17

Polynomial arithmetic and applications

In this chapter, we study algorithms for performing arithmetic on polynomials. Initially, we shall adopt a very general point of view, discussing polynomials whose coefficients lie in an arbitrary ring R, and then specialize to the case where the coefficient ring is a field F.

There are many similarities between arithmetic in $\mathbb{Z}$ and in $R[X]$, and the similarities between $\mathbb{Z}$ and $F[X]$ run even deeper. Many of the algorithms we discuss in this chapter are quite similar to the corresponding algorithms for integers.

As we did in Chapter 14 for matrices, we shall treat R as an "abstract data type," and measure the complexity of algorithms for polynomials over a ring R by counting "operations in R."

17.1 Basic arithmetic

Throughout this section, R denotes a non-trivial ring.

For computational purposes, we shall assume that a polynomial $g = \sum_{i=0}^{k-1} a_i X^i \in R[X]$ is represented as a coefficient vector $(a_0, a_1, \ldots, a_{k-1})$. Further, when g is non-zero, the coefficient a_{k-1} should be non-zero.

The basic algorithms for addition, subtraction, multiplication, and division of polynomials are quite straightforward adaptations of the corresponding algorithms for integers. In fact, because of the lack of "carries," these algorithms are actually much simpler in the polynomial case. We briefly discuss these algorithms here—analogous to our treatment of integer arithmetic, we do not discuss the details of "stripping" leading zero coefficients.

For addition and subtraction, all we need to do is to add or subtract coefficient vectors.

For multiplication, let $g = \sum_{i=0}^{k-1} a_i X^i \in R[X]$ and $h = \sum_{i=0}^{\ell-1} b_i X^i \in R[X]$, where $k \geq 1$ and $\ell \geq 1$. The product $f := g \cdot h$ is of the form $f = \sum_{i=0}^{k+\ell-2} c_i X^i$, the coefficients of which can be computed using $O(k\ell)$ operations in R as follows:

```
for $i \leftarrow 0$ to $k+\ell-2$ do $c_i \leftarrow 0$
for $i \leftarrow 0$ to $k-1$ do
        for $j \leftarrow 0$ to $\ell-1$ do
                $c_{i+j} \leftarrow c_{i+j} + a_i \cdot b_j$
```

For division, let $g = \sum_{i=0}^{k-1} a_i X^i \in R[X]$ and $h = \sum_{i=0}^{\ell-1} b_i X^i \in R[X]$, where $b_{\ell-1} \in R^*$. We want to compute polynomials $q, r \in R[X]$ such that $g = hq + r$, where $\deg(r) < \ell - 1$. If $k < \ell$, we can simply set $q \leftarrow 0$ and $r \leftarrow g$; otherwise, we can compute q and r using $O(\ell \cdot (k - \ell + 1))$ operations in R using the following algorithm:

```
$t \leftarrow b_{\ell-1}^{-1} \in R$
for $i \leftarrow 0$ to $k-1$ do $r_i \leftarrow a_i$
for $i \leftarrow k-\ell$ down to 0 do
        $q_i \leftarrow t \cdot r_{i+\ell-1}$
        for $j \leftarrow 0$ to $\ell-1$ do
                $r_{i+j} \leftarrow r_{i+j} - q_i \cdot b_j$
$q \leftarrow \sum_{i=0}^{k-\ell} q_i X^i,\ r \leftarrow \sum_{i=0}^{\ell-2} r_i X^i$
```

With these simple algorithms, we obtain the polynomial analog of Theorem 3.3. Let us define the **length** of $g \in R[X]$, denoted $\text{len}(g)$, to be the length of its coefficient vector; more precisely, we define

$$\text{len}(g) := \begin{cases} \deg(g) + 1 & \text{if } g \neq 0, \\ 1 & \text{if } g = 0. \end{cases}$$

Sometimes (but not always) it is clearer and more convenient to state the running times of algorithms in terms of the length, rather than the degree, of a polynomial (the latter has the inconvenient habit of taking on the value 0, or worse, $-\infty$).

Theorem 17.1. *Let g and h be arbitrary polynomials in $R[X]$.*

(i) We can compute $g \pm h$ with $O(\text{len}(g) + \text{len}(h))$ operations in R.

(ii) We can compute $g \cdot h$ with $O(\text{len}(g)\,\text{len}(h))$ operations in R.

(iii) If $\text{lc}(h) \in R^*$, we can compute $q, r \in R[X]$ such that $g = hq + r$ and $\deg(r) < \deg(h)$ with $O(\text{len}(h)\,\text{len}(q))$ operations in R.

Analogous to algorithms for modular integer arithmetic, we can also do arithmetic in the residue class ring $R[X]/(f)$, where $f \in R[X]$ is a polynomial with $\text{lc}(f) \in R^*$. For each $\alpha \in R[X]/(f)$, there exists a unique polynomial $g \in R[X]$ with $\deg(g) < \deg(f)$ and $\alpha = [g]_f$; we call this polynomial g the **canonical representative of** α, and denote it by $\text{rep}(\alpha)$. For computational purposes, we represent elements of $R[X]/(f)$ by their canonical representatives.

With this representation, addition and subtraction in $R[X]/(f)$ can be performed using $O(\text{len}(f))$ operations in R, while multiplication takes $O(\text{len}(f)^2)$ operations in R.

The repeated-squaring algorithm for computing powers works equally well in this setting: given $\alpha \in R[X]/(f)$ and a non-negative exponent e, we can compute α^e using $O(\text{len}(e))$ multiplications in $R[X]/(f)$, for a total of $O(\text{len}(e)\,\text{len}(f)^2)$ operations in R.

EXERCISE 17.1. State and re-work the polynomial analogs of Exercises 3.26–3.28.

EXERCISE 17.2. Given a polynomial $g \in R[X]$ and an element $x \in R$, a particularly elegant and efficient way of computing $g(x)$ is called **Horner's rule**. Suppose $g = \sum_{i=0}^{k-1} a_i X^i$, where $k \geq 0$ and $a_i \in R$ for $i = 0, \ldots, k-1$. Horner's rule computes $g(x)$ as follows:

```
$y \leftarrow 0_R$
for $i \leftarrow k-1$ down to 0 do
    $y \leftarrow yx + a_i$
output $y$
```

Show that this algorithm correctly computes $g(x)$ using k multiplications in R and k additions in R.

EXERCISE 17.3. Let $f \in R[X]$ be a polynomial of degree $\ell > 0$ with $\text{lc}(f) \in R^*$, and let $E := R[X]/(f)$. Suppose that in addition to f, we are given a polynomial $g \in R[X]$ of degree less than k and an element $\alpha \in E$, and we want to compute $g(\alpha) \in E$. This is called the **modular composition** problem.

(a) Show that a straightforward application of Horner's rule yields an algorithm that uses $O(k\ell^2)$ operations in R, and requires space for storing $O(\ell)$ elements of R.

(b) Show how to compute $g(\alpha)$ using just $O(k\ell + k^{1/2}\ell^2)$ operations in R, at the expense of requiring space for storing $O(k^{1/2}\ell)$ elements of R. Hint: first compute a table of powers $1, \alpha, \ldots, \alpha^m$, for $m \approx k^{1/2}$.

EXERCISE 17.4. Given polynomials $g, h \in R[X]$, show how to compute their composition $g(h) \in R[X]$ using $O(\text{len}(g)^2\,\text{len}(h)^2)$ operations in R.

