Chapter 2

NUMBER-THEORETICAL TOOLS

In this chapter we focus specifically on those fundamental tools and associated computational algorithms that apply to prime number and factorization studies. Enhanced integer algorithms, including various modern refinements of the classical ones of the present chapter, are detailed in Chapter 8.8. The reader may wish to refer to that chapter from time to time, especially when issues of computational complexity and optimization are paramount.

2.1 Modular arithmetic

Throughout prime-number and factorization studies the notion of modular arithmetic is a constant reminder that one of the great inventions of mathematics is to consider numbers modulo N, in so doing effectively contracting the infinitude of integers into a finite set of residues. Many theorems on prime numbers involve reductions modulo p, and most factorization efforts will use residues modulo N, where N is the number to be factored.

A word is in order on nomenclature. Here and elsewhere in the book, we denote by $x \bmod N$ the least nonnegative residue $x \pmod{N}$. The mod notation without parentheses is convenient when thought of as an algorithm step or a machine operation (more on this operator notion is said in Section 9.1.3). So, the notation $x^y \bmod N$ means the y-th power of x, reduced to the interval $[0, N-1]$ inclusive; and we allow negative values for exponents y when x is coprime to N, so that an operation $x^{-1} \bmod N$ yields a reduced inverse, and so on.

2.1.1 Greatest common divisor and inverse

In this section we exhibit algorithms for one of the very oldest operations in computational number theory, the evaluation of the greatest common divisor function $\gcd(x, y)$. Closely related is the problem of inversion, the evaluation of $x^{-1} \bmod N$, which operation yields (when it exists) the unique integer $y \in [1, N-1]$ with $xy \equiv 1 \pmod{N}$. The connection between the gcd and inversion operations is especially evident on the basis of the following fundamental result.

Theorem 2.1.1 (Linear relation for gcd). *If x, y are integers not both 0, then there are integers a, b with*

$$ax + by = \gcd(x, y). \tag{2.1}$$

Proof. Let g be the least positive integer in the form $ax + yb$, where a, b are integers. (There is at least one positive integer in this form, to wit, $x^2 + y^2$.) We claim that $g = \gcd(x, y)$. Clearly, any common divisor of x and y divides $g = ax + by$. So $\gcd(x, y)$ divides g. Suppose g does not divide x. Then $x = tg + r$, for some integer r with $0 < r < g$. We then observe that $r = (1 - ta)x - tby$, contradicting the definition of g. Thus, g divides x, and similarly, g divides y. We conclude that $g = \gcd(x, y)$. □

The connection of (2.1) to inversion is immediate: If x, y are positive integers and $\gcd(x, y) = 1$, then we can solve $ax + by = 1$, whence

$$b \bmod x, \ \ a \bmod y$$

are the inverses $y^{-1} \bmod x$ and $x^{-1} \bmod y$, respectively.

However, what is clearly lacking from the proof of Theorem 2.1.1 from a computational perspective is any clue on how one might find a solution a, b to (2.1). We investigate here the fundamental, classical methods, beginning with the celebrated centerpiece of the classical approach: the Euclid algorithm. It is arguably one of the very oldest computational schemes, dating back to 300 B.C., if not the oldest of all. In this algorithm and those following, we indicate the updating of two variables x, y by

$$(x, y) = (f(x, y), g(x, y)),$$

which means that the pair (x, y) is to be replaced by the pair of evaluations (f, g) but with the evaluations using the original (x, y) pair. In similar fashion, longer vector relations $(a, b, c, \ldots) = \cdots$ update all components on the left, each using the original values on the right side of the equation. (This rule for updating of vector components is discussed in the Appendix.)

Algorithm 2.1.2 (Euclid algorithm for greatest common divisor). For integers x, y with $x \ge y \ge 0$ and $x > 0$, this algorithm returns $\gcd(x, y)$.

```
1. [Euclid loop]
      while($y > 0$) $(x, y) = (y, x \bmod y)$;
      return $x$;
```

It is intriguing that this algorithm, which is as simple and elegant as can be, is not so easy to analyze in complexity terms. Though there are still some interesting open questions as to detailed behavior of the algorithm, the basic complexity is given by the following theorem:

Theorem 2.1.3 (Lamé, Dixon, Heilbronn). *Let $x > y$ be integers from the interval $[1, N]$. Then the number of steps in the loop of the Euclid Algorithm 2.1.2 does not exceed*

$$\left\lceil \ln\left(N\sqrt{5}\right) / \ln\left(\left(1 + \sqrt{5}\right)/2\right)\right\rceil - 2,$$

and the average number of loop steps (over all choices x, y) is asymptotic to

$$\frac{12 \ln 2}{\pi^2} \ln N.$$

The first part of this theorem stems from an interesting connection between Euclid's algorithm and the theory of simple continued fractions (see Exercise 2.4). The second part involves the measure theory of continued fractions.

If x, y are each of order of magnitude N, and we employ the Euclid algorithm together with, say, a classical mod operation, it can be shown that the overall complexity of the gcd operation will then be

$$O\left(\ln^2 N\right)$$

bit operations, essentially the square of the number of digits in an operand (see Exercise 2.6). This complexity can be genuinely bested via modern approaches, and not merely by using a faster mod operation, as we discuss in our final book chapter.

The Euclid algorithm can be extended to the problem of inversion. In fact, the appropriate extension of the Euclid algorithm will provide a complete solution to the relation (2.1):

Algorithm 2.1.4 (Euclid's algorithm extended, for gcd and inverse). For integers x, y with $x \geq y \geq 0$ and $x > 0$, this algorithm returns an integer triple (a, b, g) such that $ax + by = g = \gcd(x, y)$. (Thus when $g = 1$ and $y > 0$, the residues $b \pmod{x}$, $a \pmod{y}$ are the inverses of $y \pmod{x}, x \pmod{y}$, respectively.)

```
1. [Initialize]
       $(a, b, g, u, v, w) = (1, 0, x, 0, 1, y)$;
2. [Extended Euclid loop]
       while($w > 0$) {
           $q = \lfloor g/w \rfloor$;
           $(a, b, g, u, v, w) = (u, v, w, a - qu, b - qv, g - qw)$;
       }
       return $(a, b, g)$;
```

Because the algorithm simultaneously returns the relevant gcd and both inverses (when the input integers are coprime and positive), it is widely used as an integral part of practical computational packages. Interesting computational details of this and related algorithms are given in [Cohen 2000], [Knuth 1981]. Modern enhancements are covered in Chapter 8.8 including asymptotically faster gcd algorithms, faster inverse, inverses for special moduli, and so on. Finally, note that in Section 2.1.2 we give an "easy inverse" method (relation (2.3)) that might be considered as a candidate in computer implementations.

2.1.2 Powers

It is a celebrated theorem of Euler that

$$a^{\varphi(m)} \equiv 1 \pmod{m} \tag{2.2}$$

holds for any positive integer m as long as a, m are coprime. In particular, for prime p we have

$$a^{p-1} \equiv 1 \pmod{p},$$

which is used frequently as a straightforward initial (though not absolute) primality criterion. The point is that powering is an important operation in prime number studies, and we are especially interested in powering with modular reduction. Among the many applications of powering is this one: A straightforward method for finding inverses is to note that when $a^{-1} \pmod{m}$ exists, we always have the equality

$$a^{-1} \bmod m = a^{\varphi(m)-1} \bmod m, \tag{2.3}$$

and this inversion method might be compared with Algorithm 2.1.4 when machine implementation is contemplated.

It is a primary computational observation that one usually does not need to take an n-th power of some x by literally multiplying together n symbols as $x{*}x{*}\cdots{*}x$. We next give a radically more efficient (for large powers) recursive powering algorithm that is easily written out and also easy to understand. The objects that we raise to powers might be integers, members of a finite field, polynomials, or something else. We specify in the algorithm that the element x comes only from a semigroup, namely, a setting in which $x * x * \cdots * x$ is defined.

Algorithm 2.1.5 (Recursive powering algorithm). Given an element x in a semigroup and a positive integer n, the goal is to compute x^n.

```
1. [Recursive function $pow$]
       $pow(x, n)$ {
           if($n == 1$) return $x$;
           if($n$ even) return $pow(x, n/2)^2$;                    // Even branch.
           return $x * pow(x, (n-1)/2)^2$;                         // Odd branch.
       }
```

This algorithm is recursive and compact, but for actual implementation one should consider the ladder methods of Section 9.3.1, which are essentially equivalent to the present one but are more appropriate for large, array-stored arguments. To exemplify the recursion in Algorithm 2.1.5, consider $3^{13} \pmod{15}$. Since $n = 13$, we can see that the order of operations will be

$$\begin{aligned} 3 * pow(3,6)^2 &= 3 * \left(pow(3,3)^2\right)^2 \\ &= 3 * \left(\left(3 * pow(3,1)^2\right)^2\right)^2. \end{aligned}$$

If one desires $x^n \bmod m$, then the required modular reductions are to occur for each branch (even, odd) of the algorithm. If the modulus is $m = 15$, say, casual inspection of the final power chain above shows that the answer is $3^{13} \bmod 15 = 3 \cdot \left((-3)^2\right)^2 \bmod 15 = 3 \cdot 6 \bmod 15 = 3$. The important observation, though, is that there are three squarings and two multiplications, and such operation counts depend on the binary expansion of the exponent n, with typical operation counts being dramatically less than the value of n itself. In fact, if $x,\ n$ are integers the size of m, and we are to compute $x^n \bmod m$ via naive multiply/add arithmetic and Algorithm 2.1.5, then $O(\ln^3 m)$ bit operations suffice for the powering (see Exercise 2.17 and Section 9.3.1).

2.1.3 Chinese remainder theorem

The Chinese remainder theorem (CRT) is a clever, and very old, idea from which one may infer an integer value on the basis of its residues modulo an appropriate system of smaller moduli. The CRT was known to Sun-Zi in the first century A.D. [Hardy and Wright 1979], [Ding et al. 1996]; in fact a legendary ancient application is that of counting a troop of soldiers. If there are n soldiers, and one has them line up in justified rows of 7 soldiers each, one inspects the last row and infers $n \bmod 7$, while lining them up in rows of 11 will give $n \bmod 11$, and so on. If one does "enough" such small-modulus operations, one can infer the exact value of n. In fact, one does not need the small moduli to be primes; it is sufficient that the moduli be pairwise coprime.

Theorem 2.1.6 (Chinese remainder theorem (CRT)). *Let* $m_0, \dots, m_{r-1}$ *be positive, pairwise coprime moduli with product* $M = \Pi_{i=0}^{r-1} m_i$. *Let* r *respective residues* n_i *also be given. Then the system comprising the* r *relations and inequality*

$$n \equiv n_i \pmod{m_i}, \quad 0 \le n < M$$

has a unique solution. Furthermore, this solution is given explicitly by the least nonnegative residue modulo M *of*

$$\sum_{i=0}^{r-1} n_i v_i M_i,$$

where $M_i = M/m_i$, *and the* v_i *are inverses defined by* $v_i M_i \equiv 1 \pmod{m_i}$.

