terms with identical exponents. The penalty for not doing so is that our representations may have increasingly many terms with the same exponent. Thus, as more and more arithmetic is done, the cost begins to significantly exceed what it could have been if we had combined terms at each step.

† It is interesting to note that for $M(n) = n \log n \operatorname{loglog} n$, this figure is the best that can be obtained from Theorem 8.21, no matter how b and k are related.

If we multiply sorted sparse polynomials, we can take advantage of the fact that they are sorted and the fact that one may have many more terms than the other to make the sorting of the product as simple as possible. We present an informal algorithm to do multiplication of sparse polynomials in this way.

Algorithm 8.8. Multiplication of sorted sparse polynomials.

Input. Polynomials

$$f(x)=\sum_{i=1}^{m} a_i x^{j_i} \quad \text{and} \quad g(x)=\sum_{i=1}^{n} b_i x^{k_i},$$

represented by lists of pairs

$$(a_1, j_1), (a_2, j_2), \ldots, (a_m, j_m) \quad \text{and} \quad (b_1, k_1), (b_2, k_2), \ldots, (b_n, k_n),$$

where the j's and k's are monotonically decreasing.

Output

$$\sum_{i=1}^{p} c_i x^{l_i} = f(x)g(x),$$

represented by a list of pairs where the l_i's are monotonically decreasing.

Method. Without loss of generality, assume $m \geq n$.

1. Construct the sequences S_i, for $1 \leq i \leq n$, whose rth term, $1 \leq r \leq m$, is $(a_r b_i, j_r + k_i)$. That is, S_i represents the product of $f(x)$ with the ith term of $g(x)$.
2. Merge S_{2i-1} with S_{2i} for $1 \leq i \leq n/2$, combining terms. Then merge the resulting sequences in pairs, combining terms, and repeat until one sorted sequence remains. □

Theorem 8.22. Algorithm 8.8 requires $O(mn \log n)$ time,† where $m \geq n$ is assumed.

Proof. Step 1 is $O(mn)$, surely. Step 2 must be repeated $\lceil \log n \rceil$ times, and the total work at each pass is clearly $O(mn)$. □

Now let us see how Algorithm 8.8 and its time complexity affects the way sparse polynomial arithmetic should be done.

Example 8.13. Consider the computation of $p^4(x)$, where $p(x)$ is a polynomial with n terms in both the dense and sparse cases. Given that p is dense, it is easy to show that the best way to compute $p^4(x)$ is by two squarings. That is,

† Note we are using complexity on a RAM rather than arithmetic complexity here, since branches are inherent in the program for Algorithm 8.8, even if m and n are fixed.

assume $M(n)$, the time to multiply dense polynomials, is $cn \log n$. Then we can compute $p^2(x)$ in $cn \log n$ steps and square the result in $2cn \log 2n$ steps, for a total cost of $3cn \log n + 2cn$ steps. In comparison, if we compute $p^4 = p \times (p \times (p \times p))$, the time required is easily seen to be $6cn \log n$, on the assumption that the multiplication $p \times p^2$ requires $2M(n)$ steps and the multiplication $p \times p^3$ requires $3M(n)$ steps. Thus, for dense polynomials, we make the expected observation that p^4 should be calculated by repeated squaring.

Now suppose instead that $p(x)$ is a sparse polynomial with n terms. If we compute $(p^2)^2$ using Algorithm 8.8, the time required is $cn^2 \log n$ for the first squaring and, assuming that few terms can be combined, $cn^4 \log n^2$ for the second, a total of $c(2n^4 + n^2) \log n$. On the other hand, computation of $p \times (p \times (p \times p))$ requires $cn^2 \log n + cn^3 \log n + cn^4 \log n = c(n^4 + n^3 + n^2) \log n$. This figure is less than the time to square twice. Thus repeated squaring of sparse polynomials is not always a good way to compute p^4. The effect becomes more pronounced if we consider computation of p^{2^t} for large integers t. □

EXERCISES

8.1 Use Algorithm 8.1 to find the "reciprocal" of 429.

8.2 Use Algorithm 8.2 to find 429^2.

8.3 Use Algorithm 8.3 to find the "reciprocal" of

$$x^7 - x^6 + x^5 - 3x^4 + x^3 - x^2 + 2x + 1.$$

*8.4 Give an algorithm analogous to Algorithm 8.2 to compute squares of polynomials.