EXERCISE 17.5. Suppose you are given three polynomials $f, g, h \in \mathbb{Z}_p[X]$, where p is a large prime, in particular, $p \geq 2\deg(g)\deg(h)$. Design an efficient probabilistic algorithm that tests if $f = g(h)$ (i.e., if f equals g composed with h). Your algorithm should have the following properties: if $f = g(h)$, it should always output "true," and otherwise, it should output "false" with probability at least 0.999. The expected running time of your algorithm should be $O((\text{len}(f) + \text{len}(g) + \text{len}(h))\,\text{len}(p)^2)$.

EXERCISE 17.6. Let $x, a_0, \ldots, a_{\ell-1} \in R$, and let k be an integer with $0 < k \leq \ell$. For $i = 0, \ldots, \ell - k$, define $g_i := \sum_{j=i}^{i+k-1} a_j X^j \in R[X]$. Show how to compute the $\ell - k + 1$ values $g_0(x), \ldots, g_{\ell-k}(x)$ using $O(\ell)$ operations in R.

17.2 Computing minimal polynomials in $F[X]/(f)$ (I)

In this section, we shall examine a computational problem to which we shall return on several occasions, as it will serve to illustrate a number of interesting algebraic and algorithmic concepts.

Let F be a field, and let $f \in F[X]$ be a monic polynomial of degree $\ell > 0$. Also, let $E := F[X]/(f)$, which is an F-algebra, and in particular, an F-vector space. As an F-vector space, E has dimension ℓ. Suppose we are given an element $\alpha \in E$, and want to efficiently compute the minimal polynomial of α over F—that is, the monic polynomial $\phi \in F[X]$ of least degree such that $\phi(\alpha) = 0$, which we know has degree at most ℓ (see §16.5).

We can solve this problem using polynomial arithmetic and Gaussian elimination, as follows. Consider the F-linear map $\rho : F[X]_{\leq \ell} \to E$ that sends a polynomial $g \in F[X]$ of degree at most ℓ to $g(\alpha)$. To perform the linear algebra, we need to specify bases for $F[X]_{\leq \ell}$ and E. For $F[X]_{\leq \ell}$, let us work with the basis $\mathcal{S} := \{X^{\ell+1-i}\}_{i=1}^{\ell+1}$. With this choice of basis, for $g = \sum_{i=0}^{\ell} a_i X^i \in F[X]_{\leq \ell}$, the coordinate vector of g is $\text{Vec}_{\mathcal{S}}(g) = (a_\ell, \ldots, a_0) \in F^{1\times(\ell+1)}$. For E, let us work with the basis $\mathcal{T} := \{\xi^{i-1}\}_{i=1}^{\ell}$, where $\xi := [X]_f \in E$. Let

$$A := \text{Mat}_{\mathcal{S},\mathcal{T}}(\rho) \in F^{(\ell+1)\times\ell};$$

that is, A is the matrix of ρ relative to $\mathcal{S}$ and $\mathcal{T}$ (see §14.2). For $i = 1, \ldots, \ell + 1$, the ith row of A is the coordinate vector $\text{Vec}_{\mathcal{T}}(\alpha^{\ell+1-i}) \in F^{1\times\ell}$.

We compute the matrix A by computing the powers $1, \alpha, \ldots, \alpha^\ell$, reading off the ith row of A directly from the canonical representative of the $\alpha^{\ell+1-i}$. We then apply Gaussian elimination to A to find row vectors $v_1, \ldots, v_s \in F^{1\times(\ell+1)}$ that are coordinate vectors corresponding to a basis for the kernel of ρ. Now, the coordinate vector of the minimal polynomial of α is a linear combination of $v_1, \ldots, v_s$. To find it, we form the $s \times (\ell + 1)$ matrix B whose rows consist of $v_1, \ldots, v_s$, and apply Gaussian elimination to B, obtaining an $s\times(\ell+1)$ matrix B' in reduced row echelon form whose row space is the same as that of B. Let ϕ be the polynomial whose coordinate vector is the last row of B'.

Because of the choice of basis for $F[X]_{\leq \ell}$, and because B' is in reduced row echelon form, it is clear that no non-zero polynomial in $\operatorname{Ker}\rho$ has degree less than that of ϕ. Moreover, as ϕ is already monic (again, by the fact that B' is in reduced row echelon form), it follows that ϕ is in fact the minimal polynomial of α over F.

The total amount of work performed by this algorithm is $O(\ell^3)$ operations in F to build the matrix A (this just amounts to computing ℓ successive powers of α, that is, $O(\ell)$ multiplications in E, each of which takes $O(\ell^2)$ operations in F), and $O(\ell^3)$ operations in F to perform both Gaussian elimination steps.

17.3 Euclid's algorithm

In this section, F denotes a field, and we consider the computation of greatest common divisors in $F[X]$.

The Euclidean algorithm for integers is easily adapted to compute $\gcd(g, h)$ for polynomials $g, h \in F[X]$. Analogous to the integer case, we assume that $\deg(g) \geq \deg(h)$; however, we shall also assume that $g \neq 0$. This is not a serious restriction, of course, as $\gcd(0, 0) = 0$, and making this restriction will simplify the presentation a bit. Recall that we defined $\gcd(g, h)$ to be either zero or monic, and the assumption that $g \neq 0$ means that $\gcd(g, h)$ is non-zero, and hence monic.

The following is the analog of Theorem 4.1, and is based on the division with remainder property for polynomials.

Theorem 17.2. *Let $g, h \in F[X]$, with $\deg(g) \geq \deg(h)$ and $g \neq 0$. Define the polynomials $r_0, r_1, \ldots, r_{\lambda+1} \in F[X]$ and $q_1, \ldots, q_\lambda \in F[X]$, where $\lambda \geq 0$, as follows:*

$$\begin{aligned}
g &= r_0, \\
h &= r_1, \\
r_0 &= r_1 q_1 + r_2 \quad (0 \leq \deg(r_2) < \deg(r_1)), \\
&\ \ \vdots \\
r_{i-1} &= r_i q_i + r_{i+1} \quad (0 \leq \deg(r_{i+1}) < \deg(r_i)), \\
&\ \ \vdots \\
r_{\lambda-2} &= r_{\lambda-1} q_{\lambda-1} + r_\lambda \quad (0 \leq \deg(r_\lambda) < \deg(r_{\lambda-1})), \\
r_{\lambda-1} &= r_\lambda q_\lambda \quad (r_{\lambda+1} = 0).
\end{aligned}$$

Note that by definition, $\lambda = 0$ if $h = 0$, and $\lambda > 0$ otherwise. Then we have $r_\lambda / \operatorname{lc}(r_\lambda) = \gcd(g, h)$, and if $h \neq 0$, then $\lambda \leq \deg(h) + 1$.

Proof. Arguing as in the proof of Theorem 4.1, one sees that

$$\gcd(g, h) = \gcd(r_0, r_1) = \cdots = \gcd(r_\lambda, r_{\lambda+1}) = \gcd(r_\lambda, 0) = r_\lambda / \operatorname{lc}(r_\lambda).$$

That proves the first statement.