A simple example should serve to help clarify the notation. Let $m_0 = 3,\ m_1 = 5,\ m_2 = 7$, for which the overall product is $M = 105$, and let $n_0 = 2,\ n_1 = 2,\ n_2 = 6$. We seek a solution $n < 105$ to

$$n \equiv 2 \pmod 3,\ n \equiv 2 \pmod 5,\ n \equiv 6 \pmod 7.$$

We first establish the M_i, as

$$M_0 = 35,\ M_1 = 21,\ M_2 = 15.$$

Then we compute the inverses

$$v_0 = 2 = 35^{-1} \bmod 3, \quad v_1 = 1 = 21^{-1} \bmod 5, \quad v_2 = 1 = 15^{-1} \bmod 7.$$

Then we compute

$$\begin{aligned} n &= (n_0 v_0 M_0 + n_1 v_1 M_1 + n_2 v_2 M_2) \bmod M \\ &= (140 + 42 + 90) \bmod 105 \\ &= 62. \end{aligned}$$

Indeed, 62 modulo $3, 5, 7$, respectively, gives the required residues $2, 2, 6$.

Though ancient, the CRT algorithm still finds many applications. Some of these are discussed in Chapter 8.8 and its exercises. For the moment, we observe that the CRT affords a certain "parallelism." A set of separate machines can perform arithmetic, each machine doing this with respect to a small modulus m_i, whence some final value may be reconstructed. For example, if each of x, y has fewer than 100 digits, then a set of prime moduli $\{m_i\}$ whose product is $M > 10^{200}$ can be used for multiplication: The i-th machine would find $((x \bmod m_i) * (y \bmod m_i)) \bmod m_i$, and the final value $x * y$ would be found via the CRT. Likewise, on one computer chip, separate multipliers can perform the small-modulus arithmetic.

All of this means that the reconstruction problem is paramount; indeed, the reconstruction of n tends to be the difficult phase of CRT computations. Note, however, that if the small moduli are fixed over many computations, a certain amount of one-time precomputation is called for. In Theorem 2.1.6, one may compute the M_i and the inverses v_i just once, expecting many future computations with different residue sets $\{n_i\}$. In fact, one may precompute the products $v_i M_i$. A computer with r parallel nodes can then reconstruct $\sum n_i v_i M_i$ in $O(\ln r)$ steps.

There are other ways to organize the CRT data, such as building up one partial modulus at a time. One such method is the Garner algorithm [Menezes et al. 1997], which can also be done with preconditioning.

Algorithm 2.1.7 (CRT reconstruction with preconditioning (Garner)). Using the nomenclature of Theorem 2.1.6, we assume $r \geq 2$ fixed, pairwise coprime moduli $m_0, \ldots, m_{r-1}$ whose product is M, and a set of given residues $\{n_i \pmod{m_i}\}$. This algorithm returns the unique $n \in [0, M-1]$ with the given residues. After the precomputation step, the algorithm may be reentered for future evaluations of such n (with the $\{m_i\}$ remaining fixed).

```
1. [Precomputation]
      for($1 \le i < r$) {
         $\mu_i = \prod_{j=0}^{i-1} m_j$;
         $c_i = \mu_i^{-1} \bmod m_i$;
      }
      $M = \mu_{r-1} m_{r-1}$;
2. [Reentry point for given input residues $\{n_i\}$]
      $n = n_0$;
      for($1 \le i < r$) {
         $u = ((n_i - n)c_i) \bmod m_i$;
         $n = n + u\mu_i$;          // Now $n \equiv n_j \pmod{m_j}$ for $0 \le j \le i$;
      }
      $n = n \bmod M$;
      return $n$;
```

This algorithm can be shown to be more efficient than a naive application of Theorem 2.1.6 (see Exercise 2.8). Moreover, in case a fixed modulus M

is used for repeated CRT calculations, one can perform [Precomputation] for Algorithm 2.1.7 just once, store an appropriate set of $r-1$ integers, and allow efficient reentry.

In Section 9.5.9 we describe a CRT reconstruction algorithm that not only takes advantage of preconditioning, but of fast methods to multiply integers.

2.2 Polynomial arithmetic

Many of the algorithms for modular arithmetic have almost perfect analogues in the polynomial arena.

2.2.1 Greatest common divisor for polynomials

We next give algorithms for polynomials analogous to the Euclid forms in Section 2.1.1 for integer gcd and inverse. When we talk about polynomials, the first issue is where the coefficients come from. We may be dealing with $\mathbf{Q}[x]$, the polynomials with rational coefficients, or $\mathbf{Z}_p[x]$, polynomials with coefficients in the finite field $\mathbf{Z}_p$. Or from some other field. We may also be dealing with polynomials with coefficients drawn from a ring that is not a field, as we do when we consider $\mathbf{Z}[x]$ or $\mathbf{Z}_n[x]$ with n not a prime.

Because of the ambiguity of the arena in which we are to work, perhaps it is better to go back to first principles and begin with the more primitive concept of divide with remainder. If we are dealing with polynomials in $F[x]$, where F is a field, there is a division theorem completely analogous to the situation with ordinary integers. Namely, if $f(x), g(x)$ are in $F[x]$ with f not the zero polynomial, then there are (unique) polynomials $q(x), r(x)$ in $F[x]$ with

$$g(x) = q(x)f(x) + r(x) \text{ and either } r(x) = 0 \text{ or } \deg r(x) < \deg f(x). \quad (2.4)$$

Moreover, we can use the "grammar-school" method of building up the quotient $q(x)$ term by term to find $q(x)$ and $r(x)$. Thinking about this method, one sees that the only special property of fields that is used that is not enjoyed by a general commutative ring is that the leading coefficient of the divisor polynomial $f(x)$ is invertible. So if we are in the more general case of polynomials in $R[x]$ where R is a commutative ring with identity, we can perform a divide with remainder if the leading coefficient of the divisor polynomial is a unit, that is, it has a multiplicative inverse in the ring.

For example, say we wish to divide $3x + 2$ into x^2 in the polynomial ring $\mathbf{Z}_{10}[x]$. The inverse of 3 in $\mathbf{Z}_{10}$ (which can be found by Algorithm 2.1.4) is 7. We get the quotient $7x + 2$ and remainder 6.

In sum, if $f(x), g(x)$ are in $R[x]$, where R is a commutative ring with identity and the leading coefficient of f is a unit in R, then there are unique polynomials $q(x), r(x)$ in $R[x]$ such that (2.4) holds. We use the notation $r(x) = g(x) \bmod f(x)$. For much more on polynomial remaindering, see Section 9.6.2.

Though it is possible sometimes to define the gcd of two polynomials in the more general case of $R[x]$, in what follows we shall restrict the discussion to the much easier case of $F[x]$, where F is a field. In this setting the algorithms and theory are almost entirely the same as for integers. (For a discussion of gcd in the case where R is not necessarily a field, see Section 4.3.) We define the polynomial gcd of two polynomials, not both 0, as a polynomial of greatest degree that divides both polynomials. Any polynomial satisfying this definition of gcd, when multiplied by a nonzero element of the field F, again satisfies the definition. To standardize things, we take among all these polynomials the monic one, that is the polynomial with leading coefficient 1, and it is this particular polynomial that is indicated when we use the notation $\gcd(f(x), g(x))$. Thus, $\gcd(f(x), g(x))$ is the monic polynomial common divisor of $f(x)$ and $g(x)$ of greatest degree. To render any nonzero polynomial monic, one simply multiplies through by the inverse of the leading coefficient.

Algorithm 2.2.1 (gcd for polynomials). For given polynomials $f(x), g(x)$ in $F[x]$, not both zero, this algorithm returns $d(x) = \gcd(f(x), g(x))$.

```
1. [Initialize]
        Let $u(x), v(x)$ be $f(x), g(x)$ in some order so that either $\deg u(x) \geq$
            $\deg v(x)$ or $v(x)$ is 0;
2. [Euclid loop]
        while($v(x) \neq 0$) $(u(x), v(x)) = (v(x), u(x) \bmod v(x))$;
3. [Make monic]
        Set $c$ as the leading coefficient of $u(x)$;
        $d(x) = c^{-1}u(x)$;
        return $d(x)$;
```

Thus, for example, if we take

$$\begin{aligned} f(x) &= 7x^{11} + x^9 + 7x^2 + 1, \\ g(x) &= -7x^7 - x^5 + 7x^2 + 1, \end{aligned}$$

in $\mathbf{Q}[x]$, then the sequence in the Euclid loop is

$$\begin{aligned} &(7x^{11} + x^9 + 7x^2 + 1,\ -7x^7 - x^5 + 7x^2 + 1) \\ &\to (-7x^7 - x^5 + 7x^2 + 1,\ 7x^6 + x^4 + 7x^2 + 1) \\ &\to (7x^6 + x^4 + 7x^2 + 1,\ 7x^3 + 7x^2 + x + 1) \\ &\to (7x^3 + 7x^2 + x + 1,\ 14x^2 + 2) \\ &\to (14x^2 + 2,\ 0), \end{aligned}$$

so the final value of $u(x)$ is $14x^2+2$, and the gcd $d(x)$ is $x^2 + \frac{1}{7}$. It is, of course, understood that all calculations in the algorithm are to be performed in the polynomial ring $F[x]$. So in the above example, if $F = \mathbf{Z}_{13}$, then $d(x) = x^2+2$, if $F = \mathbf{Z}_7$, then $d(x) = 1$; and if $F = \mathbf{Z}_2$, then the loop stops one step earlier and $d(x) = x^3 + x^2 + x + 1$.

Along with the polynomial gcd we shall need a polynomial inverse. In keeping with the notion of integer inverse, we shall generate a solution to

$$s(x)f(x) + t(x)g(x) = d(x),$$

for given f, g, where $d(x) = \gcd(f(x), g(x))$.

Algorithm 2.2.2 (Extended gcd for polynomials). Let F be a field. For given polynomials $f(x), g(x)$ in $F[x]$, not both zero, with either $\deg f(x) \geq \deg g(x)$ or $g(x) = 0$, this algorithm returns $(s(x), t(x), d(x))$ in $F[x]$ such that $d = \gcd(f, g)$ and $sg + th = d$. (For ease of notation we shall drop the x argument in what follows.)

```
1. [Initialize]
      $(s, t, d, u, v, w) = (1, 0, f, 0, 1, g)$;
2. [Extended Euclid loop]
      while($w \neq 0$) {
         $q = (d - (d \bmod w))/w$;                    // $q$ is the quotient of $d \div w$.
         $(s, t, d, u, v, w) = (u, v, w, s - qu, t - qv, d - qw)$;
      }
3. [Make monic]
      Set $c$ as the leading coefficient of $d$;
      $(s, t, d) = (c^{-1}s, c^{-1}t, c^{-1}d)$;
      return $(s, t, d)$;
```

If $d(x) = 1$ and neither of $f(x), g(x)$ is 0, then $s(x)$ is the inverse of $f(x)$ (mod $g(x)$) and $t(x)$ is the inverse of $g(x)$ (mod $f(x)$). It is clear that if naive polynomial remaindering is used, as described above, then the complexity of the algorithm is $O(D^2)$ field operations, where D is the larger of the degrees of the input polynomials; see [Menezes et al. 1997].

