18
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 $R[\mathtt{X}]$, and the similarities between $\mathbb{Z}$ and $F[\mathtt{X}]$ 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 15 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."

18.1 Basic arithmetic

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

For computational purposes, we assume that a polynomial $a = \sum_{i=0}^{k-1} a_i\mathtt{X}^i \in R[\mathtt{X}]$ is represented as a coefficient vector $(a_0, a_1, \ldots, a_{k-1})$. Further, when a 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 $a = \sum_{i=0}^{k-1} a_i\mathtt{X}^i \in R[\mathtt{X}]$ and $b = \sum_{i=0}^{\ell-1} b_i\mathtt{X}^i \in R[\mathtt{X}]$, where $k \geq 1$ and $\ell \geq 1$. The product $c := a \cdot b$ is of the form $c = \sum_{i=0}^{k+\ell-2} c_i \mathtt{X}^i$, and 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 $a = \sum_{i=0}^{k-1} a_i \mathtt{X}^i \in R[\mathtt{X}]$ and $b = \sum_{i=0}^{\ell-1} b_i \mathtt{X}^i \in R[\mathtt{X}]$, where $b_{\ell-1} \in R^*$. We want to compute polynomials $q, r \in R[\mathtt{X}]$ such that $a = bq + r$, where $\deg(r) < \ell - 1$. If $k < \ell$, we can simply set $q \leftarrow 0$ and $r \leftarrow a$; 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
        $r_{i+j} \leftarrow r_{i+j} - q_i \cdot b_j$
$q \leftarrow \sum_{i=0}^{k-\ell} q_i \mathtt{X}^i$,  $r \leftarrow \sum_{i=0}^{\ell-2} r_i \mathtt{X}^i$
```

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

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

It is sometimes more convenient to state the running times of algorithms in terms of $\mathrm{len}(a)$, rather than $\deg(a)$ (the latter has the inconvenient habit of taking on the value 0, or worse, $-\infty$).

Theorem 18.1. *Let a and b be arbitrary polynomials in $R[\mathtt{X}]$.*

(i) We can compute $a \pm b$ with $O(\mathrm{len}(a) + \mathrm{len}(b))$ operations in R.

(ii) We can compute $a \cdot b$ with $O(\mathrm{len}(a)\,\mathrm{len}(b))$ operations in R.

(iii) If $b \neq 0$ and $\mathrm{lc}(b)$ is a unit in R, we can compute $q, r \in R[\mathtt{X}]$ such that $a = bq + r$ and $\deg(r) < \deg(b)$ with $O(\mathrm{len}(b)\,\mathrm{len}(q))$ operations in R.

Analogous to algorithms for modular integer arithmetic, we can also do arithmetic in the residue class ring $R[\mathtt{X}]/(n)$, where $n \in R[\mathtt{X}]$ is a polynomial of degree $\ell > 0$ whose leading coefficient $\mathrm{lc}(n)$ is a unit (in most applications, we may in fact assume that n is monic). For $\alpha \in R[\mathtt{X}]/(n)$, there exists a unique polynomial $a \in R[\mathtt{X}]$ with $\deg(a) < \ell$ and $\alpha = [a]_n$; we call this polynomial a the **canonical representative of** α, and denote it by $\mathrm{rep}(\alpha)$. For computational purposes, we represent elements of $R[\mathtt{X}]/(n)$ by their canonical representatives.

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

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

The following exercises deal with arithmetic with polynomials $R[\mathtt{X}]$ over a ring R.

EXERCISE 18.1. State and re-work the polynomial analog of Exercise 3.22.

EXERCISE 18.2. State and re-work the polynomial analog of Exercise 3.23. Assume $n_1, \ldots, n_k$ are monic polynomials.

EXERCISE 18.3. Given a polynomial $g \in R[\mathtt{X}]$ and an element $\alpha \in E$, where R is a subring of E, we may wish to compute $g(\alpha) \in E$. A particularly elegant and efficient way of doing this is called **Horner's rule**. Suppose $g = \sum_{i=0}^{k-1} g_i \mathtt{X}^i$, where $k \geq 0$ and $g_i \in R$ for $i = 0, \ldots, k-1$. Horner's rule computes $g(\alpha)$ as follows:

```
$\beta \leftarrow 0_E$
for $i \leftarrow k-1$ down to 0 do
    $\beta \leftarrow \beta \cdot \alpha + a_i$
output $\beta$
```

Show that this algorithm correctly computes $g(\alpha)$ using k multiplications in E and k additions in E.

EXERCISE 18.4. Let $f \in R[\mathtt{X}]$ be a monic polynomial of degree $\ell > 0$, and let $E := R[\mathtt{X}]/(f)$. Suppose that in addition to f, we are given a polynomial $g \in R[\mathtt{X}]$ of degree less than k and an element $\alpha \in E$, and we want to compute $g(\alpha) \in E$.

(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 18.5. Given polynomials $g, h \in R[\mathtt{X}]$, show how to compute the composition $g(h) \in R[\mathtt{X}]$ using $O(\mathrm{len}(g)^2 \,\mathrm{len}(h)^2)$ operations in R.

Exercise 18.6. Suppose you are given three polynomials $f, g, h \in \mathbb{Z}_p[\mathtt{X}]$, 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 $O((\mathrm{len}(f) + \mathrm{len}(g) + \mathrm{len}(h))\,\mathrm{len}(p)^2)$.

18.2 Computing minimal polynomials in $F[\mathtt{X}]/(f)$ (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, $f \in F[\mathtt{X}]$ a monic polynomial of degree $\ell > 0$, and let $E := F[\mathtt{X}]/(f)$. E is an F-algebra, and in particular, an F-vector space. As an F-vector space, it 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 $\phi \in F[\mathtt{X}]$ of least degree such that $\phi(\alpha) = 0$, which we know has degree at most ℓ (see §17.5).

We can solve this problem using polynomial arithmetic and Gaussian elimination, as follows. Consider the F-linear map $\rho : F[\mathtt{X}]_{\leq \ell} \to E$ that sends a polynomial $h \in F[\mathtt{X}]$ of degree at most ℓ to $h(\alpha)$. Let us fix ordered bases for $F[\mathtt{X}]_{\leq \ell}$ and E: for $F[\mathtt{X}]_{\leq \ell}$, let us take $\mathtt{X}^\ell, \mathtt{X}^{\ell-1}, \ldots, 1$, and for E, let us take $1, \eta, \ldots, \eta^{\ell-1}$, where $\eta := [\mathtt{X}]_f \in E$. The matrix A representing the map ρ (via multiplication on the right by A), is the $(\ell+1) \times \ell$ matrix A whose ith row, for $i = 1, \ldots, \ell+1$, is the coordinate vector of $\alpha^{\ell+1-i}$.

We apply Gaussian elimination to A to find a set of row vectors $v_1, \ldots, v_s$ that are coordinate vectors for 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 g be the polynomial whose coordinate vector is the last row of B'. Because of the choice of ordered basis for $F[\mathtt{X}]_{\leq \ell}$, and because B' is in reduced row echelon form, it is clear that no non-zero polynomial in $\ker(\rho)$ has degree less than that of g. Moreover, as g is already monic (again, by the fact that B' is in reduced row echelon form), it follows that g 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.

18.3 Euclid's algorithm

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

The basic Euclidean algorithm for integers is easily adapted to compute $\gcd(a, b)$, for polynomials $a, b \in F[\mathtt{X}]$. Analogous to the integer case, we assume that $\deg(a) \geq \deg(b)$; however, we shall also assume that $a \neq 0$. 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 $\gcd(a, b)$ to be either zero or monic, and the assumption that $a \neq 0$ means that $\gcd(a, b)$ is non-zero, and hence monic.

The following is the analog of Theorem 4.1.

Theorem 18.2. *Let $a, b \in F[\mathtt{X}]$, with $\deg(a) \geq \deg(b)$ and $a \neq 0$. Define the polynomials $r_0, r_1, \ldots, r_{\ell+1} \in F[\mathtt{X}]$, and $q_1, \ldots, q_\ell \in F[\mathtt{X}]$, where $\ell \geq 0$, as follows:*

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

Note that by definition, $\ell = 0$ if $b = 0$, and $\ell > 0$ otherwise; moreover, $r_\ell \neq 0$.

Then we have $r_\ell / \operatorname{lc}(r_\ell) = \gcd(a, b)$, and if $b \neq 0$, then $\ell \leq \deg(b) + 1$.

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

$$\gcd(a, b) = \gcd(r_0, r_1) = \gcd(r_\ell, r_{\ell+1}) = \gcd(r_\ell, 0) = r_\ell / \operatorname{lc}(r_\ell).$$

That proves the first statement.

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

$$\deg(r_1), \deg(r_2), \dots, \deg(r_\ell)$$

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

This gives us the following Euclidean algorithm for polynomials, which takes as input polynomials $a, b \in F[\mathtt{X}]$ with $\deg(a) \geq \deg(b)$ and $a \neq 0$, and which produces as output $d = \gcd(a, b) \in F[\mathtt{X}]$.

```
$r \leftarrow a,\ r' \leftarrow b$
while $r' \neq 0$ do
    $r'' \leftarrow r \bmod r'$
    $(r, r') \leftarrow (r', r'')$
$d \leftarrow r / \operatorname{lc}(r)$   // make monic
output $d$
```

Theorem 18.3. *Euclid's algorithm for polynomials uses* $O(\operatorname{len}(a)\operatorname{len}(b))$ *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 $d = \gcd(a, b)$, then $aF[\mathtt{X}] + bF[\mathtt{X}] = dF[\mathtt{X}]$, and so there exist polynomials s and t such that $as + bt = d$. The procedure to calculate 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 18.4. *Let* a*,* b*,* $r_0, r_1, \dots, r_{\ell+1}$ *and* $q_1, \dots, q_\ell$ *be as in Theorem 18.2. Define polynomials* $s_0, s_1, \dots, s_{\ell+1} \in F[\mathtt{X}]$ *and* $t_0, t_1, \dots, t_{\ell+1} \in F[\mathtt{X}]$ *as follows:*

$$\begin{aligned} s_0 &:= 1, \quad t_0 := 0, \\ s_1 &:= 0, \quad t_1 := 1, \end{aligned}$$

and for $i = 1, \dots, \ell$*,*

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

Then:

(i) for $i = 0, \dots, \ell + 1$, we have $s_i a + t_i b = r_i$; in particular, $s_\ell a + t_\ell b = \mathrm{lc}(r_\ell) \gcd(a, b)$;

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

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

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

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

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

$$\deg(s_i) = \deg(b) - \deg(r_{i-1}).$$

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 $i = 1, \dots, \ell$, 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 $i = 2, \dots, \ell$, we have $\deg(r_{i-1}) > \deg(r_i)$.

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

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

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

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

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

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

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

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

Note that part (iv) of the theorem implies that for $i = 1, \ldots, \ell + 1$, we have $\deg(t_i) \leq \deg(a)$ and $\deg(s_i) \leq \deg(b)$. Moreover, if $\deg(a) > 0$ and $b \neq 0$, then $\ell > 0$ and $\deg(r_{\ell-1}) > 0$, and hence $\deg(t_\ell) < \deg(a)$ and $\deg(s_\ell) < \deg(b)$.

We can easily turn the scheme described in Theorem 18.4 into a simple algorithm, taking as input polynomials $a, b \in F[\mathtt{X}]$, such that $\deg(a) \geq \deg(b)$ and $a \neq 0$, and producing as output polynomials $d, s, t \in F[\mathtt{X}]$ such that $d = \gcd(a, b)$ and $as + bt = d$:

```
$r \leftarrow a,\ r' \leftarrow b$
$s \leftarrow 1,\ s' \leftarrow 0$
$t \leftarrow 0,\ t' \leftarrow 1$
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 18.5. *The extended Euclidean algorithm for polynomials uses* $O(\mathrm{len}(a)\,\mathrm{len}(b))$ *operations in* F.

Proof. Exercise. □

18.4 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 $F[\mathtt{X}]/(n)$, where $n \in F[\mathtt{X}]$ with $\ell := \deg(n) > 0$.

Given $y \in F[\mathtt{X}]$ with $\deg(y) < \ell$, using $O(\ell^2)$ operations in F, we can determine if y is relatively prime to n, and if so, compute $y^{-1} \bmod n$ as follows. We run the extended Euclidean algorithm on inputs $a := n$ and $b := y$, obtaining polynomials d, s, t such that $d = \gcd(n, y)$ and $ns + yt = d$. If $d \neq 1$, then y does not have a multiplicative inverse modulo n. Otherwise, if $d = 1$, then t is a multiplicative inverse of y modulo n. Moreover, by Theorem 18.4, and the discussion immediately following, $\deg(t) < \ell$, and so $t = y^{-1} \bmod n$.

If the polynomial n is irreducible, then $F[\mathtt{X}]/(n)$ is a field, and the extended Euclidean algorithm, together with the basic algorithms for addition, subtraction, and multiplication modulo n, yield efficient algorithms for performing addition, subtraction, multiplication and division in the extension field $F[\mathtt{X}]/(n)$.

We also observe that the Chinese remainder theorem for polynomials (Theorem 17.17) can be made computationally effective as well:

Theorem 18.6. *Given polynomials $n_1, \ldots, n_k \in F[\mathtt{X}]$ and $a_1, \ldots, a_k \in F[\mathtt{X}]$, where $n_1, \ldots, n_k$ are pairwise relatively prime, and where $\deg(n_i) > 0$ and $\deg(a_i) < \deg(n_i)$ for $i = 1, \ldots, k$, we can compute the polynomial $z \in F[\mathtt{X}]$, such that $\deg(z) < \deg(n)$ and $z \equiv a_i \pmod{n_i}$ for $i = 1, \ldots, k$, where $n := \prod_i n_i$, using $O(\mathrm{len}(n)^2)$ operations in F.*

Proof. Exercise (just use the formulas in the proof of Theorem 2.8, which are repeated below the statement of Theorem 17.17). □

18.4.1 Chinese remaindering and polynomial interpolation

We remind the reader of the discussion following Theorem 17.17, where the point was made that when $n_i = (\mathtt{X} - b_i)$ for $i = 1, \ldots, k$, then the Chinese remainder theorem for polynomials reduces to Lagrange interpolation. Thus, Theorem 18.6 says that given distinct elements $b_1, \ldots, b_k \in F$, along with elements $a_1, \ldots, a_k \in F$, we can compute the unique polynomial $z \in F[\mathtt{X}]$ of degree less than k such that

$$z(b_i) = a_i \quad (i = 1, \ldots, k),$$

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. 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.

Exercise 18.7. State and re-work the polynomial analog of Exercises 4.3 and 4.4. In the special case of polynomial interpolation, this algorithm is called **Newton interpolation**.

18.4.2 Mutual independence and secret sharing

As we also saw in the discussion following Theorem 17.17, for $\ell \le k$ and fixed and distinct $b_1, \ldots, b_\ell \in F$, the "multi-point evaluation" map $\sigma : F[\mathtt{X}]_{<k} \to F^{\times \ell}$ that sends a polynomial $z \in F[\mathtt{X}]$ of degree less than k to $(z(b_1), \ldots, z(b_\ell)) \in F^{\times \ell}$ is a surjective F-linear map.

If F is a finite field, then this has the following probabilistic interpretation: if the coefficient vector $(z_0, \ldots, z_{k-1})$ of z is a random variable, uniformly distributed over $F^{\times k}$, then the random variable $(z(b_1), \ldots, z(b_\ell))$ is uniformly distributed over $F^{\times \ell}$ (see part (a) of Exercise 8.22). Put another way, the collection $\{z(b) : b \in F\}$ of random variables is ℓ-wise independent, where each individual $z(b)$ is uniformly distributed over F. Clearly, given z and b, we can efficiently compute the value of $z(b)$, so this construction gives us a nice way to build effectively constructible, ℓ-wise independent collections of random variables for any ℓ, thus generalizing the constructions in Example 6.17 and Exercise 6.16 of pairwise and 3-wise independent collections.