For the second statement, if $h \neq 0$, then the degree sequence

$$\deg(r_1), \deg(r_2), \ldots, \deg(r_\lambda)$$

is strictly decreasing, with $\deg(r_\lambda) \geq 0$, from which it follows that $\deg(h) = \deg(r_1) \geq \lambda - 1$. □

This gives us the following polynomial version of the Euclidean algorithm:

Euclid's algorithm. On input g, h, where $g, h \in F[X]$ with $\deg(g) \geq \deg(h)$ and $g \neq 0$, compute $d = \gcd(g, h)$ as follows:

```
$r \leftarrow g,\ r' \leftarrow h$
while $r' \neq 0$ do
    $r'' \leftarrow r \bmod r'$
    $(r, r') \leftarrow (r', r'')$
$d \leftarrow r / \mathrm{lc}(r)$   // make monic
output $d$
```

Theorem 17.3. *Euclid's algorithm for polynomials performs $O(\mathrm{len}(g)\,\mathrm{len}(h))$ operations in F.*

Proof. The proof is almost identical to that of Theorem 4.2. Details are left to the reader. □

Just as for integers, if $d = \gcd(g, h)$, then $gF[X] + hF[X] = dF[X]$, and so there exist polynomials s and t such that $gs + ht = d$. The procedure for calculating s and t is precisely the same as in the integer case; however, in the polynomial case, we can be much more precise about the relative sizes of the objects involved in the calculation.

Theorem 17.4. *Let g, h, $r_0, \ldots, r_{\lambda+1}$ and $q_1, \ldots, q_\lambda$ be as in Theorem 17.2. Define polynomials $s_0, \ldots, s_{\lambda+1} \in F[X]$ and $t_0, \ldots, t_{\lambda+1} \in F[X]$ as follows:*

$$s_0 := 1, \quad t_0 := 0,$$
$$s_1 := 0, \quad t_1 := 1,$$

and for $i = 1, \ldots, \lambda$,

$$s_{i+1} := s_{i-1} - s_i q_i, \quad t_{i+1} := t_{i-1} - t_i q_i.$$

Then:

(i) for $i = 0, \ldots, \lambda + 1$, we have $gs_i + ht_i = r_i$; in particular, $gs_\lambda + ht_\lambda = \mathrm{lc}(r_\lambda)\gcd(g, h)$;

(ii) for $i = 0, \ldots, \lambda$, we have $s_i t_{i+1} - t_i s_{i+1} = (-1)^i$;

(iii) *for* $i = 0, \dots, \lambda + 1$, *we have* $\gcd(s_i, t_i) = 1$;

(iv) *for* $i = 1, \dots, \lambda + 1$, *we have*

$$\deg(t_i) = \deg(g) - \deg(r_{i-1}),$$

and for $i = 2, \dots, \lambda + 1$, *we have*

$$\deg(s_i) = \deg(h) - \deg(r_{i-1});$$

(v) *for* $i = 1, \dots, \lambda + 1$, *we have* $\deg(t_i) \leq \deg(g)$ *and* $\deg(s_i) \leq \deg(h)$; *if* $\deg(g) > 0$ *and* $h \neq 0$, *then* $\deg(t_\lambda) < \deg(g)$ *and* $\deg(s_\lambda) < \deg(h)$.

Proof. (i), (ii), and (iii) are proved just as in the corresponding parts of Theorem 4.3.

For (iv), the proof will hinge on the following facts:

- For $i = 1, \dots, \lambda$, we have $\deg(r_{i-1}) \geq \deg(r_i)$, and since q_i is the quotient in dividing r_{i-1} by r_i, we have $\deg(q_i) = \deg(r_{i-1}) - \deg(r_i)$.
- For $i = 2, \dots, \lambda$, we have $\deg(r_{i-1}) > \deg(r_i)$.

We prove the statement involving the t_i's by induction on i, and leave the proof of the statement involving the s_i's to the reader.

One can see by inspection that this statement holds for $i = 1$, since $\deg(t_1) = 0$ and $r_0 = g$. If $\lambda = 0$, there is nothing more to prove, so assume that $\lambda > 0$ and $h \neq 0$.

Now, for $i = 2$, we have $t_2 = 0 - 1 \cdot q_1 = -q_1$. Thus, $\deg(t_2) = \deg(q_1) = \deg(r_0) - \deg(r_1) = \deg(g) - \deg(r_1)$.

Now for the induction step. Assume $i \geq 3$. Then we have

$$\begin{aligned}
\deg(t_{i-1}q_{i-1}) &= \deg(t_{i-1}) + \deg(q_{i-1}) \\
&= \deg(g) - \deg(r_{i-2}) + \deg(q_{i-1}) \quad \text{(by induction)} \\
&= \deg(g) - \deg(r_{i-1}) \\
&\qquad \text{(since } \deg(q_{i-1}) = \deg(r_{i-2}) - \deg(r_{i-1})\text{)} \\
&> \deg(g) - \deg(r_{i-3}) \quad \text{(since } \deg(r_{i-3}) > \deg(r_{i-1})\text{)} \\
&= \deg(t_{i-2}) \quad \text{(by induction).}
\end{aligned}$$

By definition, $t_i = t_{i-2} - t_{i-1}q_{i-1}$, and from the above reasoning, we see that

$$\deg(g) - \deg(r_{i-1}) = \deg(t_{i-1}q_{i-1}) > \deg(t_{i-2}),$$

from which it follows that $\deg(t_i) = \deg(g) - \deg(r_{i-1})$.

(v) follows easily from (iv). □

From this theorem, we obtain the following algorithm:

The extended Euclidean algorithm. On input g, h, where $g, h \in F[X]$ with $\deg(g) \geq \deg(h)$ and $g \neq 0$, compute d, s, and t, where $d, s, t \in F[X]$, $d = \gcd(g, h)$ and $gs + ht = d$, as follows:

```
$r \leftarrow g,\ r' \leftarrow h$
$s \leftarrow 1,\ s' \leftarrow 0$
$t \leftarrow 0,\ t' \leftarrow 1$
while $r' \neq 0$ do
    compute $q, r''$ such that $r = r'q + r''$, with $\deg(r'') < \deg(r')$
    $(r, s, t, r', s', t') \leftarrow (r', s', t', r'', s - s'q, t - t'q)$
$c \leftarrow \mathrm{lc}(r)$
$d \leftarrow r/c,\ s \leftarrow s/c,\ t \leftarrow t/c$   // make monic
output $d, s, t$
```

Theorem 17.5. *The extended Euclidean algorithm for polynomials performs $O(\mathrm{len}(g)\,\mathrm{len}(h))$ operations in F.*

Proof. Exercise. □

EXERCISE 17.7. State and re-work the polynomial analogs of Exercises 4.2, 4.3, 4.4, 4.5, and 4.8.

17.4 Computing modular inverses and Chinese remaindering

In this and the remaining sections of this chapter, we explore various applications of Euclid's algorithm for polynomials. Most of these applications are analogous to their integer counterparts, although there are some differences to watch for. Throughout this section, F denotes a field.

We begin with the obvious application of the extended Euclidean algorithm for polynomials to the problem of computing multiplicative inverses in $F[X]/(f)$.