2.2.2 Finite fields

Examples of infinite fields are the rational numbers $\mathbf{Q}$, the real numbers $\mathbf{R}$, and the complex numbers $\mathbf{C}$. In this book, however, we are primarily concerned with finite fields. A common example: If p is prime, the field

$$\mathbf{F}_p = \mathbf{Z}_p$$

consists of all residues $0, 1, \ldots, p-1$ with arithmetic proceeding under the usual modular rules.

Given a field F and a polynomial $f(x)$ in $F[x]$ of positive degree, we may consider the quotient ring $F[x]/(f(x))$. The elements of $F[x]/(f(x))$ are subsets of $F[x]$ of the form $\{g(x) + f(x)h(x) : h(x) \in F[x]\}$; we denote this subset by $g(x) + (f(x))$. It is a coset of the ideal $(f(x))$ with coset representative $g(x)$. (Actually, *any* polynomial in a coset can stand in as a representative for the coset, so that $g(x) + (f(x)) = G(x) + (f(x))$ if and only if $G(x) \in g(x) + (f(x))$ if and only if $G(x) - g(x) = f(x)h(x)$ for some

$h(x) \in F[x]$ if and only if $G(x) \equiv g(x) \pmod{f(x)}$. Thus, working with cosets can be thought of as a fancy way of working with congruences.) Each coset has a canonical representative, that is, a unique and natural choice, which is either 0 or has degree smaller than $\deg f(x)$.

We can add and multiply cosets by doing the same with their representatives:

$$\begin{aligned} \big(g_1(x) + (f(x))\big) \;+\; \big(g_2(x) + (f(x))\big) &= g_1(x) + g_2(x) \;+\; (f(x)), \\ \big(g_1(x) + (f(x))\big) \;\cdot\; \big(g_2(x) + (f(x))\big) &= g_1(x)g_2(x) \;+\; (f(x)). \end{aligned}$$

With these rules for addition and multiplication, $F[x]/(f(x))$ is a ring that contains an isomorphic copy of the field F: An element $a \in F$ is identified with the coset $a + (f(x))$.

Theorem 2.2.3. *If F is a field and $f(x) \in F[x]$ has positive degree, then $F[x]/(f(x))$ is a field if and only if $f(x)$ is irreducible in $F[x]$.*

Via this theorem we can create new fields out of old fields. For example, starting with $\mathbf{Q}$, the field of rational numbers, consider the irreducible polynomial $x^2 - 2$ in $\mathbf{Q}[x]$. Let us denote the coset $a + bx + (f(x))$, where $a, b \in \mathbf{Q}$, more simply by $a + bx$. We have the addition and multiplication rules

$$\begin{aligned} (a_1 + b_1x) + (a_2 + b_2x) &= (a_1 + a_2) + (b_1 + b_2)x, \\ (a_1 + b_1x) \cdot (a_2 + b_2x) &= (a_1a_2 + 2b_1b_2) + (a_1b_2 + a_2b_1)x. \end{aligned}$$

That is, one performs ordinary addition and multiplication of polynomials, except that the relation $x^2 = 2$ is used for reduction. We have “created” the field

$$\mathbf{Q}\left[\sqrt{2}\right] = \left\{a + b\sqrt{2} : a, b \in \mathbf{Q}\right\}.$$

Let us try this idea starting from the finite field $\mathbf{F}_7$. Say we take $f(x) = x^2 + 1$. A degree-2 polynomial is irreducible over a field F if and only if it has no roots in F. A quick check shows that $x^2 + 1$ has no roots in $\mathbf{F}_7$, so it is irreducible over this field. Thus, by Theorem 2.2.3, $\mathbf{F}_7[x]/(x^2 + 1)$ is a field. We can abbreviate elements by $a + bi$, where $a, b \in \mathbf{F}_7$ and $i^2 = -1$. Our new field has 49 elements.

More generally, if p is prime and $f(x) \in \mathbf{F}_p[x]$ is irreducible and has degree $d \geq 1$, then $\mathbf{F}_p[x]/(f(x))$ is again a finite field, and it has p^d elements. Interestingly, *all* finite fields up to isomorphism can be constructed in this manner.

An important difference between finite fields and fields such as $\mathbf{Q}$ and $\mathbf{C}$ is that repeatedly adding 1 to itself in a finite field, you will eventually get 0. In fact, the number of times must be a prime, for otherwise, one can get the product of two nonzero elements being 0.

Definition 2.2.4. The characteristic of a field is the additive order of 1, unless said order is infinite, in which case the characteristic is 0.

As indicated above, the characteristic of a field, if it is positive, must be a prime number. Fields of characteristic 2 play a special role in applications, mainly because of the simplicity of doing arithmetic in such fields.

We collect some relevant classical results on finite fields as follows:

Theorem 2.2.5 (Basic results on finite fields).

(1) *A finite field F has nonzero characteristic, which must be a prime.*

(2) *For a, b in a finite field F of characteristic p, $(a+b)^p = a^p + b^p$.*

(3) *Every finite field has p^k elements for some positive integer k, where p is the characteristic.*

(4) *For given prime p and exponent k, there is exactly one field with p^k elements (up to isomorphism), which field we denote by $\mathbf{F}_{p^k}$.*

(5) *$\mathbf{F}_{p^k}$ contains as subfields unique copies of $\mathbf{F}_{p^j}$ for each $j|k$, and no other subfields.*

(6) *The multiplicative group $\mathbf{F}_{p^k}^*$ of nonzero elements in $\mathbf{F}_{p^k}$ is cyclic; that is, there is a single element whose powers constitute the whole group.*

The multiplicative group $\mathbf{F}_{p^k}^*$ is an important concept in studies of powers, roots, and cryptography.

Definition 2.2.6. A primitive root of a field $\mathbf{F}_{p^k}$ is an element whose powers constitute all of $\mathbf{F}_{p^k}^*$. That is, the root is a generator of the cyclic group $\mathbf{F}_{p^k}^*$.

For example, in the example above where we created a field with 49 elements, namely $\mathbf{F}_{7^2}$, the element $3+i$ is a primitive root.

A cyclic group with n elements has $\varphi(n)$ generators in total, where φ is the Euler totient function. Thus, a finite field $\mathbf{F}_{p^k}$ has $\varphi(p^k-1)$ primitive roots.

One way to detect primitive roots is to use the following result.

Theorem 2.2.7 (Test for primitive root). *An element g in $\mathbf{F}_{p^k}^*$ is a primitive root if and only if*

$$g^{(p^k-1)/q} \neq 1$$

for every prime q dividing p^k-1.

As long as p^k-1 can be factored, this test provides an efficient means of establishing a primitive root. A simple algorithm, then, for finding a primitive root is this: Choose random $g \in \mathbf{F}_{p^k}^*$, compute powers $g^{(p^k-1)/q} \bmod p$ for successive prime factors q of p^k-1, and if any one of these powers is 1, choose another g. If g survives the chain of powers, it is a primitive root by Theorem 2.2.7.

Much of this book is concerned with arithmetic in $\mathbf{F}_p$, but at times we shall have occasion to consider higher prime-power fields. Though general $\mathbf{F}_{p^k}$ arithmetic can be complicated, it is intriguing that some algorithms can actually enjoy improved performance when we invoke such higher fields. As we saw above, we can "create" the finite field $\mathbf{F}_{p^k}$ by coming up with an irreducible polynomial $f(x)$ in $\mathbf{F}_p[x]$ of degree k. We thus say a little about how one might do this.

Every element a in $\mathbf{F}_{p^k}$ has the property that $a^{p^k} = a$, that is, a is a root of $x^{p^k} - x$. In fact this polynomial splits into linear factors over $\mathbf{F}_{p^k}$ with no repeated factors. We can use this idea to see that $x^{p^k} - x$ is the product of *all* monic irreducible polynomials in $\mathbf{F}_p[x]$ of degrees dividing k. From this we get a formula for the number $N_k(p)$ of monic irreducible polynomials in $\mathbf{F}_p[x]$ of exact degree k: One begins with the identity

$$\sum_{d|k} dN_d(p) = p^k,$$

on which we can use Möbius inversion to get

$$N_k(p) = \frac{1}{k}\sum_{d|k} p^d \mu(k/d). \tag{2.5}$$

Here, μ is the Möbius function discussed in Section 1.4.1. It is easy to see that the last sum is dominated by the term $d = k$, so that $N_k(p)$ is approximately p^k/k. That is, about 1 out of every k monic polynomials of degree k in $\mathbf{F}_p[x]$ is irreducible. Thus a random search for one of these should be successful in $O(k)$ trials. But how can we recognize an irreducible polynomial? An answer is afforded by the following result.

Theorem 2.2.8. *Suppose that $f(x)$ is a polynomial in $\mathbf{F}_p[x]$ of positive degree k. The following statements are equivalent:*

(1) *$f(x)$ is irreducible;*

(2) $\gcd(f(x), x^{p^j} - x) = 1$ *for each $j = 1, 2, \dots, \lfloor k/2 \rfloor$;*

(3) $x^{p^k} \equiv x \pmod{f(x)}$ *and* $\gcd(f(x), x^{p^{k/q}} - x) = 1$ *for each prime $q|k$.*

This theorem, whose proof is left as Exercise 2.15, is then what is behind the following two irreducibility tests.

Algorithm 2.2.9 (Irreducibility test 1). Given prime p and a polynomial $f(x) \in \mathbf{F}_p[x]$ of degree $k \geq 2$, this algorithm determines whether $f(x)$ is irreducible over $\mathbf{F}_p$.

```
1. [Initialize]
       $g(x) = x$;
2. [Testing loop]
       for($1 \le i \le \lfloor k/2 \rfloor$) {
           $g(x) = g(x)^p \bmod f(x)$;              // Powering by Algorithm 2.1.5.
           $d(x) = \gcd(f(x), g(x) - x)$;     // Polynomial gcd by Algorithm 2.2.1.
           if($d(x) \ne 1$) return NO;
       }
       return YES;                                          // $f$ is irreducible.
```

Algorithm 2.2.10 (Irreducibility test 2). Given a prime p, a polynomial $f(x) \in \mathbf{F}_p[x]$ of degree $k \geq 2$, and the distinct primes $q_1 > q_2 > \ldots > q_l$ which divide k, this algorithm determines whether $f(x)$ is irreducible over $\mathbf{F}_p$.

```
1. [Initialize]
      $q_{l+1} = 1$;
      $g(x) = x^{p^{k/q_1}} \bmod f(x)$;                // Powering by Algorithm 2.1.5.
2. [Testing loop]
      for($1 \leq i \leq l$) {
         $d(x) = \gcd(f(x), g(x) - x)$;     // Polynomial gcd by Algorithm 2.2.1.
         if($d(x) \neq 1$) return NO;
         $g(x) = g(x)^{p^{k/q_{i+1}} - p^{k/q_i}} \bmod f(x)$;  // Powering by Algorithm 2.1.5.
      }
3. [Final congruence]
      if($g(x) \neq x$) return NO;
      return YES;                                        // $f$ is irreducible.
```

Using the naive arithmetic subroutines of this chapter, Algorithm 2.2.9 is slower than Algorithm 2.2.10 for large values of k, given the much larger number of gcd's which must be computed in the former algorithm. However, using a more sophisticated method for polynomial gcd's, (see [von zur Gathen and Gerhard 1999, Sec. 11.1]), the two methods are roughly comparable in time.