As a particular application of this idea, we describe a simple **secret sharing scheme**. Suppose Alice wants to share a secret among some number m of parties, call them $P_1, \ldots, P_m$, in such a way that if less than k parties share their individual secret shares with one another, then Alice's secret is still well hidden, while any subset of k parties can reconstruct Alice's secret.

She can do this as follows. Suppose her secret s is (or can be encoded as) an element of a finite field F, and that $b_0, b_1, \ldots, b_m$ are some fixed, distinct elements of F, where $b_0 = 0$. This presumes, of course, that $|F| \ge m+1$. To share her secret s, Alice chooses $z_1, \ldots, z_{k-1} \in F$ at random, and sets $z_0 := s$. Let $z \in F[\mathtt{X}]$ be the polynomial whose coefficient vector is $(z_0, \ldots, z_{k-1})$; that is,

$$z = \sum_{i=0}^{k-1} z_i \mathtt{X}^i.$$

For $i = 1, \ldots, m$, Alice gives party P_i its share

$$a_i := z(b_i).$$

For the purposes of analysis, it is convenient to define

$$a_0 := z(b_0) = z(0) = z_0 = s.$$

Clearly, if any k parties pool their shares, they can reconstruct Alice's secret by interpolating a polynomial of degree less than k at k points—the constant term of this polynomial is equal to Alice's secret s.

It remains to show that Alice's secret remains well hidden provided less than k parties pool their shares. To do this, first assume that Alice's secret s is uniformly distributed over F, independently of $z_1, \ldots, z_{k-1}$ (we will relax this assumption below). With this assumption, $z_0, z_1, \ldots, z_{k-1}$ are independently and uniformly distributed over F. Now consider any subset of $k-1$ parties; to simplify notation, assume the parties are $P_1, \ldots, P_{k-1}$. Then the random variables $a_0, a_1, \ldots, a_{k-1}$ are mutually independent. The variables $a_1, \ldots, a_{k-1}$ are of course the shares of $P_1, \ldots, P_{k-1}$, while a_0 is equal to Alice's secret (the fact that a_0 has two interpretations, one as the value of z at a point, and one as a coefficient of z, plays a crucial role in the analysis). Because of mutual independence, the distribution of a_0, conditioned on fixed values of the shares $a_1, \ldots, a_{k-1}$, is still uniform over F, and so even by pooling their shares, these $k-1$ parties would have no better chance of guessing Alice's secret than they would have without pooling their shares.

Continuing the analysis of the previous paragraph, consider the conditional probability distribution in which we condition on the event that $a_0 = s$ for some specific, fixed value of $s \in F$. Because the $z_0, z_1, \ldots, z_{k-1}$ were initially independently and uniformly distributed over F, and because $z_0 = a_0$, in this conditional probability distribution, we have $z_0 = s$ and $z_1, \ldots, z_{k-1}$ are independently and uniformly distributed over F. So this conditional probability distribution perfectly models the secret sharing algorithm performed by Alice for a specific secret s, without presuming that s is drawn from any particular distribution. Moreover, because the $a_0, a_1, \ldots, a_{k-1}$ were initially independently and uniformly distributed over F, when we condition on the event $a_0 = s$, the variables $a_1, \ldots, a_{k-1}$ are still independently and uniformly distributed over F.

The argument in the previous two paragraphs shows that

> *for any fixed secret s, the shares $a_1, \ldots, a_m$ are $(k-1)$-wise independent, with each individual share a_i uniformly distributed over F.*

This property ensures that Alice's secret is *perfectly* hidden, provided that less than k parties pool their shares: for any secret s, these parties just see a bunch of random values in F, with no particular bias that would give any hint whatsoever as to the actual value of s.

Secret sharing has a number of cryptographic applications, but one simple motivation is the following. Alice may have some data that she wants to "back up" on some file servers, who play the role of the parties $P_1, \ldots, P_m$.

To do this, Alice gives each server a share of her secret data (if she has a lot of data, she can break it up into many small blocks, and process each block separately). If at a later time, Alice wants to restore her data, she contacts any k servers who will give Alice their shares, from which Alice can reconstruct the original data. In using a secret sharing scheme in this way, Alice trusts that the servers are reliable to the extent that they do not modify the value of their shares (as otherwise, this would cause Alice to reconstruct the wrong data). We shall discuss later in this chapter how one can relax this trust assumption. But even with this trust assumption, Alice does gain something above and beyond the simpler solution of just backing up her data on a single server, namely:

- even if some of the servers crash, or are otherwise unreachable, she can still recover her data, as long as at least k are available at the time she wants to do the recovery;
- even if the data on some (but strictly less than k) of the servers is "leaked" to some outside attacker, the attacker gains no information about Alice's data.

EXERCISE 18.8. Suppose that Alice shares secrets $s_1, \ldots, s_t \in F$ with parties $P_1, \ldots, P_m$, so that each P_i has one share of each s_j. At a later time, Alice obtains all the shares held by k of the parties. Show how Alice can reconstruct all of the secrets $s_1, \ldots, s_t$ using $O(k^2 + tk)$ operations in F.

EXERCISE 18.9. Suppose that Alice shares secrets $s_1, \ldots, s_t \in F$ with parties $P_1, \ldots, P_m$, so that each P_i has one share of each s_j. Moreover, Alice does not want to trust that the parties do not maliciously (or accidentally) modify their shares. Show that if Alice has a small amount of secure storage, namely, space for $O(m)$ elements of F that cannot be read or modified by the other parties, then she can effectively protect herself from malicious parties, so that if any particular party tries to give her modified shares, Alice will fail to detect this with probability at most $t/|F|$. If $|F|$ is very large (say, $|F| = 2^{128}$), and t is any realistic value (say, $t \leq 2^{40}$), this failure probability will be acceptably small for all practical purposes. Hint: see Exercise 9.12.