Theorem 17.6. *Suppose we are given polynomials $f, h \in F[X]$, where $\deg(h) < \deg(f)$. Then using $O(\mathrm{len}(f)^2)$ operations in F, we can determine if h is relatively prime to f, and if so, compute $h^{-1} \bmod f$.*

Proof. We may assume $\deg(f) > 0$, since $\deg(f) = 0$ implies $h = 0 = h^{-1} \bmod f$. We run the extended Euclidean algorithm on input f, h, obtaining polynomials d, s, t such that $d = \gcd(f, h)$ and $fs + ht = d$. If $d \neq 1$, then h does not have a multiplicative inverse modulo f. Otherwise, if $d = 1$, then t is a multiplicative inverse of h modulo f. Moreover, by part (v) of Theorem 17.4, we have $\deg(t) < \deg(f)$, and so $t = h^{-1} \bmod f$. Based on Theorem 17.5, it is clear that all the computations can be performed using $O(\mathrm{len}(f)^2)$ operations in F. □

We also observe that the Chinese remainder theorem for polynomials (Theorem 16.19) can be made computationally effective as well:

Theorem 17.7 (Effective Chinese remainder theorem). *Suppose we are given polynomials $f_1, \ldots, f_k \in F[X]$ and $g_1, \ldots, g_k \in F[X]$, where the family $\{f_i\}_{i=1}^k$ is pairwise relatively prime, and where $\deg(f_i) > 0$ and $\deg(g_i) < \deg(f_i)$ for $i = 1, \ldots, k$. Let $f := \prod_{i=1}^k f_i$. Then using $O(\operatorname{len}(f)^2)$ operations in F, we can compute the unique polynomial $g \in F[X]$ satisfying $\deg(g) < \deg(f)$ and $g \equiv g_i \pmod{f_i}$ for $i = 1, \ldots, k$.*

Proof. Exercise (just use the formulas given after Theorem 16.19). □

Polynomial interpolation

We remind the reader of the discussion following Theorem 16.19, where the point was made that when $f_i = X - x_i$ and $g_i = y_i$, for $i = 1, \ldots, k$, then the Chinese remainder theorem for polynomials reduces to Lagrange interpolation. Thus, Theorem 17.7 says that given distinct elements $x_1, \ldots, x_k \in F$, along with elements $y_1, \ldots, y_k \in F$, we can compute the unique polynomial $g \in F[X]$ of degree less than k such that

$$g(x_i) = y_i \quad (i = 1, \ldots, k),$$

using $O(k^2)$ operations in F.

It is perhaps worth noting that we could also solve the polynomial interpolation problem using Gaussian elimination, by inverting the corresponding Vandermonde matrix (see Example 14.2). However, this algorithm would use $O(k^3)$ operations in F. This is a specific instance of a more general phenomenon: there are many computational problems involving polynomials over fields that can be solved using Gaussian elimination, but which can be solved more efficiently using more specialized algorithmic techniques.

Speeding up algorithms via modular computation

In §4.4, we discussed how the Chinese remainder theorem could be used to speed up certain types of computations involving integers. The example we gave was the multiplication of integer matrices. We can use the same idea to speed up certain types of computations involving polynomials. For example, if one wants to multiply two matrices whose entries are elements of $F[X]$, one can use the Chinese remainder theorem for polynomials to speed things up. This strategy is most easily implemented if F is sufficiently large, so that we can use polynomial evaluation and interpolation directly, and do not have to worry about constructing irreducible polynomials.

EXERCISE 17.8. Adapt the algorithms of Exercises 4.14 and 4.15 to obtain an algorithm for polynomial interpolation. This algorithm is called **Newton interpolation**.

17.5 Rational function reconstruction and applications

Throughout this section, F denotes a field.

We next state and prove the polynomial analog of Theorem 4.9. As we are now "reconstituting" a rational function, rather than a rational number, we call this procedure **rational function reconstruction**. Because of the relative simplicity of polynomials compared to integers, the rational reconstruction theorem for polynomials is a bit "sharper" than the rational reconstruction theorem for integers, and much simpler to prove.

To state the result precisely, let us introduce some notation. For polynomials $g, h \in F[X]$ with $\deg(g) \geq \deg(h)$ and $g \neq 0$, let us define

$$\mathrm{EEA}(g,h) := \big\{(r_i, s_i, t_i)\big\}_{i=0}^{\lambda+1},$$

where r_i, s_i, and t_i, for $i = 0, \ldots, \lambda + 1$, are defined as in Theorem 17.4.

Theorem 17.8 (Rational function reconstruction). *Let $f, h \in F[X]$ be polynomials, and let r^*, t^* be non-negative integers, such that*

$$\deg(h) < \deg(f) \text{ and } r^* + t^* \leq \deg(f).$$

Further, let $\mathrm{EEA}(f,h) = \{(r_i, s_i, t_i)\}_{i=0}^{\lambda+1}$, and let j be the smallest index (among $0, \ldots, \lambda + 1$) such that $\deg(r_j) < r^*$, and set

$$r' := r_j, \quad s' := s_j, \quad \text{and} \quad t' := t_j.$$

Finally, suppose that there exist polynomials $r, s, t \in F[X]$ such that

$$r = fs + ht, \quad \deg(r) < r^*, \quad \text{and} \quad 0 \leq \deg(t) \leq t^*.$$

Then for some non-zero polynomial $q \in F[X]$, we have

$$r = r'q, \quad s = s'q, \quad t = t'q.$$

Proof. Since $\deg(r_0) = \deg(f) \geq r^* > -\infty = \deg(r_{\lambda+1})$, the value of j is well defined, and moreover, $j \geq 1$, $\deg(r_{j-1}) \geq r^*$, and $t_j \neq 0$.

From the equalities $r_j = fs_j + ht_j$ and $r = fs + ht$, we have the two congruences:

$$r_j \equiv ht_j \pmod{f},$$
$$r \equiv ht \pmod{f}.$$

Subtracting t times the first from t_j times the second, we obtain

$$rt_j \equiv r_j t \pmod{f}.$$

This says that f divides $rt_j - r_j t$.

We want to show that, in fact, $rt_j - r_j t = 0$. To this end, first observe that by part (iv) of Theorem 17.4 and the inequality $\deg(r_{j-1}) \geq r^*$, we have

$$\deg(t_j) = \deg(f) - \deg(r_{j-1}) \leq \deg(f) - r^*.$$

Combining this with the inequality $\deg(r) < r^*$, we see that

$$\deg(rt_j) = \deg(r) + \deg(t_j) < \deg(f).$$

Furthermore, using the inequalities

$$\deg(r_j) < r^*, \quad \deg(t) \leq t^*, \quad \text{and} \quad r^* + t^* \leq \deg(f),$$

we see that

$$\deg(r_j t) = \deg(r_j) + \deg(t) < \deg(f),$$

and it immediately follows that

$$\deg(rt_j - r_j t) < \deg(f).$$

Since f divides $rt_j - r_j t$ and $\deg(rt_j - r_j t) < \deg(f)$, the only possibility is that

$$rt_j - r_j t = 0.$$

The rest of the proof follows exactly the same line of reasoning as in the last paragraph in the proof of Theorem 4.9, as the reader may easily verify. □

17.5.1 Application: recovering rational functions from their reversed Laurent series

We now discuss the polynomial analog of the application in §4.6.1. This is an entirely straightforward translation of the results in §4.6.1, but we shall see in the next chapter that this problem has its own interesting applications.

Suppose Alice knows a rational function $z = s/t \in F(X)$, where s and t are polynomials with $\deg(s) < \deg(t)$, and tells Bob some of the high-order coefficients of the reversed Laurent series (see §16.8) representing z in $F((X^{-1}))$. We shall show that if $\deg(t) \leq \ell$ and Bob is given the bound ℓ on $\deg(t)$, along with the high-order 2ℓ coefficients of z, then Bob can determine z, expressed as a rational function in lowest terms.