Let us now recapitulate the manner of field computations. Armed with a suitable irreducible polynomial f of degree k over $\mathbf{F}_p$, one represents any element $a \in \mathbf{F}_{p^k}$ as

$$a = a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1},$$

with each $a_i \in \{0, \ldots, p-1\}$. That is, we represent a as a vector in $\mathbf{F}_p^k$. Note that there are clearly p^k such vectors. Addition is ordinary vector addition, but of course the arithmetic in each coordinate is modulo p. Multiplication is more complicated: We view it merely as multiplication of polynomials, but not only is the coordinate arithmetic modulo p, but we also reduce high-degree polynomials modulo $f(x)$. That is to say, to multiply $a * b$ in $\mathbf{F}_{p^k}$, we simply form a polynomial product $a(x)b(x)$, doing a mod p reduction when a coefficient during this process exceeds $p-1$, then taking this product mod $f(x)$ via polynomial mod, again reducing mod p whenever appropriate during that process. In principle, one could just form the unrestricted product $a(x)b(x)$, do a mod f reduction, then take a final mod p reduction, in which case the final result would be the same but the interior integer multiplies might run out of control, especially if there were many polynomials being multiplied. It is best to take a reduction modulo p at every meaningful juncture.

Here is an example for explicit construction of a field of characteristic 2, namely $\mathbf{F}_{16}$. According to our formula (2.5), there are exactly 3 irreducible degree-4 polynomials in $\mathbf{F}_2[x]$, and a quick check shows that they are x^4+x+1, $x^4 + x^3 + 1$, and $x^4 + x^3 + x^2 + x + 1$. Though each of these can be used to create $\mathbf{F}_{16}$, the first has the pleasant property that reduction of high powers of x to lower powers is particularly simple: The mod $f(x)$ reduction is realized through the simple rule $x^4 = x + 1$ (recall that we are in characteristic 2, so that $1 = -1$). We may abbreviate typical field elements $a_0 + a_1x + a_2x^2 + a_3x^3$, where each $a_i \in \{0, 1\}$ by the binary string $(a_0a_1a_2a_3)$. We add componentwise modulo 2, which amounts to an “exclusive-or” operation, for example

$$(0111) + (1011) = (1100).$$

To multiply $a * b = (0111) * (1011)$ we can simulate the polynomial multiplication by doing a convolution on the coordinates, first getting (0110001), a string of length 7. (Calling this $(c_0c_1c_2c_3c_4c_5c_6)$ we have $c_j = \sum_{i_1+i_2=j} a_{i_1}b_{i_2}$, where the sum is over pairs i_1, i_2 of integers in $\{0, 1, 2, 3\}$ with sum j.) To get the final answer, we take any 1 in places $6, 5, 4$, in this order, and replace them via the modulo $f(x)$ relation. In our case, the 1 in place 6 gets replaced with 1’s in places 2 and 3, and doing the exclusive-or, we get (0101000). There are no more high-order 1’s to replace, and our product is (0101); that is, we have

$$(0111) * (1011) = (0101).$$

Though this is only a small example, all the basic notions of general field arithmetic via polynomials are present.

2.3 Squares and roots

2.3.1 Quadratic residues

We start with some definitions.

Definition 2.3.1. For coprime integers $m,\ a$ with m positive, we say that a is a quadratic residue (mod m) if and only if the congruence

$$x^2 \equiv a \pmod{m}$$

is solvable for integer x. If the congruence is not so solvable, a is said to be a quadratic nonresidue (mod m).

Note that quadratic residues and nonresidues are defined only when $\gcd(a, m) = 1$. So, for example, 0 (mod m) is always a square but is neither a quadratic residue nor a nonresidue. Another example is 3 (mod 9). This residue is not a square, but it is *not* considered a quadratic nonresidue since 3 and 9 are not coprime. When the modulus is prime the only non-coprime case is the 0 residue, which is one of the choices in the next definition.

Definition 2.3.2. For odd prime p, the Legendre symbol $\left(\frac{a}{p}\right)$ is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{if } a \equiv 0 \pmod{p}, \\ 1, & \text{if } a \text{ is a quadratic residue } \pmod{p}, \\ -1, & \text{if } a \text{ is a quadratic nonresidue } \pmod{p}. \end{cases}$$

Thus, the Legendre symbol signifies whether or not $a \not\equiv 0 \pmod{p}$ is a square $\pmod{p}$. Closely related, but differing in some important ways, is the Jacobi symbol:

Definition 2.3.3. For odd natural number m (whether prime or not), and for any integer a, the Jacobi symbol $\left(\frac{a}{m}\right)$ is defined in terms of the (unique) prime factorization

$$m = \prod p_i^{t_i}$$

as

$$\left(\frac{a}{m}\right) = \prod \left(\frac{a}{p_i}\right)^{t_i},$$

where $\left(\frac{a}{p_i}\right)$ are Legendre symbols, with $\left(\frac{a}{1}\right) = 1$ understood.

Note, then, that the function $\chi(a) = \left(\frac{a}{m}\right)$, defined for all integers a, is a character modulo m; see Section 1.4.3. It is important to note right off that for composite, odd m, a Jacobi symbol $\left(\frac{a}{m}\right)$ can sometimes be $+1$ when $x^2 \equiv a \pmod{m}$ is unsolvable. An example is

$$\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right)\left(\frac{2}{5}\right) = (-1)(-1) = 1,$$

even though 2 is not, in fact, a square modulo 15. However, if $\left(\frac{a}{m}\right) = -1$, then a is coprime to m and the congruence $x^2 \equiv a \pmod{m}$ is not solvable. And $\left(\frac{a}{m}\right) = 0$ if and only if $\gcd(a, m) > 1$.

It is clear that in principle the symbol $\left(\frac{a}{m}\right)$ is computable: One factors m into primes, and then computes each underlying Legendre symbol by exhausting all possibilities to see whether the congruence $x^2 \equiv a \pmod{p}$ is solvable. What makes Legendre and Jacobi symbols so very useful, though, is that they are indeed very easy to compute, with no factorization or primality test necessary, and with no exhaustive search. The following theorem gives some of the beautiful properties of Legendre and Jacobi symbols, properties that make their evaluation a simple task, about as hard as taking a gcd.

Theorem 2.3.4 (Relations for Legendre and Jacobi symbols). *Let p denote an odd prime, let m, n denote arbitrary positive odd integers (including possibly primes), and let a, b denote integers. Then we have the Euler test for quadratic residues modulo primes, namely*

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}. \tag{2.6}$$

We have the multiplicative relations

$$\left(\frac{ab}{m}\right) = \left(\frac{a}{m}\right)\left(\frac{b}{m}\right), \tag{2.7}$$

$$\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right) \tag{2.8}$$

and special relations

$$\left(\frac{-1}{m}\right) = (-1)^{(m-1)/2}, \tag{2.9}$$

$$\left(\frac{2}{m}\right) = (-1)^{(m^2-1)/8}. \tag{2.10}$$

Furthermore, we have the law of quadratic reciprocity for coprime m, n:

$$\left(\frac{m}{n}\right)\left(\frac{n}{m}\right) = (-1)^{(m-1)(n-1)/4}. \tag{2.11}$$

Already (2.6) shows that when $|a| < p$, the Legendre symbol $\left(\frac{a}{p}\right)$ can be computed in $O\left(\ln^3 p\right)$ bit operations using naive arithmetic and Algorithm 2.1.5; see Exercise 2.17. But we can do better, and we do not even need to recognize primes.

Algorithm 2.3.5 (Calculation of Legendre/Jacobi symbol). Given positive odd integer m, and integer a, this algorithm returns the Jacobi symbol $\left(\frac{a}{m}\right)$, which for m an odd prime is also the Legendre symbol.

```
1. [Reduction loops]
    $a = a \bmod m$;
    $t = 1$;
    while($a \neq 0$) {
        while($a$ even) {
            $a = a/2$;
            if($m \bmod 8 \in \{3, 5\}$) $t = -t$;
        }
        $(a, m) = (m, a)$;                        // Swap variables.
        if($a \equiv m \equiv 3 \pmod 4$) $t = -t$;
        $a = a \bmod m$;
    }
2. [Termination]
    if($m == 1$) return $t$;
    return $0$;
```

It is clear that this algorithm does not take materially longer than using Algorithm 2.1.2 to find $\gcd(a, m)$, and so runs in $O\left(\ln^2 m\right)$ bit operations when $|a| < m$.

In various other sections of this book we make use of a celebrated connection between the Legendre symbol and exponential sums. The study of this connection runs deep; for the moment we state one central, useful result, starting with the following definition:

Definition 2.3.6. The quadratic Gauss sum $G(a;m)$ is defined for integers a, N, with N positive, as

$$G(a;N) = \sum_{j=0}^{N-1} e^{2\pi iaj^2/N}.$$

This sum is—up to conjugation perhaps—a discrete Fourier transform (DFT) as used in various guises in Chapter 8.8. A more general form—a character sum—is used in primality proving (Section 4.4). The central result we wish to cite makes an important connection with the Legendre symbol:

Theorem 2.3.7 (Gauss). *For odd prime p and integer $a \not\equiv 0 \pmod p$,*

$$G(a;p) = \left(\frac{a}{p}\right) G(1;p),$$

and generally, for positive integer m,

$$G(1;m) = \frac{1}{2}\sqrt{m}(1+i)(1+(-i)^m).$$

The first assertion is really very easy, the reader might consider proving it without looking up references. The two assertions of the theorem together allow for Fourier inversion of the sum, so that one can actually express the Legendre symbol for $a \not\equiv 0 \pmod p$ by

$$\left(\frac{a}{p}\right) = \frac{c}{\sqrt{p}} \sum_{j=0}^{p-1} e^{2\pi iaj^2/p} = \frac{c}{\sqrt{p}} \sum_{j=0}^{p-1} \left(\frac{j}{p}\right) e^{2\pi iaj/p}, \tag{2.12}$$

where $c = 1, -i$ as $p \equiv 1, 3 \pmod 4$, respectively. This shows that the Legendre symbol is, essentially, its own discrete Fourier transform (DFT). For practice in manipulating Gauss sums, see Exercises 1.66, 2.27, 2.28, and 9.41.

2.3.2 Square roots

Armed now with algorithms for gcd, inverse (actually the -1 power), and positive integer powers, we turn to the issue of square roots modulo a prime. As we shall see, the technique actually calls for raising residues to high integral powers, and so the task is not at all like taking square roots in the real numbers.

We have seen that for odd prime p, the solvability of a congruence

$$x^2 \equiv a \not\equiv 0 \pmod p$$

is signified by the value of the Legendre symbol $\left(\frac{a}{p}\right)$. When $\left(\frac{a}{p}\right) = 1$, an important problem is to find a "square root" x, of which there will be two, one the other's negative (mod p). We shall give two algorithms for extracting such square roots, both computationally efficient but raising different issues of implementation.