18.4.3 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 $F[\mathtt{X}]$, 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. We leave the details as an exercise.

EXERCISE 18.10. You are give two matrices $A, B \in F[\mathtt{X}]^{\ell\times\ell}$. All entries of A and B are polynomials of degree at most M. Assume that $|F| \geq 2M+1$. Using polynomial evaluation and interpolation, show how to compute the product matrix $C = A \cdot B$ using $O(\ell^2 M^2 + \ell^3 M)$ operations in F. Compare this to the cost of computing C directly, which would be $O(\ell^3 M^2)$.

18.5 Rational function reconstruction and applications

We next state and prove the polynomial analog of Theorem 4.6. 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. Throughout this section, F denotes a field.

Theorem 18.7. *Let r^*, t^* be non-negative integers, and let $n, y \in F[\mathtt{X}]$ be polynomials such that $r^* + t^* \leq \deg(n)$ and $\deg(y) < \deg(n)$. Suppose we run the extended Euclidean algorithm with inputs $a := n$ and $b := y$. Then, adopting the notation of Theorem 18.4, the following hold:*

(i) There exists a unique index $i = 1, \ldots, \ell+1$, such that $\deg(r_i) < r^* \leq \deg(r_{i-1})$, and for this i, we have $t_i \neq 0$.

Let $r' := r_i$, $s' := s_i$, and $t' := t_i$.

(ii) Furthermore, for any polynomials $r, s, t \in F[\mathtt{X}]$ such that

$$r = sn + ty, \quad \deg(r) < r^*, \quad 0 \leq \deg(t) \leq t^*, \tag{18.1}$$

we have

$$r = r'\alpha, \ s = s'\alpha, \ t = t'\alpha,$$

for some non-zero polynomial $\alpha \in F[\mathtt{X}]$.

Proof. By hypothesis, $0 \leq r^* \leq \deg(n) = \deg(r_0)$. Moreover, since

$$\deg(r_0), \ldots, \deg(r_\ell), \deg(r_{\ell+1}) = -\infty$$

is a decreasing sequence, and $t_i \neq 0$ for $i = 1, \ldots, \ell + 1$, the first statement of the theorem is clear.

Now let i be defined as in the first statement of the theorem. Also, let r, s, t be as in (18.1).

From part (iv) of Theorem 18.4 and the inequality $r^* \leq \deg(r_{i-1})$, we have

$$\deg(t_i) = \deg(n) - \deg(r_{i-1}) \leq \deg(n) - r^*.$$

From the equalities $r_i = s_i n + t_i y$ and $r = sn + ty$, we have the two congruences:

$$r \equiv ty \pmod{n},$$
$$r_i \equiv t_i y \pmod{n}.$$

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

$$rt_i \equiv r_i t \pmod{n}.$$

This says that n divides $rt_i - r_i t$; however, using the bounds $\deg(r) < r^*$ and $\deg(t_i) \leq \deg(n) - r^*$, we see that $\deg(rt_i) < \deg(n)$, and using the bounds $\deg(r_i) < r^*$, $\deg(t) \leq t^*$, and $r^* + t^* \leq \deg(n)$, we see that $\deg(r_i t) < \deg(n)$; it immediately follows that

$$\deg(rt_i - r_i t) < \deg(n).$$

Since n divides $rt_i - r_i t$ and $\deg(rt_i - r_i t) < \deg(n)$, the only possibility is that

$$rt_i - r_i t = 0.$$

The rest of the proof runs *exactly* the same as the corresponding part of the proof of Theorem 4.6, as the reader may easily verify. □

18.5.1 Application: polynomial interpolation with errors

We now discuss the polynomial analog of the application in §4.5.1.

If we "encode" a polynomial $z \in F[\mathtt{X}]$, with $\deg(z) < k$, as the sequence $(a_1, \ldots, a_k) \in F^{\times k}$, where $a_i = z(b_i)$, then we can efficiently recover z from this encoding, using an algorithm for polynomial interpolation. Here, of course, the b_i are distinct elements of F, and F is a finite field (which must have at least k elements, of course).

Now suppose that Alice encodes z as $(a_1, \ldots, a_k)$, and sends this encoding to Bob, but that some, say at most ℓ, of the a_i may be corrupted during transmission. Let $(\tilde{a}_1, \ldots, \tilde{a}_k)$ denote the vector actually received by Bob.

Here is how we can use Theorem 18.7 to recover the original value of z from $(\tilde{a}_1, \ldots, \tilde{a}_k)$, assuming:

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

Let us set $n_i := (\mathtt{X} - b_i)$ for $i = 1, \ldots, k$, and $n := n_1 \cdots n_k$. Now, suppose Bob obtains the corrupted encoding $(\tilde{a}_1, \ldots, \tilde{a}_k)$. Here is what Bob does to recover z:

1. Interpolate, obtaining a polynomial y, with $\deg(y) < k$ and $y(b_i) = \tilde{a}_i$ for $i = 1, \ldots, k$.
2. Run the extended Euclidean algorithm on $a := n$ and $b := y$, and let r', t' be the values obtained from Theorem 18.7 applied with $r^* := k' + \ell$ and $t^* := \ell$.
3. If $t' \mid r'$, output r'/t'; otherwise, output "error."