So suppose that $z = s/t = \sum_{i=1}^{\infty} z_i X^{-i}$, and that Alice tells Bob the coefficients $z_1, \ldots, z_{2\ell}$. Equivalently, Alice gives Bob the polynomial

$$h := z_1 X^{2\ell-1} + \cdots + z_{2\ell-1} X + z_{2\ell}.$$

Also, let us define $f := X^{2\ell}$. Here is Bob's algorithm for recovering z:

1. Run the extended Euclidean algorithm on input f, h to obtain $\mathrm{EEA}(f, h)$, and apply Theorem 17.8 with f, h, $r^* := \ell$, and $t^* := \ell$, to obtain the polynomials r', s', t'.
2. Output s', t'.

We claim that $z = -s'/t'$. To prove this, first observe that $h = \lfloor fz \rfloor = \lfloor fs/t \rfloor$ (see Theorem 16.32). So if we set $r := fs \bmod t$, then we have

$$r = fs - ht,\ \deg(r) < r^*,\ 0 \leq \deg(t) \leq t^*,\ \text{and } r^* + t^* \leq \deg(f).$$

It follows that the polynomials s', t' from Theorem 17.8 satisfy $s = s'q$ and $-t = t'q$ for some non-zero polynomial q, and thus, $s'/t' = -s/t$, which proves the claim.

We may further observe that since the extended Euclidean algorithm guarantees that $\gcd(s', t') = 1$, not only do we obtain z, but we obtain z expressed as a fraction in lowest terms.

It is clear that this algorithm takes $O(\ell^2)$ operations in F.

17.5.2 Application: polynomial interpolation with errors

We now discuss the polynomial analog of the application in §4.6.2.

If we "encode" a polynomial $g \in F[X]$, with $\deg(g) < k$, as the sequence $(y_1, \ldots, y_k) \in F^{\times k}$, where $y_i = g(x_i)$, then we can efficiently recover g from this encoding, using an algorithm for polynomial interpolation. Here, of course, the x_i's are distinct elements of F.

Now suppose that Alice encodes g as $(y_1, \ldots, y_k)$, and sends this encoding to Bob, but that some, say at most ℓ, of the y_i's may be corrupted during transmission. Let $(z_1, \ldots, z_k)$ denote the vector actually received by Bob.

Here is how we can use Theorem 17.8 to recover the original value of g from $(z_1, \ldots, z_k)$, assuming:

- the original polynomial g has degree less than m,
- at most ℓ errors occur in transmission, and
- $k \geq 2\ell + m$.

Let us set $f_i := X - x_i$ for $i = 1, \ldots, k$, and $f := f_1 \cdots f_k$. Now, suppose Bob obtains the corrupted encoding $(z_1, \ldots, z_k)$. Here is what Bob does to recover g:

1. Interpolate, obtaining a polynomial h, with $\deg(h) < k$ and $h(x_i) = z_i$ for $i = 1, \ldots, k$.
2. Run the extended Euclidean algorithm on input f, h to obtain $\mathrm{EEA}(f, h)$, and apply Theorem 17.8 with f, h, $r^* := m + \ell$ and $t^* := \ell$, to obtain the polynomials r', s', t'.
3. If $t' \mid r'$, output r'/t'; otherwise, output "error."

We claim that the above procedure outputs g, under the assumptions listed above. To see this, let t be the product of the f_i's for those values of i where an error occurred. Now, assuming at most ℓ errors occurred, we have $\deg(t) \leq \ell$. Also, let $r := gt$, and note that $\deg(r) < m + \ell$. We claim that

$$r \equiv ht \pmod{f}. \tag{17.1}$$

To show that (17.1) holds, it suffices to show that

$$gt \equiv ht \pmod{f_i} \tag{17.2}$$

for all $i = 1, \ldots, k$. To show this, consider first an index i at which no error occurred, so that $y_i = z_i$. Then $gt \equiv y_i t \pmod{f_i}$ and $ht \equiv z_i t \equiv y_i t \pmod{f_i}$, and so (17.2) holds for this i. Next, consider an index i for which an error occurred. Then by construction, $gt \equiv 0 \pmod{f_i}$ and $ht \equiv 0 \pmod{f_i}$, and so (17.2) holds for this i. Thus, (17.1) holds, from which it follows that the values r', t' obtained from Theorem 17.8 satisfy

$$\frac{r'}{t'} = \frac{r}{t} = \frac{gt}{t} = g.$$

One easily checks that both the procedures to encode and decode a value g run in time $O(k^2)$. The above scheme is an example of an **error correcting code** called a **Reed–Solomon code**.

17.5.3 Applications to symbolic algebra

Rational function reconstruction has applications in symbolic algebra, analogous to those discussed in §4.6.3. In that section, we discussed the application of solving systems of linear equations over the integers using rational reconstruction. In exactly the same way, one can use rational function reconstruction to solve systems of linear equations over $F[X]$—the solution to such a system of equations will be a vector whose entries are elements of $F(X)$, the field of rational functions.

EXERCISE 17.9. Consider again the secret sharing problem, as discussed in Example 8.28. There, we presented a scheme that distributes shares of a secret among several parties in such a way that no coalition of k or fewer parties can reconstruct the secret, while every coalition of $k+1$ parties can. Now suppose that some parties may be corrupt: in the protocol to reconstruct the secret, a corrupted party may contribute an incorrect share. Show how to modify the protocol in Example 8.28 so that if shares are distributed among several parties, then

(a) no coalition of k or fewer parties can reconstruct the secret, and

(b) if at most k parties are corrupt, then every coalition of $3k+1$ parties (which may include some of the corrupted parties) can correctly reconstruct the secret.

The following exercises are the polynomial analogs of Exercises 4.20, 4.22, and 4.23.

EXERCISE 17.10. Let F be a field. Show that given polynomials $s, t \in F[X]$ and integer k, with $\deg(s) < \deg(t)$ and $k > 0$, we can compute the kth coefficient in the reversed Laurent series representing s/t using $O(\text{len}(k)\,\text{len}(t)^2)$ operations in F.

EXERCISE 17.11. Let F be a field. Let $z \in F((X^{-1}))$ be a reversed Laurent series whose coefficient sequence is ultimately periodic. Show that $z \in F(X)$.

EXERCISE 17.12. Let F be a field. Let $z = s/t$, where $s, t \in F[X]$, $\deg(s) < \deg(t)$, and $\gcd(s, t) = 1$.

(a) Show that if F is finite, there exist integers k, k' such that $0 \leq k < k'$ and $sX^k \equiv sX^{k'} \pmod{t}$.

(b) Show that for integers k, k' with $0 \leq k < k'$, the sequence of coefficients of the reversed Laurent series representing z is $(k, k'-k)$-periodic if and only if $sX^k \equiv sX^{k'} \pmod{t}$.

(c) Show that if F is finite and $X \nmid t$, then the reversed Laurent series representing z is purely periodic with period equal to the multiplicative order of $[X]_t \in (F[X]/(t))^*$.

(d) More generally, show that if F is finite and $t = X^k t'$, with $X \nmid t'$, then the reversed Laurent series representing z is ultimately periodic with pre-period k and period equal to the multiplicative order of $[X]_{t'} \in (F[X]/(t'))^*$.