The first algorithm starts from Euler's test (2.6). If the prime p is 3 (mod 4) and $\left(\frac{a}{p}\right) = 1$, then Euler's test says that $a^t \equiv 1 \pmod{p}$, where $t = (p-1)/2$. Then $a^{t+1} \equiv a \pmod{p}$, and as $t+1$ is even in this case, we may take for our square root $x \equiv a^{(t+1)/2} \pmod{p}$. Surely, this delightfully simple solution to the square root problem can be generalized! Yes, but it is not so easy. In general, we may write $p - 1 = 2^s t$, with t odd. Euler's test (2.6) guarantees us that $a^{2^{s-1}t} \equiv 1 \pmod{p}$, but it does not appear to say anything about $A = a^t \pmod{p}$.

Well, it does say something; it says that the multiplicative order of A modulo p is a divisor of 2^{s-1}. Suppose that d is a quadratic *nonresidue* modulo p, and let $D = d^t \bmod p$. Then Euler's test (2.6) says that the multiplicative order of D modulo p is exactly 2^s, since $D^{2^{s-1}} \equiv -1 \pmod{p}$. The same is true about $D^{-1} \pmod{p}$, namely, its multiplicative order is 2^s. Since the multiplicative group $\mathbf{Z}_p^*$ is cyclic, it follows that A is in the cyclic subgroup generated by D^{-1}, and in fact, A is an even power of D^{-1}, that is, $A \equiv D^{-2\mu} \pmod{p}$ for some integer μ with $0 \le \mu < 2^{s-1}$. Substituting for A we have $a^t D^{2\mu} \equiv 1 \pmod{p}$. Then after multiplying this congruence by a, the left side has all even exponents, and we can extract the square root of a modulo p as $a^{(t+1)/2} D^{\mu} \pmod{p}$.

To make this idea into an algorithm, there are two problems that must be solved:

(1) Find a quadratic nonresidue $d \pmod{p}$.

(2) Find an integer μ with $A \equiv D^{-2\mu} \pmod{p}$.

It might seem that problem (1) is simple and that problem (2) is difficult, since there are many quadratic nonresidues modulo p and we only need one of them, any one, while for problem (2) there is a specific integer μ that we are searching for. In some sense, these thoughts are correct. However, we know no rigorous, *deterministic* way to find a quadratic nonresidue quickly. We will get around this impasse by using a *random* algorithm. And though problem (2) is an instance of the notoriously difficult discrete logarithm problem (see Chapter 5), the particular instance we have in hand here is simple. The following algorithm is due to A. Tonelli in 1891, based on earlier work of Gauss.

Algorithm 2.3.8 (Square roots (mod p)). Given an odd prime p and an integer a with $\left(\frac{a}{p}\right) = 1$, this algorithm returns a solution x to $x^2 \equiv a \pmod{p}$.

```
1. [Check simplest cases: $p \equiv 3, 5, 7 \pmod{8}$]
        $a = a \bmod p$;
        if($p \equiv 3, 7 \pmod{8}$) {
            $x = a^{(p+1)/4} \bmod p$;
            return $x$;
        }
        if($p \equiv 5 \pmod{8}$) {
```

```
        $x = a^{(p+3)/8} \bmod p$;
        $c = x^2 \bmod p$;                                  // Then $c \equiv \pm a \pmod p$.
        if($c \neq a \bmod p$) $x = x2^{(p-1)/4} \bmod p$;
        return $x$;
    }
2. [Case $p \equiv 1 \pmod 8$]
    Find a random integer $d \in [2, p-1]$ with $\left(\frac{d}{p}\right) = -1$;
                                    // Compute Jacobi symbols via Algorithm 2.3.5.
    Represent $p - 1 = 2^s t$, with $t$ odd;
    $A = a^t \bmod p$;
    $D = d^t \bmod p$;
    $m = 0$;                                        // $m$ will be $2\mu$ of text discussion.
    for($0 \le i < s$){                             // One may start at $i = 1$; see text.
        if($(AD^m)^{2^{s-1-i}} \equiv -1 \pmod p$) $m = m + 2^i$;
    }                                               // Now we have $AD^m \equiv 1 \pmod p$.
    $x = a^{(t+1)/2} D^{m/2} \bmod p$;
    return $x$;
```

Note the following interesting features of this algorithm. First, it turns out that the $p \equiv 1 \pmod 8$ branch—the hardest case—will actually handle all the cases. (We have essentially used in the $p \equiv 5 \pmod 8$ case that we may choose $d = 2$. And in the $p \equiv 3 \pmod 4$ cases, the exponent m is 0, so we do not need a value of d.) Second, notice that built into the algorithm is the check that $A^{2^{s-1}} \equiv 1 \pmod p$, which is what ensures that m is even. If this fails, then we do not have $\left(\frac{a}{p}\right) = 1$, and so the algorithm may be amended to leave out this requirement, with a break called for if the case $i = 0$ in the loop produces the residue -1. If one is taking many square roots of residues a for which it is unknown whether a is a quadratic residue or nonresidue, then one may be tempted to just let Algorithm 2.3.8 decide the issue for us. However, if nonresidues occur a positive fraction of the time, it will be faster on average to first run Algorithm 2.3.5 to check the quadratic character of a, and thus avoid running the more expensive Algorithm 2.3.8 on the nonresidues.

As we have mentioned, there is no known deterministic, polynomial time algorithm for finding a quadratic nonresidue d for the prime p. However, if one assumes the ERH, it can be shown there is a quadratic nonresidue $d < 2\ln^2 p$; see Theorem 1.4.5, and so an exhaustive search to this limit succeeds in finding a quadratic nonresidue in polynomial time. Thus, on the ERH, one can find square roots for quadratic residues modulo the prime p in deterministic, polynomial time. It is interesting, from a theoretical standpoint, that for *a fixed*, R. Schoof has a rigorously proved, deterministic, polynomial time algorithm for square root extraction; see [Schoof 1985]. (The bit complexity is polynomial in the length of p, but exponential in the length of a, so that for a fixed it is correct to say that the algorithm is polynomial time.) Still, in spite of this fascinating theoretical state of affairs, the fact that half of all nonzero residues $d \pmod p$ satisfy $\left(\frac{d}{p}\right) = -1$ leads to the expectation of only a few random attempts to find a suitable d. In fact, the expected number of random attempts is 2.

The complexity of Algorithm 2.3.8 is dominated by the various exponentiations called for, and so is $O(s^2 + \ln t)$ modular operations. Assuming naive arithmetic subroutines, this comes out to, in the worst case (when s is large), $O\left(\ln^4 p\right)$ bit operations. However, if one is applying Algorithm 2.3.8 to many prime moduli p, it is perhaps better to consider its average case, which is just $O\left(\ln^3 p\right)$ bit operations. This is because there are very few primes p with $p-1$ divisible by a large power of 2.

The following algorithm is asymptotically faster than the worst case of Algorithm 2.3.8. A beautiful application of arithmetic in the finite field $\mathbf{F}_{p^2}$, the method is a 1907 discovery of M. Cipolla.

Algorithm 2.3.9 (Square roots (mod p) via $\mathbf{F}_{p^2}$ arithmetic). Given an odd prime p and a quadratic residue a modulo p, this algorithm returns a solution x to $x^2 \equiv a \pmod p$.

```
1. [Find a certain quadratic nonresidue]
     Find a random integer $t \in [0, p-1]$ such that $\left(\frac{t^2-a}{p}\right) = -1$;
                                  // Compute Jacobi symbols via Algorithm 2.3.5.
2. [Find a square root in $\mathbf{F}_{p^2} = \mathbf{F}_p(\sqrt{t^2-a})$ ]
     $x = (t + \sqrt{t^2-a})^{(p+1)/2}$;                  // Use $\mathbf{F}_{p^2}$ arithmetic.
     return $x$;
```

The probability that a random value of t will be successful in Step [Find a certain quadratic nonresidue] is $(p-1)/2p$. It is not hard to show that the element $x \in \mathbf{F}_{p^2}$ is actually an element of the subfield $\mathbf{F}_p$ of $\mathbf{F}_{p^2}$, and that $x^2 \equiv a \pmod p$. (In fact, the second assertion forces x to be in $\mathbf{F}_p$, since a has the same square roots in $\mathbf{F}_p$ as it has in the larger field $\mathbf{F}_{p^2}$.)

A word is in order on the field arithmetic, which for this case of $\mathbf{F}_{p^2}$ is especially simple, as might be expected on the basis of Section 2.2.2. Let $\omega = \sqrt{t^2-a}$. Representing this field by

$$\mathbf{F}_{p^2} = \{x + \omega y : x, y \in \mathbf{F}_p\} = \{(x, y)\},$$

all arithmetic may proceed using the rule

$$\begin{aligned} (x, y) * (u, v) &= (x + y\omega)(u + v\omega) \\ &= xu + yv\omega^2 + (xv + yu)\omega \\ &= (xu + yv(t^2-a), xv + yu), \end{aligned}$$

noting that $\omega^2 = t^2 - a$ is in $\mathbf{F}_p$. Of course, we view x, y, u, v, t, a as residues modulo p and the above expressions are always reduced to this modulus. Any of the binary ladder powering algorithms in this book may be used for the computation of x in step [Find a square root ...]. An equivalent algorithm for square roots is given in [Menezes et al. 1997], in which one finds a quadratic nonresidue $b^2 - 4a$, defines the polynomial $f(x) = x^2 - bx + a$ in $\mathbf{F}_p[x]$, and simply computes the desired root $r = x^{(p+1)/2} \bmod f$ (using polynomial-mod operations). Note finally that the special cases $p \equiv 3, 5, 7 \pmod 8$ can also be ferreted out of any of these algorithms, as was done in Algorithm 2.3.8, to improve average performance.

The complexity of Algorithm 2.3.9 is $O(\ln^3 p)$ bit operations (assuming naive arithmetic), which is asymptotically better than the worst case of Algorithm 2.3.8. However, if one is loath to implement the modified powering ladder for the $\mathbf{F}_{p^2}$ arithmetic, the asymptotically slower algorithm will usually serve. Incidentally, there is yet another, equivalent, approach for square rooting by way of Lucas sequences (see Exercise 2.31).

It is very interesting to note at this juncture that there is no known fast method of computing square roots of quadratic residues for general composite moduli. In fact, as we shall see later, doing so is essentially equivalent to factoring the modulus (see Exercise 6.5).

2.3.3 Finding polynomial roots

Having discussed issues of existence and calculation of square roots, we now consider the calculation of roots of a polynomial of arbitrary degree over a finite field. We specify the finite field as $\mathbf{F}_p$, but much of what we say generalizes to an arbitrary finite field.

Let $g \in \mathbf{F}_p[x]$ be a polynomial; that is, it is a polynomial with integer coefficients reduced (mod p). We are looking for the roots of g in $\mathbf{F}_p$, and so we might begin by replacing $g(x)$ with the gcd of $g(x)$ and $x^p - x$, since as we have seen, the latter polynomial is the product of $x - a$ as a runs over all elements of $\mathbf{F}_p$. If $p > \deg g$, one should first compute $x^p \bmod g(x)$ via Algorithm 2.1.5. If the gcd has degree not exceeding 2, the prior methods we have learned settle the matter. If it has degree greater than 2, then we take a further gcd with $(x+a)^{(p-1)/2} - 1$ for a random $a \in \mathbf{F}_p$. Any particular $b \neq 0$ in $\mathbf{F}_p$ is a root of $(x+a)^{(p-1)/2} - 1$ with probability 1/2, so that we have a positive probability of splitting $g(x)$ into two polynomials of smaller degree. This suggests a recursive algorithm, which is what we describe below.