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

$$r \equiv ty \pmod{n}. \tag{18.2}$$

To show that (18.2) holds, it suffices to show that

$$tz \equiv ty \pmod{n_i} \tag{18.3}$$

for all $i = 1, \ldots, k$. To show this, consider first an index i at which no error occurred, so that $a_i = \tilde{a}_i$. Then $tz \equiv ta_i \pmod{n_i}$ and $ty \equiv t\tilde{a}_i \equiv ta_i \pmod{n_i}$, and so (18.3) holds for this i. Next, consider an index i for which an error occurred. Then by construction, $tz \equiv 0 \pmod{n_i}$ and $ty \equiv 0 \pmod{n_i}$, and so (18.3) holds for this i. Thus, (18.2) holds, from which it follows that the values r', t' obtained from Theorem 18.7 satisfy

$$\frac{r'}{t'} = \frac{r}{t} = \frac{tz}{t} = z.$$

One easily checks that both the procedures to encode and decode a value z run in time $O(k^2)$. The above scheme is an example of an **error correcting code** called a **Reed–Solomon code**. Note that we are completely free to choose the finite field F however we want, just so long as it is big enough. An attractive choice in some settings is to choose $F = \mathbb{Z}_2[\mathtt{Y}]/(f)$, where $f \in \mathbb{Z}_2[\mathtt{Y}]$ is an irreducible polynomial; with this choice, elements of F may be encoded as bit strings of length $\deg(f)$.

One can combine the above error correction technique with the idea of secret sharing (see §18.4.2) to obtain a secret sharing scheme that is robust, even in the presence of erroneous (as opposed to just missing) shares. More precisely, Alice can share a secret $s \in F$ among parties $P_1, \dots, P_m$, in such a way that (1) if less than k' parties pool their shares, Alice's secret remains well hidden, and (2) from any k shares, we can correctly reconstruct Alice's secret, provided at most ℓ of the shares are incorrect, and $k \geq 2\ell + k'$. To do this, Alice chooses $z_1, \dots, z_{k'-1} \in F$ at random, sets $z_0 := s$, and $z := \sum_{i=0}^{k'-1} z_i \mathtt{X}^i \in F[\mathtt{X}]$, and computes the ith share as $a_i := z(b_i)$, for $i = 1, \dots, m$. Here, we assume that the b_i are distinct, non-zero elements of F. Now, just as in §18.4.2, as long as less than k' parties pool their shares, Alice's secret remains well hidden; however, provided $k \geq 2\ell + k'$, we can correctly and efficiently reconstruct Alice's secret given any k values $\tilde{a}_i$, as long as at most ℓ of the $\tilde{a}_i$ differ from the corresponding value of a_i.

18.5.2 Application: recovering rational functions from their reversed formal Laurent series

We now discuss the polynomial analog of the application in §4.5.2. This is an entirely straightforward translation of the results in §4.5.2, but we shall see in the next chapter that this problem has its own interesting applications.

Suppose Alice knows a rational function $z = s/t \in F(\mathtt{X})$, where s and t are polynomials with $\deg(s) < \deg(t)$, and tells Bob some of the high-order coefficients of the reversed formal Laurent series (see §17.7) representing z in $F((\mathtt{X}^{-1}))$. We shall show that if $\deg(t) \leq M$ and Bob is given the bound M on $\deg(t)$, along with the high-order $2M$ coefficients of z, then Bob can determine z, expressed as a rational function in lowest terms.

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

$$y := z_1 \mathtt{X}^{2M-1} + \cdots + z_{2M-1}\mathtt{X} + z_{2M} = \lfloor z\mathtt{X}^{2M} \rfloor.$$

Let us define $n := \mathtt{X}^{2M}$, so that $y = \lfloor zn \rfloor$. Here is Bob's algorithm for recovering z:

1. Run the extended Euclidean algorithm on inputs $a := n$ and $b := y$, and let s', t' be as in Theorem 18.7, using $r^* := M$ and $t^* := M$.
2. Output s', t'.

We claim that $z = -s'/t'$. To prove this, observe that since $y = \lfloor zn \rfloor = \lfloor (ns)/t \rfloor$, if we set $r := (ns) \bmod t$, then we have

$$r = sn - ty, \ \deg(r) < r^*, \ 0 \leq \deg(t) \leq t^*, \text{ and } r^* + t^* \leq \deg(n).$$

It follows that the polynomials s', t' from Theorem 18.7 satisfy $s = s'\alpha$ and $-t = t'\alpha$ for some non-zero polynomial α. 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 $O(M^2)$ operations in F.

The following exercises are the polynomial analogs of Exercises 4.7, 4.9, and 4.10.

Exercise 18.11. Let F be a field. Show that given polynomials $s, t \in F[\mathtt{X}]$ and integer k, with $\deg(s) < \deg(t)$ and $k > 0$, we can compute the kth coefficient in the reversed formal Laurent series representing s/t using $O(\mathrm{len}(k)\,\mathrm{len}(t)^2)$ operations in F.

Exercise 18.12. Let F be a field. Let $z \in F((\mathtt{X}^{-1}))$ be a reversed formal Laurent series whose coefficient sequence is ultimately periodic. Show that $z \in F(\mathtt{X})$.

Exercise 18.13. Let F be a field. Let $z = s/t$, where $s, t \in F[\mathtt{X}]$, $\deg(s) < \deg(t)$, and $\gcd(s, t) = 1$. Let $d > 1$ be an integer.