8.5 Use your algorithm from Exercise 8.4 to compute $(x^3 - x^2 + x - 2)^2$.

8.6 Use Algorithm 8.4 to find the representation for one million when the moduli are 2, 3, 5, 7, 11, 13, 17, 19.

8.7 Write a complete algorithm to find residues of a polynomial modulo a collection of polynomial moduli.

8.8 Find the residues of $x^7 + 3x^6 + x^5 + 3x^4 + x^2 + 1$ modulo $x + 3$, $x^3 - 3x + 1$, $x^2 + x - 2$, and $x^2 - 1$.

8.9 The polynomials in Exercise 8.8 were carefully selected to make hand computation feasible. Select at random four polynomials of degrees 1, 2, 3, and 4 and find the residues of $x^9 - 4$ with respect to the four polynomials. What happens?

8.10 Let 5, 6, 7, 11 be four moduli. Find that u less than their product such that $u \leftrightarrow (1, 2, 3, 4)$.

*8.11 Generalize Lemma 8.3 to apply to arbitrary polynomial moduli whose roots are known or can be found (e.g., polynomials of degree less than 5). What is the

complexity of polynomial Chinese remaindering by use of your algorithm, as a function of the number and degree of the moduli?

8.12 Find a polynomial whose values at 0, 1, 2, 3 are, respectively, 1, 1, 2, 2.

8.13 Find the greatest common divisor of

$$x^6 + 3x^5 + 3x^4 + x^3 - x^2 - x - 1,$$
$$x^6 + 2x^5 + x^4 + 2x^3 + 2x^2 + x + 1.$$

8.14 The polynomials in Exercise 8.13 were carefully selected to make hand computation feasible. Select two arbitrary polynomials of degrees 7 and 8 respectively and attempt to find their GCD. What happens? What do you suspect the GCD would turn out to be?

*8.15 Give a complete algorithm to find integer GCD's that runs in $O_B(n \log^2 n \log\log n)$ time on n-bit integers.

8.16 Use your algorithm from Exercise 8.15 to find GCD(377,233).

*8.17 Suppose $p(x)$ is a sparse nth-degree polynomial. Determine, as a function of n, the best method of evaluating $p^8(x)$ if multiplication is by Algorithm 8.8.

**8.18 The following method of computing $f(x)/g(x)$ for polynomials f and g, on the assumption that g divides f, is proposed. Let f and g be of $(n-1)$st degree or less.

(1) Compute F and G, the discrete Fourier transforms of f and g, respectively.
(2) Divide the terms of F by corresponding terms in G to yield sequence H.
(3) Take the inverse transform of H. The result is f/g. Will this algorithm work?

**8.19 Let $M(n)$ be the time to multiply n-bit numbers and $Q(n)$ the time to find $\lfloor\sqrt{i}\rfloor$ for an n-bit integer i. Assume $M(an) \geq aM(n)$ for $a \geq 1$ and similarly for $Q(n)$. Show that $M(n)$ and $Q(n)$ are within a constant factor of one another.

*8.20 Generalize Exercise 8.19 to (a) polynomials, (b) rth roots for fixed r.

**8.21 Give an $O_A(n \log n)$ algorithm to evaluate an $(n-1)$st-degree polynomial and all its derivatives at one point.

**8.22 A dense polynomial in r variables† can be represented as

$$\sum_{i_1=0}^{n-1} \sum_{i_2=0}^{n-1} \cdots \sum_{i_r=0}^{n-1} a_{i_1 i_2 \cdots i_r} x_1^{i_1} x_2^{i_2} \cdots x_r^{i_r}.$$

Show that by evaluating and interpolating these polynomials at points $x_1 = \omega^{j_1}$, $x_2 = \omega^{j_2}, \ldots, x_r = \omega^{j_r}$, for $0 \leq j_k < 2n$, where ω is the principal $2n$th root of unity, we can multiply such polynomials in $O_A(n^r \log n)$.

**8.23 Show that under reasonable assumptions about the smoothness of $M(n)$ and $D(n)$, the times needed to multiply and divide dense polynomials in r variables, $M(n)$ and $D(n)$, are within a constant factor of one another.

**8.24 Find an $O_B(n \log^2 n \log\log n)$ algorithm to convert (a) n-bit binary numbers to decimal; (b) n-place decimal number to binary.

† As r gets large, the dense assumption becomes progressively less useful.