17.6 Faster polynomial arithmetic (∗)

The algorithms discussed in §3.5 for faster integer arithmetic are easily adapted to polynomials over a ring. Throughout this section, R denotes a non-trivial ring.

EXERCISE 17.13. State and re-work the analog of Exercise 3.41 for $R[X]$. Your algorithm should multiply two polynomials over R of length at most ℓ using $O(\ell^{\log_2 3})$ operations in R.

It is in fact possible to multiply polynomials over R of length at most ℓ using $O(\ell \operatorname{len}(\ell) \operatorname{len}(\operatorname{len}(\ell)))$ operations in R—we shall develop some of the ideas that lead to such a result below in Exercises 17.21–17.24 (see also the discussion in §17.7).

In Exercises 17.14–17.19 below, assume that we have an algorithm that multiplies two polynomials over R of length at most ℓ using at most $M(\ell)$ operations in R, where M is a well-behaved complexity function (as defined in §3.5).

EXERCISE 17.14. State and re-work the analog of Exercises 3.46 and 3.47 for $R[X]$.

EXERCISE 17.15. This problem is the analog of Exercise 3.48 for $R[X]$. Let us first define the notion of a "floating point" reversed Laurent series $\hat{z}$, which is represented as a pair (g, e), where $g \in R[X]$ and $e \in \mathbb{Z}$—the value of $\hat{z}$ is $gX^e \in R((X^{-1}))$, and we call $\operatorname{len}(g)$ the **precision** of $\hat{z}$. We say that $\hat{z}$ is a **length** k **approximation** of $z \in R((X^{-1}))$ if $\hat{z}$ has precision k and $\hat{z} = (1+\varepsilon)z$ for $\varepsilon \in R((X^{-1}))$ with $\deg(\varepsilon) \leq -k$, which is the same as saying that the high-order k coefficients of $\hat{z}$ and z are equal. Show that given $h \in R[X]$ with $\operatorname{lc}(h) \in R^*$, and positive integer k, we can compute a length k approximation of $1/h \in R((X^{-1}))$ using $O(M(k))$ operations in R. Hint: using Newton iteration, show how to go from a length t approximation of $1/h$ to a length $2t$ approximation, making use of just the high-order $2t$ coefficients of h, and using $O(M(t))$ operations in R.

EXERCISE 17.16. State and re-work the analog of Exercise 3.49 for $R[X]$.

EXERCISE 17.17. State and re-work the analog of Exercise 3.50 for $R[X]$. Conclude that a polynomial of length at most k can be evaluated at k points using $O(M(k)\operatorname{len}(k))$ operations in R.

EXERCISE 17.18. State and re-work the analog of Exercise 3.52 for $R[X]$, assuming $2_R \in R^*$.

The next two exercises develop a useful technique known as **Kronecker substitution**.

EXERCISE 17.19. Let $g, h \in R[X, Y]$ with $g = \sum_{i=0}^{m-1} g_i Y^i$ and $h = \sum_{i=0}^{m-1} h_i Y^i$, where each g_i and h_i is a polynomial in X of degree less than k. The product $f := gh \in R[X, Y]$ may be written $f = \sum_{i=0}^{2m-2} f_i Y^i$, where each f_i is a polynomial in X. Show how to compute f, given g and h, using $O(M(km))$ operations in R. Hint: for an appropriately chosen integer $t > 0$, first convert g, h to $\tilde{g}, \tilde{h} \in R[X]$, where $\tilde{g} := \sum_{i=0}^{m-1} g_i X^{ti}$ and $\tilde{h} := \sum_{i=0}^{m-1} h_i X^{ti}$; next, compute $\tilde{f} := \tilde{g}\tilde{h} \in R[X]$; finally, "read off" the f_i's from the coefficients of $\tilde{f}$.

EXERCISE 17.20. Assume that integers of length at most ℓ can be multiplied in time $\overline{M}(\ell)$, where $\overline{M}$ is a well-behaved complexity function. Let $g, h \in \mathbb{Z}[X]$ with $g = \sum_{i=0}^{m-1} a_i X^i$ and $h = \sum_{i=0}^{m-1} b_i X^i$, where each a_i and b_i is a non-negative integer, strictly less than 2^k. The product $f := gh \in \mathbb{Z}[X]$ may be written $f = \sum_{i=0}^{2m-2} c_i X^i$, where each c_i is a non-negative integer. Show how to compute f, given g and h, using $O(\overline{M}((k + \text{len}(m))m))$ operations in R. Hint: for an appropriately chosen integer $t > 0$, first convert g, h to $a, b \in \mathbb{Z}$, where $a := \sum_{i=0}^{m-1} a_i 2^{ti}$ and $b := \sum_{i=0}^{m-1} b_i 2^{ti}$; next, compute $c := ab \in \mathbb{Z}$; finally, "read off" the c_i's from the bits of c.

The following exercises develop an important algorithm for multiplying polynomials in almost-linear time. For an integer $n \geq 0$, let us call $\omega \in R$ a **primitive 2^nth root of unity** if $n \geq 1$ and $\omega^{2^{n-1}} = -1_R$, or $n = 0$ and $\omega = 1_R$; if $2_R \neq 0_R$, then in particular, ω has multiplicative order 2^n. For $n \geq 0$, and $\omega \in R$ a primitive 2^nth root of unity, let us define the R-linear map $\mathcal{E}_{n,\omega} : R^{\times 2^n} \to R^{\times 2^n}$ that sends the vector $(a_0, \ldots, a_{2^n-1})$ to the vector $(g(1_R), g(\omega), \ldots, g(\omega^{2^n-1}))$, where $g := \sum_{i=0}^{2^n-1} a_i X^i \in R[X]$.

EXERCISE 17.21. Suppose $2_R \in R^*$ and $\omega \in R$ is a primitive 2^nth root of unity.

(a) Let k be any integer, and consider $\gcd(k, 2^n)$, which must be of the form 2^m for some $m = 0, \ldots, n$. Show that ω^k is a primitive 2^{n-m}th root of unity.

(b) Show that if $n \geq 1$, then $\omega - 1_R \in R^*$.

(c) Show that $\omega^k - 1_R \in R^*$ for all integers $k \not\equiv 0 \pmod{2^n}$.

(d) Show that for every integer k, we have

$$\sum_{i=0}^{2^n-1} \omega^{ki} = \begin{cases} 2_R^n & \text{if } k \equiv 0 \pmod{2^n}, \\ 0_R & \text{if } k \not\equiv 0 \pmod{2^n}. \end{cases}$$

(e) Let M_2 be the 2-multiplication map on $R^{\times 2^n}$, which is a bijective, R-linear map. Show that

$$\mathcal{E}_{n,\omega} \circ \mathcal{E}_{n,\omega^{-1}} = M_2^n = \mathcal{E}_{n,\omega^{-1}} \circ \mathcal{E}_{n,\omega},$$

and conclude that $\mathcal{E}_{n,\omega}$ is bijective, with $M_2^{-n} \circ \mathcal{E}_{n,\omega^{-1}}$ being its inverse. Hint: write down the matrices representing the maps $\mathcal{E}_{n,\omega}$ and $\mathcal{E}_{n,\omega^{-1}}$.