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

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

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

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

18.5.3 Applications to symbolic algebra

Rational function reconstruction has applications in symbolic algebra, analogous to those discussed in §4.5.3. 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 $F[\mathtt{X}]$—the solution to such a system of equations will be a vector whose entries are elements of $F(\mathtt{X})$, the field of rational functions.

18.6 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 18.14. State and re-work the analog of Exercise 3.32 for $R[\mathtt{X}]$. 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 18.23–18.26 (see also the discussion in §18.7).

In Exercises 18.15–18.21 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 18.15. State and re-work the analog of Exercise 3.34 for $R[\mathtt{X}]$.

Exercise 18.16. This problem is the analog of Exercise 3.35 for $R[\mathtt{X}]$. Let us first define the notion of a "floating point" reversed formal Laurent series $\hat{z}$, which is represented as a pair (a, e), where $a \in R[\mathtt{X}]$ and $e \in \mathbb{Z}$—the value of $\hat{z}$ is $a\mathtt{X}^e \in R((\mathtt{X}^{-1}))$, and we call $\operatorname{len}(a)$ the **precision** of $\hat{z}$. We say that $\hat{z}$ is a **length k approximation** of $z \in R((\mathtt{X}^{-1}))$ if $\hat{z}$ has precision k and $\hat{z} = (1+\epsilon)z$ for $\epsilon \in R((\mathtt{X}^{-1}))$ with $\deg(\epsilon) \leq -k$, which is the same as saying that the high-order k coefficients of $\hat{z}$ and z are equal. Show how to compute—given monic $b \in R[\mathtt{X}]$ and positive integer k—a length k approximation of $1/b \in R((\mathtt{X}^{-1}))$ using $O(M(k))$ operations in R. Hint: using Newton iteration, show how to go from a length t approximation of $1/b$ to a length $2t$ approximation, making use of just the high-order $2t$ coefficients of b, and using $O(M(t))$ operations in R.

Exercise 18.17. State and re-work the analog of Exercise 3.36 for $R[\mathtt{X}]$. Assume that b is a monic polynomial.

Exercise 18.18. State and re-work the analog of Exercise 3.37 for $R[\mathtt{X}]$.

Conclude that a polynomial of length ℓ can be evaluated at ℓ points using $O(M(\ell)\operatorname{len}(\ell))$ operations in R.

Exercise 18.19. State and re-work the analog of Exercise 3.38 for $R[\mathtt{X}]$, assuming that R is a field of odd characteristic.

Exercise 18.20. State and re-work the analog of Exercise 3.40 for $R[\mathtt{X}]$. Assume that $2_R \in R^*$.

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

Exercise 18.21. Let $E := R[\mathtt{X}]$. Let $a, b \in E[\mathtt{Y}]$ with $a = \sum_{i=0}^{m-1} a_i \mathtt{Y}^i$ and $b = \sum_{i=0}^{m-1} b_i \mathtt{Y}^i$, where each a_i and b_i is a polynomial in $\mathtt{X}$ of degree less than k. The product $c := ab \in E[\mathtt{Y}]$ may be written $c = \sum_{i=0}^{2m-2} c_i \mathtt{Y}^i$, where each c_i is a polynomial in $\mathtt{X}$. Show how to compute c, given a and b, using $O(M(km))$ operations in R. Hint: for an appropriately chosen integer $t > 0$, first convert a, b to $\tilde{a}, \tilde{b} \in R[\mathtt{X}]$, where $\tilde{a} := \sum_{i=0}^{m-1} a_i \mathtt{X}^{ti}$ and $\tilde{b} := \sum_{i=0}^{m-1} b_i \mathtt{X}^{ti}$; next, compute $\tilde{c} := \tilde{a}\tilde{b} \in R[\mathtt{X}]$; finally, “read off” the values c_i from the coefficients of $\tilde{c}$.

Exercise 18.22. Assume that ℓ-bit integers can be multiplied in time $\bar{M}(\ell)$, where $\bar{M}$ is a well-behaved complexity function. Let $a, b \in \mathbb{Z}[\mathtt{X}]$ with $a = \sum_{i=0}^{m-1} a_i \mathtt{X}^i$ and $b = \sum_{i=0}^{m-1} b_i \mathtt{X}^i$, where each a_i and b_i is a non-negative integer, strictly less than 2^k. The product $c := ab \in \mathbb{Z}[\mathtt{X}]$ may be written $c = \sum_{i=0}^{2m-2} c_i \mathtt{X}^i$, where each c_i is a non-negative integer. Show how to compute c, given a and b, using $O(\bar{M}((k + \operatorname{len}(m))m))$ operations in R. Hint: for an appropriately chosen integer $t > 0$, first convert a, b to $\tilde{a}, \tilde{b} \in \mathbb{Z}$, where $\tilde{a} := \sum_{i=0}^{m-1} a_i 2^{ti}$ and $\tilde{b} := \sum_{i=0}^{m-1} b_i 2^{ti}$; next, compute $\tilde{c} := \tilde{a}\tilde{b} \in \mathbb{Z}$; finally, “read off” the values c_i from the bits of $\tilde{c}$.

The following exercises develop an important algorithm for multiplying polynomials in almost-linear time. For 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 $(g_0, \ldots, g_{2^n-1})$ to the vector $(g(1_R), g(\omega), \ldots, g(\omega^{2^n-1}))$, where $g := \sum_{i=0}^{2^n-1} g_i \mathtt{X}^i \in R[\mathtt{X}]$.