**8.25 The *least common multiple* (*LCM*) of integers (polynomials) x and y is that z (some z in the polynomial case) divisible by x and y which divides any other integer (polynomial) also divisible by x and y. Show that the time to find the LCM of two n-bit numbers is at least as great as the time to multiply n-bit numbers.

8.26 Does the method of Section 2.6 for integer multiplication yield an $O_A(n^{1.59})$ time polynomial multiplication algorithm?

**8.27 Show that we may evaluate an $(n-1)$st-degree polynomial at the points $a^0, a^1, \ldots, a^{n-1}$ in $O_A(n \log n)$ steps. [*Hint:* Show that $c_j = \sum_{i=0}^{n-1} b_i a^{ij}$ can be written as $c_j = \sum_{i=0}^{n-1} f(i)g(j-i)$ for some functions f and g.]

**8.28 Show that we may evaluate an $(n-1)$st-degree polynomial at the n points $ba^{2j} + ca^j + d$, for $0 \leq j < n$, in $O_A(n \log n)$ steps.

8.29 Give recursive versions of Algorithms 8.4 and 8.5.

Research Problems

8.30 In this chapter, we have shown a number of polynomial and integer problems to be

i) essentially as complex as multiplication, or
ii) at most a log factor more complex than multiplication.

Some of these problems appear in the exercises of this chapter. Exercise 9.9 gives another problem in group (ii)—the problem of "and-or" multiplication.

A reasonable research problem is to add to the set of problems in either class (i) or (ii). Another area is to show that some of the problems in group (ii) must necessarily be of the same complexity. For example, one might conjecture that this is true for GCD's and LCM's.

8.31 Another problem which appears deceptively simple is to determine whether Algorithm 8.8 is the best that can be done to multiply sorted sparse polynomials and sort the result. Johnson [1974] has considered some aspects of this problem.

8.32 The value of n for which many of the algorithms described here become practical is enormous. However, we could carefully combine them with the $O(n^{1.59})$ methods mentioned in Section 2.6 and Exercise 8.26 for some small n's and the obvious $O(n^2)$ method for the smallest values of n. Obtain in this way upper bounds on the values of n for which the problems of this chapter have better algorithms than the obvious ones. Some work along these lines has been done by Kung [1973].

BIBLIOGRAPHIC NOTES

Theorem 8.2, the fact that finding reciprocals is no harder than multiplication, is from Cook and Aanderaa [1969]. Curiously, the fact that the algorithm has a polynomial analog was not realized for several years. Moenck and Borodin [1972] gave an $O_B(n \log^2 n \log\log n)$ algorithm for division, and shortly thereafter the $O_B(n \log n \log\log n)$ division algorithm was observed independently by several people, including Sieveking [1972].

The development of algorithms for modular arithmetic, and for interpolation and evaluation of polynomials follows the lines of Moenck and Borodin [1972]. An $O_B(n \log^2 n \log\log n)$ preconditioned algorithm for Chinese remaindering was found by Heindel and Horowitz [1971]. Borodin and Munro [1971] gave an $O(n^{1.91})$ multipoint polynomial evaluation algorithm, and Horowitz [1972] extended this to interpolation. Kung [1973] developed an $O_B(n \log^2 n)$ polynomial evaluation algorithm without using a fast $(n \log n)$ division algorithm. The unity of Chinese remaindering, interpolation, the evaluation of polynomials, and the computation of integer residues is expressed in Lipson [1971].

The $O_B(n \log^2 n \log\log n)$ integer GCD algorithm is by Schönhage [1971]. It was adapted to polynomials and general Euclidean domains by Moenck [1973].

A survey of classical techniques for GCD's can be found in Knuth [1969]. A sampling of the material on the complexity of sparse polynomial arithmetic can be found in Brown [1971], Gentleman [1972] and Gentleman and Johnson [1973]. Additional results on computer implementation of polynomial arithmetic can be found in Hall [1971], Brown [1973], and Collins [1973]. Algorithm 8.8 with a heap data structure has been implemented by S. Johnson in ALTRAN [Brown, 1973].

Exercises 8.19 and 8.20 are due to R. Karp and J. Ullman. Exercise 8.21 on the evaluation of a polynomial and its derivatives was observed by Vari [1974] and Aho, Steiglitz, and Ullman [1974], independently. Exercise 8.28 is from the latter. Exercise 8.27 is due to Bluestein [1970] and Rabiner, Schafer, and Rader [1969]. An expanded treatment of polynomial and integer arithmetic is found in Borodin and Munro [1975].