Algorithm 2.3.10 (Roots of a polynomial over $\mathbf{F}_p$).
Given a nonzero polynomial $g \in \mathbf{F}_p[x]$, with p an odd prime, this algorithm returns the set r of the roots (without multiplicity) in $\mathbf{F}_p$ of g. The set r is assumed global, augmented as necessary during all recursive calls.

```
1. [Initial adjustments]
      $r = \{\ \}$;                              // Root list starts empty.
      $g(x) = \gcd(x^p - x, g(x))$;             // Using Algorithm 2.2.1.
      if($g(0) == 0$) {                          // Check for 0 root.
         $r = r \cup \{0\}$;
         $g(x) = g(x)/x$;
      }
2. [Call recursive procedure and return]
```

```
        $r = r \cup roots(g)$;
        return $r$;
3. [Recursive function $roots()$]
        $roots(g)$ {
            If $\deg(g) \le 2$, use quadratic (or lower) formula, via Algorithm 2.3.8, or
                2.3.9, to append to $r$ all roots of $g$, and return;
            while($h == 1$ or $h == g$) {                    // Random splitting.
                Choose random $a \in [0, p-1]$;
                $h(x) = \gcd((x+a)^{(p-1)/2} - 1, g(x))$;
            }
            $r = r \cup roots(h) \cup roots(g/h)$;
            return;
        }
```

The computation of $h(x)$ in the random-splitting loop can be made easier by using Algorithm 2.1.5 to first compute $(x+a)^{(p-1)/2} \bmod g(x)$ (and of course, the coefficients are always reduced (mod p)). It can be shown that the probability that a random a will succeed in splitting $g(x)$ (where $\deg(g) \ge 3$) is at least about 3/4 if p is large, and is always bounded above 0. Note that we can use the random splitting idea on degree-2 polynomials as well, and thus we have a third square root algorithm! (If $g(x)$ has degree 2, then the probability that a random choice for a in Step [Recursive ...] will split g is at least $(p-1)/(2p)$.) Various implementation details of this algorithm are discussed in [Cohen 2000]. Note that the algorithm is not actually factoring the polynomial; for example, a polynomial f might be the product of two irreducible polynomials, each of which is devoid of roots in $\mathbf{F}_p$. For actual polynomial factoring, there is the Berlekamp algorithm [Menezes et al. 1997], [Cohen 2000], but many important algorithms require only the root finding we have exhibited.

We now discuss the problem of finding roots of a polynomial to a composite modulus. Suppose the modulus is $n = ab$, where a, b are coprime. If we have an integer r with $f(r) \equiv 0 \pmod a$ and an integer s with $f(s) \equiv 0 \pmod b$, we can find a root to $f(x) \equiv 0 \pmod{ab}$ that "corresponds" to r and s. Namely, if the integer t simultaneously satisfies $t \equiv r \pmod a$ and $t \equiv s \pmod b$, then $f(t) \equiv 0 \pmod{ab}$. And such an integer t may be found by the Chinese remainder theorem; see Theorem 2.1.6. Thus, if the modulus n can be factored into primes, and we can solve the case for prime power moduli, then we can solve the general case.

To this end, we now turn our attention to solving polynomial congruences modulo prime powers. Note that for any polynomial $f(x) \in \mathbf{Z}[x]$ and any integer r, there is a polynomial $g_r(x) \in \mathbf{Z}[x]$ with

$$f(x+r) = f(r) + xf'(r) + x^2 g_r(x). \tag{2.13}$$

This can be seen either through the Taylor expansion for $f(x+r)$ or through the binomial theorem in the form

$$(x+r)^d = r^d + dr^{d-1}x + x^2 \sum_{j=2}^{d} \binom{d}{j} r^{d-j} x^{j-2}.$$

We can use Algorithm 2.3.10 to find solutions to $f(x) \equiv 0 \pmod{p}$, if there are any. The question is how we might be able to "lift" a solution to one modulo p^k for various exponents k. Suppose we have been successful in finding a root modulo p^i, say $f(r) \equiv 0 \pmod{p^i}$, and we wish to find a solution to $f(t) \equiv 0 \pmod{p^{i+1}}$ with $t \equiv r \pmod{p^i}$. We write t as $r + p^i y$, and so we wish to solve for y. We let $x = p^i y$ in (2.13). Thus

$$f(t) = f(r + p^i y) \equiv f(r) + p^i y f'(r) \pmod{p^{2i}}.$$

If the integer $f'(r)$ is not divisible by p, then we can use the methods of Section 2.1.1 to solve the congruence

$$f(r) + p^i y f'(r) \equiv 0 \pmod{p^{2i}},$$

namely by dividing through by p^i (recall that $f(r)$ is divisible by p^i), finding an inverse z for $f'(r) \pmod{p^i}$, and letting $y = -z f(r) p^{-i} \bmod p^i$. Thus, we have done more than we asked for, having instantly gone from the modulus p^i to the modulus p^{2i}. But there was a requirement that the integer r satisfy $f'(r) \not\equiv 0 \pmod{p}$. In general, if $f(r) \equiv f'(r) \equiv 0 \pmod{p}$, then there may be no integer $t \equiv r \pmod{p}$ with $f(t) \equiv 0 \pmod{p^2}$. For example, take $f(x) = x^2 + 3$ and consider the prime $p = 3$. We have the root $x = 0$; that is, $f(0) \equiv 0 \pmod{3}$. But the congruence $f(x) \equiv 0 \pmod{9}$ has no solution. For more on criteria for when a polynomial solution lifts to higher powers of the modulus, see Section 3.5.3 in [Cohen 2000].

The method described above is known as Hensel lifting, after the German mathematician K. Hensel. The argument essentially gives a criterion for there to be a solution of $f(x) = 0$ in the "p-adic" numbers: There is a solution if there is an integer r with $f(r) \equiv 0 \pmod{p}$ and $f'(r) \not\equiv 0 \pmod{p}$. What is more important for us, though, is using this idea as an algorithm to solve polynomial congruences modulo high powers of a prime. We summarize the above discussion in the following.

Algorithm 2.3.11 (Hensel lifting). We are given a polynomial $f(x) \in \mathbf{Z}[x]$, a prime p, and an integer r that satisfies $f(r) \equiv 0 \pmod{p}$ (perhaps supplied by Algorithm 2.3.10) and $f'(r) \not\equiv 0 \pmod{p}$. This algorithm describes how one constructs a sequence of integers $r_0, r_1, \ldots$ such that for each $i < j$, $r_i \equiv r_j \pmod{p^{2^j}}$ and $f(r_i) \equiv 0 \pmod{p^{2^i}}$. The description is iterative, that is, we give r_0 and show how to find r_{i+1} as a function of an already known r_i.

```
1. [Initial term]
        $r_0 = r$;
```

```
2. [Function $newr()$ that gives $r_{i+1}$ from $r_i$]
        $newr(r_i)$ {
            $x = f(r_i)p^{-2^i}$;
            $z = (f'(r))^{-1} \bmod p^{2^i}$;                    // Via Algorithm 2.1.4.
            $y = -xz \bmod p^{2^i}$;
            $r_{i+1} = r_i + yp^{2^i}$;
            return $r_{i+1}$;
        }
```

Note that for $j \geq i$ we have $r_j \equiv r_i \pmod{p^{2^i}}$, so that the sequence (r_i) converges in the p-adic numbers to a root of $f(x)$. In fact, Hensel lifting may be regarded as a p-adic version of the Newton methods discussed in Section 9.2.2.

2.3.4 Representation by quadratic forms

We next turn to a problem important to such applications as elliptic curves and primality testing. This is the problem of finding quadratic Diophantine representations, for positive integer d and odd prime p, in the form

$$x^2 + dy^2 = p,$$

or, in studies of complex quadratic orders of discriminant $D < 0$, $D \equiv 0, 1 \pmod 4$, the form [Cohen 2000]

$$x^2 + |D|y^2 = 4p.$$

There is a beautiful approach for these Diophantine problems. The next two algorithms are not only elegant, they are very efficient. Incidentally, the following algorithm was attributed usually to Cornacchia until recently, when it became known that H. Smith had discovered it earlier, in 1885 in fact.

Algorithm 2.3.12 (Represent p as $x^2 + dy^2$: Cornacchia–Smith method). Given an odd prime p and a positive integer d not divisible by p, this algorithm either reports that $p = x^2 + dy^2$ has no integral solution, or returns a solution.

```
1. [Test for solvability]
        if($\left(\frac{-d}{p}\right) == -1$) return { };         // Return empty: no solution.
2. [Initial square root]
        $x_0 = \sqrt{-d} \bmod p$;                     // Via Algorithm 2.3.8 or 2.3.9.
        if($2x_0 < p$) $x_0 = p - x_0$;                          // Justify the root.
3. [Initialize Euclid chain]
        $(a, b) = (p, x_0)$;
        $c = \lfloor \sqrt{p} \rfloor$;                              // Via Algorithm 9.2.11.
4. [Euclid chain]
        while($b > c$) $(a, b) = (b, a \bmod b)$;
5. [Final report]
```

```
    $t = p - b^2$;
    if($t \not\equiv 0 \pmod{d}$) return { };                    // Return empty.
    if($t/d$ not a square) return { };                           // Return empty.
    return $(\pm b, \pm\sqrt{t/d})$;                             // Solution(s) found.
```

This completely solves the computational Diophantine problem at hand. Note that an integer square-root finding routine (Algorithm 9.2.11) is invoked at two junctures. The second invocation—the determination as to whether t/d is a perfect square—can be done along the lines discussed in the text following the Algorithm 9.2.11 description. Incidentally, the proof that Algorithm 2.3.12 works is, in words from [Cohen 2000], "a little painful." There is an elegant argument, due to H. Lenstra, in [Schoof 1995], and a clear explanation from an algorist's point of view (for $d = 1$) in [Bressoud and Wagon 2000, p. 283].

The second case, namely for the Diophantine equation $x^2 + |D|y^2 = 4p$, for $D < 0$, can be handled in the following way [Cohen 2000]. First we observe that if $D \equiv 0 \pmod 4$, then x is even, whence the problem comes down to solving $(x/2)^2+(|D|/4)y^2 = p$, which we have already done. If $D \equiv 1 \pmod 8$, we have $x^2 - y^2 \equiv 4 \pmod 8$, and so x, y are both even, and again we defer to the previous method. Given the above argument, one could use the next algorithm only for $D \equiv 5 \pmod 8$, but in fact, the following will work for what turn out to be convenient cases $D \equiv 0, 1 \pmod 4$:

Algorithm 2.3.13. (Represent $4p$ as $x^2 + |D|y^2$ (modified Cornacchia–Smith)) Given a prime p and $-4p < D < 0$ with $D \equiv 0, 1 \pmod 4$, this algorithm either reports that no solution exists, or returns a solution (x, y).

```
1. [Case $p = 2$]
    if($p == 2$) {
        if($D + 8$ is a square) return $(\sqrt{D+8}, 1)$;
        return { };                                   // Return empty: no solution.
    }
2. [Test for solvability]
    if($\left(\frac{D}{p}\right) < 1$) return { };                    // Return empty.
3. [Initial square root]
    $x_0 = \sqrt{D} \bmod p$;                        // Via Algorithm 2.3.8 or 2.3.9.
    if($x_0 \not\equiv D \pmod 2$) $x_0 = p - x_0$;            // Ensure $x_0^2 \equiv D \pmod{4p}$.
4. [Initialize Euclid chain]
    $(a, b) = (2p, x_0)$;
    $c = \lfloor 2\sqrt{p} \rfloor$;                                  // Via Algorithm 9.2.11.
5. [Euclid chain]
    while($b > c$) $(a, b) = (b, a \bmod b)$;
6. [Final report]
    $t = 4p - b^2$;
    if($t \not\equiv 0 \pmod{|D|}$) return { };                  // Return empty.
    if($t/|D|$ not a square) return { };                         // Return empty.
    return $(\pm b, \pm\sqrt{t/|D|})$;                           // Found solution(s).
```

Again, the algorithm either says that there is no solution, or reports the essentially unique solution to $x^2 + |D|y^2 = 4p$.