Exercise 18.23. 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 $m = 0, \dots, n$. 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 any 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 18.24. 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 $FFT(n, \omega; g_0, \dots, g_{2^n-1})$ that takes as inputs integer $n \geq 0$, a primitive 2^nth root of unity $\omega \in R$, and elements $g_0, \dots, g_{2^n-1} \in R$, and runs as follows:

```
if $n = 0$ then
    return $g_0$
else
    $(\alpha_0, \dots, \alpha_{2^{n-1}-1}) \leftarrow FFT(n-1, \omega^2; g_0, g_2, \dots, g_{2^n-2})$
    $(\beta_0, \dots, \beta_{2^{n-1}-1}) \leftarrow FFT(n-1, \omega^2; g_1, g_3, \dots, g_{2^n-1})$
    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 $(\gamma_0, \dots, \gamma_{2^n-1})$
```

Show that this algorithm correctly computes $\mathcal{E}_{n,\omega}(g_0, \dots, g_{2^n-1})$ using $O(2^n n)$ operations in R.

Exercise 18.25. Assume $2_R \in R^*$. Suppose that we are given two polynomials $a, b \in R[\mathtt{X}]$ 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" a and b, writing $a = \sum_{i=0}^{2^n-1} a_i\mathtt{X}_i$ and $b = \sum_{i=0}^{2^n-1} b_i\mathtt{X}_i$, where a_i and b_i are zero for $i \geq \ell$. Show that the following algorithm correctly computes the product of a and b using $O(\ell \operatorname{len}(\ell))$ operations in R:

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

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

Exercise 18.26. 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 as in 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 $a, b \in R[\mathtt{X}]$ are of length at most ℓ. Set $k := \lfloor \sqrt{\ell/2} \rfloor$, $m := \lceil \ell/k \rceil$. We may write $a = \sum_{i=0}^{m-1} a_i \mathtt{X}^{ki}$ and $b = \sum_{i=0}^{m-1} b_i \mathtt{X}^{ki}$, where the a_i and b_i are polynomials of length at most k. Let n be the integer determined by $2m \le 2^n < 4m$. Let $f := \mathtt{X}^{2^{n-1}} + 1_R \in R[\mathtt{X}]$, $E := R[\mathtt{X}]/(f)$, and $\omega := [\mathtt{X}]_f \in E$.

(a) Show that ω is a primitive 2^nth root of unity in E, and that given an element $\delta \in E$ and an integer i between 0 and $2^n - 1$, we can compute $\delta\omega^i \in E$ using $O(\ell^{1/2})$ operations in R.

(b) Let $\bar{a} := \sum_{i=0}^{m-1} [a_i]_f \mathtt{Y}^i \in E[\mathtt{Y}]$ and $\bar{b} := \sum_{i=0}^{m-1} [b_i]_f \mathtt{Y}^i \in E[\mathtt{Y}]$. Using the FFT (over E), show how to compute $\bar{c} := \bar{a}\bar{b} \in E[\mathtt{Y}]$ by computing $O(\ell^{1/2})$ products in $R[\mathtt{X}]$ 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 $c := ab \in R[\mathtt{X}]$ from the value $\bar{c} \in E[\mathtt{Y}]$ 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 18.27. 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 ℓ-bit, positive integers a and b (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 5.24 (and the result of Exercise 5.22), and assuming a deterministic, polynomial-time primality test (such as the one to be presented in Chapter 22), 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
$$2^{2k} m < p \le \ell^{O(1)}.$$
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 $\bar{a} := \sum_{i=0}^{m-1} [a_i]_p \mathtt{X}^i \in \mathbb{Z}_p[\mathtt{X}]$ and $\bar{b} := \sum_{i=0}^{m-1} [b_i]_p \mathtt{X}^i \in \mathbb{Z}_p[\mathtt{X}]$, and show how to compute $\bar{c} := \bar{a}\bar{b} \in \mathbb{Z}_p[\mathtt{X}]$ 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 $\bar{c} \in \mathbb{Z}_p[\mathtt{X}]$ as in part (b), show how to obtain $c := ab$ in time $O(\ell)$.

(d) Conclude that assuming Conjecture 5.24, we can multiply two ℓ-bit integers 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 18.28. 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 5.25, show how to make this algorithm fully deterministic and rigorous, provided 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 18.29. This exercise shows how the algorithm in Exercise 18.27 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 18.27 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 18.27, 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 18.30. 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 $\mathbb{Z}_n[\mathtt{X}]$, 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 16, while faster, are either probabilistic, or deterministic but heuristic. Assume that we can multiply polynomials in $\mathbb{Z}_n[\mathtt{X}]$ 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 18.26 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 all of the integers $a_1, \ldots, a_\ell$ 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.

18.7 Notes

Exercise 18.4 is based on an algorithm of Brent and Kung [20]. Using fast matrix arithmetic, Brent and Kung show how this problem can be solved using $O(\ell^{(\omega+1)/2})$ operations in R, where ω is the exponent for matrix multiplication (see §15.6), and so $(\omega + 1)/2 < 1.7$.

The interpretation of Lagrange interpolation as "secret sharing" (see §18.4.2), and its application to cryptography, was made by Shamir [85].

Reed–Solomon codes were first proposed by Reed and Solomon [77], although the decoder presented here was developed later. Theorem 18.7 was proved by Mills [64]. 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 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 18.25), 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 18.26). 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 [37] 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.