EXERCISE 17.22. This exercise develops a fast algorithm, called the **fast Fourier transform** or **FFT**, for computing the function $\mathcal{E}_{n,\omega}$. This is a recursive algorithm

$\mathrm{FFT}(n, \omega; a_0, \ldots, a_{2^n-1})$ that takes as input an integer $n \geq 0$, a primitive 2^nth root of unity $\omega \in R$, and elements $a_0, \ldots, a_{2^n-1} \in R$, and runs as follows:

```
if $n = 0$ then
    return $a_0$
else
    $(\alpha_0, \ldots, \alpha_{2^{n-1}-1}) \leftarrow \mathrm{FFT}(n-1, \omega^2; a_0, a_2, \ldots, a_{2^n-2})$
    $(\beta_0, \ldots, \beta_{2^{n-1}-1}) \leftarrow \mathrm{FFT}(n-1, \omega^2; a_1, a_3, \ldots, a_{2^n-1})$
    for $i \leftarrow 0$ to $2^{n-1} - 1$ do
        $\gamma_i \leftarrow \alpha_i + \beta_i\omega^i,\ \gamma_{i+2^{n-1}} \leftarrow \alpha_i - \beta_i\omega^i$
    return $(\gamma_0, \ldots, \gamma_{2^n-1})$
```

Show that this algorithm correctly computes $\mathcal{E}_{n,\omega}(a_0, \ldots, a_{2^n-1})$ using $O(2^n n)$ operations in R.

EXERCISE 17.23. Assume $2_R \in R^*$. Suppose that we are given two polynomials $g, h \in R[X]$ of length at most ℓ, along with a primitive 2^nth root of unity $\omega \in R$, where $2\ell \leq 2^n < 4\ell$. Let us "pad" g and h, writing $g = \sum_{i=0}^{2^n-1} a_i X^i$ and $h = \sum_{i=0}^{2^n-1} b_i X^i$, where a_i and b_i are zero for $i \geq \ell$. Show that the following algorithm correctly computes the product of g and h using $O(\ell \operatorname{len}(\ell))$ operations in R:

```
$(\alpha_0, \ldots, \alpha_{2^n-1}) \leftarrow \mathrm{FFT}(n, \omega; a_0, \ldots, a_{2^n-1})$
$(\beta_0, \ldots, \beta_{2^n-1}) \leftarrow \mathrm{FFT}(n, \omega; b_0, \ldots, b_{2^n-1})$
$(\gamma_0, \ldots, \gamma_{2^n-1}) \leftarrow (\alpha_0\beta_0, \ldots, \alpha_{2^n-1}\beta_{2^n-1})$
$(c_0, \ldots, c_{2^n-1}) \leftarrow 2_R^{-n}\,\mathrm{FFT}(n, \omega^{-1}; \gamma_0, \ldots, \gamma_{2^n-1})$
output $\sum_{i=0}^{2\ell-2} c_i X^i$
```

Also, argue more carefully that the algorithm performs $O(\ell \operatorname{len}(\ell))$ additions and subtractions in R, $O(\ell \operatorname{len}(\ell))$ multiplications in R by powers of ω, and $O(\ell)$ other multiplications in R.

EXERCISE 17.24. Assume $2_R \in R^*$. In this exercise, we use the FFT to develop an algorithm that multiplies polynomials over R of length at most ℓ using $O(\ell \operatorname{len}(\ell)^\beta)$ operations in R, where β is a constant. Unlike the previous exercise, we do not assume that R contains any particular primitive roots of unity; rather, the algorithm will create them "out of thin air." Suppose that $g, h \in R[X]$ are of length at most ℓ. Set $k := \lfloor\sqrt{\ell/2}\rfloor$, $m := \lceil \ell/k \rceil$. We may write $g = \sum_{i=0}^{m-1} g_i X^{ki}$ and $h = \sum_{i=0}^{m-1} h_i X^{ki}$, where the g_i's and h_i's are polynomials of length at most k. Let n be the integer determined by $2m \leq 2^n < 4m$. Let $q := X^{2^{n-1}} + 1_R \in R[X]$, $E := R[X]/(q)$, and $\omega := [X]_q \in E$.

(a) Show that ω is a primitive 2^nth root of unity in E, and that given an element

$\zeta \in E$ and an integer i between 0 and $2^n - 1$, we can compute $\zeta\omega^i \in E$ using $O(\ell^{1/2})$ operations in R.

(b) Let $\bar{g} := \sum_{i=0}^{m-1}[g_i]_q Y^i \in E[Y]$ and $\bar{h} := \sum_{i=0}^{m-1}[h_i]_q Y^i \in E[Y]$. Using the FFT (over E), show how to compute $\bar{f} := \bar{g}\bar{h} \in E[Y]$ by computing $O(\ell^{1/2})$ products in $R[X]$ of polynomials of length $O(\ell^{1/2})$, along with $O(\ell \operatorname{len}(\ell))$ additional operations in R.

(c) Show how to compute the coefficients of $f := gh \in R[X]$ from the value $\bar{f} \in E[Y]$ computed in part (b), using $O(\ell)$ operations in R.

(d) Based on parts (a)–(c), we obtain a recursive multiplication algorithm: on inputs of length at most ℓ, it performs at most $\alpha_0\ell \operatorname{len}(\ell)$ operations in R, and calls itself recursively on at most $\alpha_1\ell^{1/2}$ subproblems, each of length at most $\alpha_2\ell^{1/2}$; here, α_0, α_1 and α_2 are constants. If we just perform one level of recursion, and immediately switch to a quadratic multiplication algorithm, we obtain an algorithm whose operation count is $O(\ell^{1.5})$. If we perform two levels of recursion, this is reduced to $O(\ell^{1.25})$. For practical purposes, this is probably enough; however, to get an asymptotically better complexity bound, we can let the algorithm recurse all the way down to inputs of some (appropriately chosen) constant length. Show that if we do this, the operation count of the recursive algorithm is $O(\ell \operatorname{len}(\ell)^\beta)$ for some constant β (whose value depends on α_1 and α_2).

The approach used in the previous exercise was a bit sloppy. With a bit more care, one can use the same ideas to get an algorithm that multiplies polynomials over R of length at most ℓ using $O(\ell \operatorname{len}(\ell) \operatorname{len}(\operatorname{len}(\ell)))$ operations in R, assuming $2_R \in R^*$. The next exercise applies similar ideas, but with a few twists, to the problem of *integer* multiplication.

EXERCISE 17.25. This exercise uses the FFT to develop a linear-time algorithm for integer multiplication; however, a rigorous analysis depends on an unproven conjecture (which follows from a generalization of the Riemann hypothesis). Suppose we want to multiply two positive integers a and b, each of length at most ℓ (represented internally using the data structure described in §3.3). Throughout this exercise, assume that all computations are done on a RAM, and that arithmetic on integers of length $O(\operatorname{len}(\ell))$ takes time $O(1)$. Let k be an integer parameter with $k = \Theta(\operatorname{len}(\ell))$, and let $m := \lceil \ell/k \rceil$. We may write $a = \sum_{i=0}^{m-1} a_i 2^{ki}$ and $b = \sum_{i=0}^{m-1} b_i 2^{ki}$, where $0 \le a_i < 2^k$ and $0 \le b_i < 2^k$. Let n be the integer determined by $2m \le 2^n < 4m$.