2.4 Exercises

2.1. Prove that 16 is, modulo any odd number, an eighth power.

2.2. Show that the least common multiple $\mathrm{lcm}\,(a, b)$ satisfies

$$\mathrm{lcm}\,(a, b) = \frac{ab}{\gcd(a, b)},$$

and generalize this formula for more than two arguments. Then, using the prime number theorem (PNT), find a reasonable estimate for the lcm of all the integers from 1 through (a large) n.

2.3. Recall that $\omega(n)$ denotes the number of distinct prime factors of n. Prove that for any positive squarefree integer n,

$$\#\{(x, y)\ :\ x, y \text{ positive integers, } \mathrm{lcm}\,(x, y) = n\} = 3^{\omega(n)}.$$

2.4. Study the relation between the Euclid algorithm and simple continued fractions, with a view to proving the Lamé theorem (the first part of Theorem 2.1.3).

2.5. Fibonacci numbers are defined $u_0 = 0$, $u_1 = 1$, and $u_{n+1} = u_n + u_{n-1}$ for $n \geq 1$. Prove the remarkable relation

$$\gcd(u_a, u_b) = u_{\gcd(a,b)},$$

which shows, among many other things, that u_n, u_{n+1} are coprime for $n > 1$, and that if u_n is prime, then n is prime. Find a counterexample to the converse (find a prime p such that u_p is composite). By analyzing numerically several Fibonacci numbers, guess—then prove—a simple, general formula for the inverse of $u_n \pmod{u_{n+1}}$.

Fibonacci numbers appear elsewhere in this book, e.g., in Sections 1.3.3, 3.6.1 and Exercises 3.25, 3.41, 9.50.

2.6. Show that for $x \approx y \approx N$, and assuming classical divide with remainder, the bit-complexity of the classical Euclid algorithm is $O\left(\ln^2 N\right)$. It is helpful to observe that to find the quotient–remainder pair q, r with $x = qy+r$ requires $O((1 + \ln q) \ln x)$ bit operations, and that the quotients are constrained in a certain way during the Euclid loop.

2.7. Prove that Algorithm 2.1.4 works; that is, the correct gcd and inverse pair are returned. Answer the following question: When, if ever, do the returned a, b have to be reduced further, to $a \bmod y$ and $b \bmod x$, to yield legitimate, unique inverses?

2.8. Argue that for a naive application of Theorem 2.1.6 the mod operations involved consume at least $O\left(\ln^2 M\right)$ bit operations if arithmetic be done in grammar-school fashion, but only $O\left(r \ln^2 m\right)$ via Algorithm 2.1.7, where m denotes the maximum of the m_i.

2.9. Write a program to effect the asymptotically fast, preconditioned CRT Algorithm 9.5.26, and use this to multiply two numbers each of, say, 100 decimal digits, using sufficiently many small prime moduli.

2.10. Following Exercise 1.48 one can use, for CRT moduli, Mersenne numbers having pairwise coprime exponents (the Mersenne numbers need not themselves be prime). What computational advantages might there be in choosing such a moduli set (see Section 9.2.3)? Is there an easy way to find inverses $(2^a - 1)^{-1} \pmod{2^b - 1}$?

2.11. Give the computational complexity of the "straightforward inverse" algorithm implied by relation (2.3). Is there ever a situation when one should use this, or use instead Algorithm 2.1.4 to obtain $a^{-1} \bmod m$?

2.12. Let $N_k(p)$ be the number of monic irreducible polynomials in $\mathbf{F}_p[x]$ of degree k. Using formula (2.5), show that $p^k/k \ge N_k(p) > p^k/k - 2p^{k/2}/k$ for every prime p and every positive integer k. Show too that we always have $N_k(p) > 0$.

2.13. Does formula (2.5) generalize to give the number of irreducible polynomials of degree k in $\mathbf{F}_{p^n}[x]$?

2.14. Show how Algorithm 2.2.2 plays a role in finite field arithmetic, namely in the process of finding a multiplicative inverse of an element in $\mathbf{F}_{p^n}$.

2.15. Prove Theorem 2.2.8.

2.16. Show how Algorithms 2.3.8 and 2.3.9 may be appropriately generalized to find square roots of squares in the finite field $\mathbf{F}_{p^n}$.

2.17. By considering the binary expansion of the exponent n, show that the computational complexity of Algorithm 2.1.5 is $O(\ln n)$ operations. Argue that if x, n are each of size m and we are to compute $x^n \bmod m$, and classical multiply-mod is used, that the overall bit complexity of this powering grows as the cube of the number of bits in m.

2.18. Say we wish to compute a power $x^y \bmod N$, with $N = pq$, the product of two distinct primes. Describe an algorithm that combines a binary ladder and Chinese remainder theorem (CRT) ideas, and that yields the desired power more rapidly than does a standard, (mod N)-based ladder.

2.19. The "repunit" number $r_{1031} = (10^{1031} - 1)/9$, composed of 1031 decimal ones, is known to be prime. Determine, via reciprocity, which of $7, -7$ is a quadratic residue of this repunit. Then give an explicit square root (mod r_{1031}) of the quadratic residue.

2.20. Using appropriate results of Theorem 2.3.4, prove that for prime $p > 3$,

$$\left(\frac{-3}{p}\right) = (-1)^{\frac{(p-1) \bmod 6}{4}}.$$

Find a similar closed form for $\left(\frac{5}{p}\right)$ when $p \neq 2, 5$.

2.21. Show that for prime $p \equiv 1 \pmod 4$, the sum of the quadratic residues in $[1, p-1]$ is $p(p-1)/4$.

2.22. Show that if a is a nonsquare integer, then $\left(\frac{a}{p}\right) = -1$ for infinitely many primes p. (Hint: First assume that a is positive and odd. Show that there is an integer b such that $\left(\frac{b}{a}\right) = -1$ and $b \equiv 1 \pmod 4$. Then any positive integer $n \equiv b \pmod{4a}$ satisfies $\left(\frac{a}{n}\right) = -1$, and so is divisible by a prime p with $\left(\frac{a}{p}\right) = -1$. Show that infinitely many primes p arise in this way. Then deal with the cases when a is even or negative.)

2.23. Use Exercise 2.22 to show that if $f(x)$ is an irreducible quadratic polynomial in $\mathbf{Z}[x]$, then there are infinitely many primes p such that $f(x) \bmod p$ is irreducible in $\mathbf{Z}_p[x]$. Show that $x^4 + 1$ is irreducible in $\mathbf{Z}[x]$, but is reducible in each $\mathbf{Z}_p[x]$. What about cubic polynomials?

2.24. Develop an algorithm for computing the Jacobi symbol $\left(\frac{a}{m}\right)$ along the lines of the binary *gcd* method of Algorithm 9.4.2.

2.25. Prove: For prime p with $p \equiv 3 \pmod 4$, given any pair of square roots of a given $x \not\equiv 0 \pmod p$, one root is itself a quadratic residue and the other is not. (The root that is the quadratic residue is known as the principal square root.) See Exercises 2.26 and 2.42 for applications of the principal root.

2.26. We denote by $\mathbf{Z}_n^*$ the multiplicative group of the elements in $\mathbf{Z}_n$ that are coprime to n.

(1) Suppose n is odd and has exactly k distinct prime factors. Let J denote the set of elements $x \in \mathbf{Z}_n^*$ with the Jacobi symbol $\left(\frac{x}{n}\right) = 1$ and let S denote the set of squares in $\mathbf{Z}_n^*$. Show that J is a subgroup of $\mathbf{Z}_n^*$ of $\varphi(n)/2$ elements, and that S is a subgroup of J.

(2) Show that squares in $\mathbf{Z}_n^*$ have exactly 2^k square roots in $\mathbf{Z}_n^*$ and conclude that $\#S = \varphi(n)/2^k$.

(3) Now suppose n is a Blum integer; that is, $n = pq$ is a product of two different primes $p, q \equiv 3 \pmod 4$. (Blum integers have importance in cryptography (see [Menezes et al. 1997] and our Section 8.2).) From parts (1) and (2), $\#S = \frac{1}{2}\#J$, so that half of J's elements are squares, and half are not. From part (2), an element of S has exactly 4 square roots. Show that exactly one of these square roots is itself in S.

(4) For a Blum integer $n = pq$, show that the squaring function $s(x) = x^2 \bmod n$ is a permutation on the set S, and that its inverse function is

$$s^{-1}(y) = y^{((p-1)(q-1)+4)/8} \bmod n.$$

2.27. Using Theorem 2.3.7 prove the two equalities in relations (2.12).

2.28. Here we prove the celebrated quadratic reciprocity relation (2.11) for two distinct odd primes p, q. Starting with Definition 2.3.6, show that G is multiplicative; that is, if $\gcd(m,n) = 1$, then

$$G(m;n)G(n;m) = G(1;mn).$$

(Hint: $mj^2/n + nk^2/m$ is similar—in a specific sense—to $(mj+nk)^2/(mn)$.) Infer from this and Theorem 2.3.7 the relation (now for primes p, q)

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}.$$

These are examples *par excellence* of the potential power of exponential sums; in fact, this approach is one of the more efficient ways to arrive at reciprocity. Extend the result to obtain the formula of Theorem 2.3.4 for $\left(\frac{2}{p}\right)$. Can this approach be extended to the more general reciprocity statement (i.e., for coprime m, n) in Theorem 2.3.4? Incidentally, Gauss sums for nonprime arguments m, n can be evaluated in closed form, using the techniques of Exercise 1.66 or the methods summarized in references such as [Graham and Kolesnik 1991].

2.29. This exercise is designed for honing one's skills in manipulating Gauss sums. The task is to count, among quadratic residues modulo a prime p, the exact number of arithmetic progressions of given length. The formal count of length-3 progressions is taken to be

$$A(p) = \#\left\{(r,s,t)\ :\ \left(\tfrac{r}{p}\right) = \left(\tfrac{s}{p}\right) = \left(\tfrac{t}{p}\right) = 1;\ r \neq s;\ s - r \equiv t - s \pmod{p}\right\}.$$

Note we are taking $0 \le r, s, t \le p-1$, we are ignoring trivial progressions (r,r,r), and that 0 is not a quadratic residue. So the prime $p = 11$, for which the quadratic residues are $\{1,3,4,5,9\}$, enjoys a total of $A(11) = 10$ arithmetic progressions of length three. (One of these is $4, 9, 3$; i.e., we allow wraparound (mod 11); and also, descenders such as $5, 4, 3$ are allowed.)

First, prove that

$$A(p) = -\frac{p-1}{2} + \frac{1}{p}\sum_{k=0}^{p-1}\sum_{r,s,t} e^{2\pi i k(r-2s+t)/p},$$

where each of r, s, t runs through the quadratic residues. Then, use relations (2.12) to prove that

$$A(p) = \frac{p-1}{8}\left(p - 6 - 2\left(\frac{2}{p}\right) - \left(\frac{-1}{p}\right)\right).$$

Finally, derive for the exact progression count the attractive expression

$$A(p) = (p-1)\left\lfloor \frac{p-2}{8} \right\rfloor.$$

An interesting extension to this exercise is to analyze progressions of longer length. Another direction: How many progressions of a given length would be expected to exist amongst a *random* half of all residues $\{1,\ 2,\ 3,\ \ldots,\ p-1\}$ (see Exercise 2.41)?

2.30. Prove that square-root Algorithms 2.3.8 and 2.3.9 work.

2.31. Prove that the following algorithm (certainly reminiscent of the text Algorithm 2.3.9) works for square roots (mod p), for p an odd prime. Let x be the quadratic residue for which we desire a square root. Define a particular Lucas sequence (V_k) by $V_0 = 2, V_1 = h$, and for $k > 1$

$$V_k = hV_{k-1} - xV_{k-2},$$

where h is such that $\left(\frac{h^2-4x}{p}\right) = -1$. Then compute a square root of x as

$$y = \frac{1}{2}V_{(p+1)/2} \pmod{p}.$$

Note that the Lucas numbers can be computed via a binary Lucas chain; see Algorithm 3.6.7.

2.32. Implement Algorithm 2.3.8 or 2.3.9 or some other variant to solve each of

$$x^2 \equiv 3615 \pmod{2^{16}+1},$$

$$x^2 \equiv 552512556430486016984082237 \pmod{2^{89}-1}.$$

2.33. Show how to enhance Algorithm 2.3.8 by avoiding some of the powerings called for, perhaps by a precomputation.

2.34. Prove that a primitive root of an odd prime p is a quadratic nonresidue.

2.35. Prove that Algorithm 2.3.12 (alternatively 2.3.13) works. As intimated in the text, the proof is not entirely easy. It may help to first prove a special-case algorithm, namely for finding representations $p = a^2 + b^2$ when $p \equiv 1 \pmod 4$. Such a representation always exists and is unique.

2.36. Since we have algorithms that extract square roots modulo primes, give an algorithm for extracting square roots (mod n), where $n = pq$ is the product of two explicitly given primes. (The Chinese remainder theorem (CRT) will be useful here.) How can one extract square roots of a prime power $n = p^k$? How can one extract square roots modulo n if the complete prime factorization of n is known?

Note that in ignorance of the factorization of n, square root extraction is extremely hard—essentially equivalent to factoring itself; see Exercise 6.5.

2.37. Prove that for odd prime p, the number of roots of $ax^2 + bx + c \equiv 0 \pmod p$, where $a \not\equiv 0 \pmod p$, is given by $1 + \left(\frac{D}{p}\right)$, where $D = b^2 - 4ac$ is the discriminant. For the case $1 + \left(\frac{D}{p}\right) > 0$, give an algorithm for calculation of all the roots.

2.38. Find a prime p such that the least primitive root of p exceeds the number of binary bits in p. Find an example of such a prime p that is also a Mersenne prime (i.e., some $p = M_q = 2^q - 1$ whose least primitive root exceeds q). These findings show that the least primitive root can exceed $\lg p$. For more exploration along these lines see Exercise 2.39.

2.5 Research problems

2.39. Implement a primitive root-finding algorithm, and study the statistical occurrence of least primitive roots.

The study of least primitive roots is highly interesting. It is known on the GRH that 2 is a primitive root of infinitely many primes, in fact for a positive proportion $\alpha = \prod(1 - 1/p(p-1)) \approx 0.3739558$, the product running over all primes (see Exercise 1.90). Again on the GRH, a positive proportion whose least primitive root is not 2, has 3 as a primitive root and so on; see [Hooley 1976]. It is conjectured that the least primitive root for prime p is $O((\ln p)(\ln\ln p))$; see [Bach 1997a]. It is known, on the GRH, that the least primitive root for prime p is $O\left(\ln^6 p\right)$; see [Shoup 1992]. It is known unconditionally that the least primitive root for prime p is $O(p^{1/4+\epsilon})$ for every $\epsilon > 0$, and for infinitely many primes p it exceeds $c \ln p \ln\ln\ln p$ for some positive constant c, the latter a result of S. Graham and C. Ringrosee. The study of the least primitive root is not unlike the study of the least quadratic nonresidue—in this regard see Exercise 2.41.

2.40. Investigate the use of CRT in the seemingly remote domains of integer convolution, or fast Fourier transforms, or public-key cryptography. A good reference is [Ding et al. 1996].

2.41. Here we explore what might be called "statistical" features of the Legendre symbol. For odd prime p, denote by $N(a,b)$ the number of residues whose *successive* quadratic characters are (a,b); that is, we wish to count those integers $x \in [1, p-2]$ such that

$$\left(\left(\frac{x}{p}\right), \left(\frac{x+1}{p}\right)\right) = (a,b),$$

with each of a, b attaining possible values ± 1. Prove that

$$4N(a,b) = \sum_{x=1}^{p-2} \left(1 + a\left(\frac{x}{p}\right)\right)\left(1 + b\left(\frac{x+1}{p}\right)\right)$$

and therefore that

$$N(a,b) = \frac{1}{4}\left(p - 2 - b - ab - a\left(\frac{-1}{p}\right)\right).$$

Establish the corollary that the number of pairs of consecutive quadratic residues is $(p-5)/4, (p-3)/4$, respectively, as $p \equiv 1, 3 \pmod 4$. Using the formula for $N(a,b)$, prove that for every prime p the congruence

$$x^2 + y^2 \equiv -1 \pmod p$$

is solvable.

One satisfying aspect of the $N(a,b)$ formula is the statistical notion that sure enough, if the Legendre symbol is thought of as generated by a "random coin flip," there ought to be about $p/4$ occurrences of a given pair $(\pm 1, \pm 1)$.

All of this makes sense: The Legendre symbol is in some sense random. But in another sense, it is not quite so random. Let us estimate a sum:

$$s_{A,B} = \sum_{A \le x < B} \left(\frac{x}{p}\right),$$

which can be thought of, in some heuristic sense we suppose, as a random walk with $N = B - A$ steps. On the basis of remarks following Theorem 2.3.7, show that

$$|s_{A,B}| \le \frac{1}{\sqrt{p}} \sum_{b=0}^{p-1} \left| \frac{\sin(\pi N b/p)}{\sin(\pi b/p)} \right| \le \frac{1}{\sqrt{p}} \sum_{b=0}^{p-1} \frac{1}{|\sin(\pi b/p)|}.$$

Finally, arrive at the Pólya–Vinogradov inequality:

$$|s_{A,B}| < \sqrt{p} \ln p.$$

Actually, the inequality is often expressed more generally, where instead of the Legendre symbol as character, any nonprincipal character applies. This attractive inequality says that indeed, the "statistical fluctuation" of the quadratic residue/nonresidue count, starting from any initial $x = A$, is always bounded by a "variance factor" $\sqrt{p}$ (times a log term). One can prove more than this; for example, using an inequality in [Cochrane 1987] one can obtain

$$|s_{A,B}| < \frac{4}{\pi^2} \sqrt{p} \ln p + 0.41\sqrt{p} + 0.61,$$

and it is known that on the GRH, $s_{A,B} = O\left(\sqrt{p} \ln \ln p\right)$; see [Davenport 1980]. In any case, we deduce that out of any N consecutive integers, $N/2 + O(p^{1/2} \ln p)$ are quadratic residues (mod p). We also conclude that the least quadratic nonresidue (mod p) is bounded above by, at worst, $\sqrt{p} \ln p$. Further results on this interesting inequality are discussed in [Hildebrand 1988a, 1988b].

The Pólya–Vinogradov inequality thus restricted to quadratic characters tells us that *not* just any coin-flip sequence can be a Legendre-symbol sequence. The inequality says that we cannot, for large p say, have a Legendre-symbol sequence such as $(1, 1, 1, \ldots, -1 - 1 - 1)$ (i.e., first half are 1's second half -1's). We cannot even build up more than an $O\left(\sqrt{p}\ln p\right)$ excess of one symbol over the other. But in a truly random coin-flip game, any pattern of 1's and -1's is allowed; and even if you constrain such a game to have equal numbers of 1's and -1's as does the Legendre-symbol game, there are still vast numbers of possible coin-flip sequences that cannot be symbol sequences. In some sense, however, the Pólya–Vinogradov inequality puts the Legendre symbol sequence smack in the middle of the distribution of possible sequences: It is what we might expect for a *random* sequence of coin flips. Incidentally, in view of the coin-flip analogy, what would be the expected value of the least quadratic nonresidue (mod p)? In this regard see Exercise 2.39. For a different kind of constraint on presumably random quadratic residues, see the remarks at the end of Exercise 2.29.

2.42. Here is a fascinating line of research: Using the age-old and glorious theory of the arithmetic–geometric mean (AGM), investigate the notion of what we might call a "discrete arithmetic–geometric mean (DAGM)." It was a *tour de force* of analysis, due to Gauss, Legendre, Jacobi, to conceive of the analytic AGM, which is the asymptotic fixed point of the elegant iteration

$$(a, b) \mapsto \left(\frac{a+b}{2}, \sqrt{ab}\right),$$

that is, one replaces the pair (a, b) of real numbers with the new pair of arithmetic and geometric means, respectively. The classical AGM, then, is the real number c to which the two numbers converge; sure enough, $(c, c) \mapsto (c, c)$ so the process tends to stabilize for appropriate initial choices of a and b. This scheme is connected with the theory of elliptic integrals, the calculation of π to (literally) billions of decimal places, and so on [Borwein and Borwein 1987].

But consider doing this procedure not on real numbers but on residues modulo a prime $p \equiv 3 \pmod 4$, in which case an $x \pmod p$ that has a square root always has a so-called principal root (and so an unambiguous choice of square root can be taken; see Exercise 2.25). Work out a theory of the DAGM modulo p. Perhaps you would want to cast $\sqrt{ab}$ as a principal root if said root exists, but something like a different principal root, say $\sqrt{gab}$, for some fixed nonresidue g when ab is a nonresidue. Interesting theoretical issues are these: Does the DAGM have an interesting cycle structure? Is there any relation between your DAGM and the classical, analytic AGM? If there *were* any fortuitous connection between the discrete and analytic means, one might have a new way to evaluate with high efficiency certain finite hypergeometric series, as appear in Exercise 7.26.

Springer

http://www.springer.com/978-0-387-25282-7

Prime Numbers
A Computational Perspective
Crandall, R.; Pomerance, C.
2005, XV, 597 p., Hardcover
ISBN: 978-0-387-25282-7