(a) Assuming Conjecture 5.22, and assuming a deterministic, polynomial-time primality test (such as the one to be presented in Chapter 21), show how to efficiently generate a prime $p \equiv 1 \pmod{2^n}$ and an element $\omega \in \mathbb{Z}_p^*$ of multiplicative order 2^n, such that

$$2^{2k}m < p \leq \ell^{O(1)}.$$

Your algorithm should be probabilistic, and run in expected time polynomial in $\operatorname{len}(\ell)$.

(b) Assuming you have computed p and ω as in part (a), let $g := \sum_{i=0}^{m-1}[a_i]_pX^i \in \mathbb{Z}_p[X]$ and $h := \sum_{i=0}^{m-1}[b_i]_pX^i \in \mathbb{Z}_p[X]$, and show how to compute $f := gh \in \mathbb{Z}_p[X]$ in time $O(\ell)$ using the FFT (over $\mathbb{Z}_p$). Here, you may store elements of $\mathbb{Z}_p$ in single memory cells, so that operations in $\mathbb{Z}_p$ take time $O(1)$.

(c) Assuming you have computed $f \in \mathbb{Z}_p[X]$ as in part (b), show how to obtain $c := ab$ in time $O(\ell)$.

(d) Conclude that assuming Conjecture 5.22, we can multiply two integers of length at most ℓ on a RAM in time $O(\ell)$.

Note that even if one objects to our accounting practices, and insists on charging $O(\operatorname{len}(\ell)^2)$ time units for arithmetic on numbers of length $O(\operatorname{len}(\ell))$, the algorithm in the previous exercise runs in time $O(\ell \operatorname{len}(\ell)^2)$, which is "almost" linear time.

EXERCISE 17.26. Continuing with the previous exercise:

(a) Show how the algorithm presented there can be implemented on a RAM that has only built-in addition, subtraction, and branching instructions, but no multiplication or division instructions, and still run in time $O(\ell)$. Also, memory cells should store numbers of length at most $\operatorname{len}(\ell) + O(1)$. Hint: represent elements of $\mathbb{Z}_p$ as sequences of base-2^t digits, where $t \approx \alpha \operatorname{len}(\ell)$ for some constant $\alpha < 1$; use table lookup to multiply t-bit numbers, and to perform $2t$-by-t-bit divisions—for α sufficiently small, you can build these tables in time $o(\ell)$.

(b) Using Theorem 5.23, show how to make this algorithm fully deterministic and rigorous, assuming that on inputs of length ℓ, it is provided with a certain bit string σ_ℓ of length $O(\operatorname{len}(\ell))$ (this is called a *non-uniform* algorithm).

EXERCISE 17.27. This exercise shows how the algorithm in Exercise 17.25 can be made quite concrete, and fairly practical, as well.

(a) The number $p := 2^{59}27 + 1$ is a 64-bit prime. Show how to use this value of p in conjunction with the algorithm in Exercise 17.25 with $k = 20$ and any value of ℓ up to 2^{27}.

(b) The numbers $p_1 := 2^{30}3 + 1$, $p_2 := 2^{28}13 + 1$, and $p_3 := 2^{27}29 + 1$ are 32-bit primes. Show how to use the Chinese remainder theorem to modify the algorithm in Exercise 17.25, so that it uses the three primes p_1, p_2, p_3, and so that it works with $k = 32$ and any value of ℓ up to 2^{31}. This variant may be quite practical on a 32-bit machine with built-in instructions for 32-bit multiplication and 64-by-32-bit division.

The previous three exercises indicate that we can multiply integers in essentially linear time, both in theory and in practice. As mentioned in §3.6, there is a different, fully deterministic and rigorously analyzed algorithm that multiplies integers in linear time on a RAM. In fact, that algorithm works on a very restricted type of machine called a "pointer machine," which can be simulated in "real time" on a RAM with a very restricted instruction set (including the type in the previous exercise). That algorithm works with finite approximations to complex roots of unity, rather than roots of unity in a finite field.

We close this section with a cute application of fast polynomial multiplication to the problem of factoring integers.

EXERCISE 17.28. Let n be a large, positive integer. We can factor n using trial division in time $n^{1/2+o(1)}$; however, using fast polynomial arithmetic in $\mathbb{Z}_n[X]$, one can get a simple, deterministic, and rigorous algorithm that factors n in time $n^{1/4+o(1)}$. Note that all of the factoring algorithms discussed in Chapter 15, while faster, are either probabilistic, or deterministic but heuristic. Assume that we can multiply polynomials in $\mathbb{Z}_n[X]$ of length at most ℓ using $M(\ell)$ operations in $\mathbb{Z}_n$, where M is a well-behaved complexity function, and $M(\ell) = \ell^{1+o(1)}$ (the algorithm from Exercise 17.24 would suffice).

(a) Let ℓ be a positive integer, and for $i = 1, \ldots, \ell$, let

$$a_i := \prod_{j=0}^{\ell-1} (i\ell - j) \bmod n.$$

Using fast polynomial arithmetic, show how to compute $(a_1, \ldots, a_\ell)$ in time $\ell^{1+o(1)} \operatorname{len}(n)^{O(1)}$.

(b) Using the result of part (a), show how to factor n in time $n^{1/4+o(1)}$ using a deterministic algorithm.

17.7 Notes

Reed–Solomon codes were first proposed by Reed and Solomon [81], although the decoder presented here was developed later. Theorem 17.8 was proved by Mills [68]. The Reed–Solomon code is just one way of detecting and correcting errors—we have barely scratched the surface of this subject.

Just as in the case of integer arithmetic, the basic "pencil and paper" quadratic-time algorithms discussed in this chapter for polynomial arithmetic are not the best possible. The fastest known algorithms for multiplication of polynomials of length at most ℓ over a ring R take $O(\ell \operatorname{len}(\ell) \operatorname{len}(\operatorname{len}(\ell)))$ operations in R. These algorithms are all variations on the basic FFT algorithm (see Exercise 17.23), but work without assuming that $2_R \in R^*$ or that R contains any particular primitive roots of unity (we developed some of the ideas in Exercise 17.24). The Euclidean and extended Euclidean algorithms for polynomials over a field F can be implemented so as to take $O(\ell \operatorname{len}(\ell)^2 \operatorname{len}(\operatorname{len}(\ell)))$ operations in F, as can the algorithms for Chinese remaindering and rational function reconstruction. See the book by von zur Gathen and Gerhard [39] for details (as well for an analysis of the Euclidean algorithm for polynomials over the field of rational numbers and over function fields). Depending on the setting and many implementation details, such asymptotically fast algorithms for multiplication and division can be significantly faster than the quadratic-time algorithms, even for quite moderately sized inputs of practical interest. However, the fast Euclidean algorithms are only useful for significantly larger inputs.

Exercise 17.3 is based on an algorithm of Brent and Kung [20]. Using fast matrix and polynomial arithmetic, Brent and Kung show how to solve the modular composition problem using $O(\ell^{(\omega+1)/2})$ operations in R, where ω is the exponent for matrix multiplication (see §14.6), and so $(\omega+1)/2 < 1.7$. Modular composition arises as a subproblem in a number of algorithms.†

† Very recently, faster algorithms for modular composition have been discovered. See the papers by C. Umans [Fast polynomial factorization and modular composition in small characteristic, to appear in *40th Annual ACM Symposium on Theory of Computing*, 2008] and K. Kedlaya and C. Umans [Fast modular composition in any characteristic, manuscript, April 2008], both of which are available at `www.cs.caltech.edu/~umans/research`.