Chapter 2

Basic Algorithmic Number Theory

The aim of this chapter is to give a brief summary of some fundamental algorithms for arithmetic in finite fields. The intention is not to provide an implementation guide; instead, we sketch some important concepts and state some complexity results that will be used later in the book. We do not give a consistent level of detail for all algorithms; instead we only give full details for algorithms that will play a significant role in later chapters of the book.

More details of these subjects can be found in Crandall and Pomerance [162], Shoup [556], Buhler and Stevenhagen [113], Brent and Zimmermann [100], Knuth [343], von zur Gathen and Gerhard [238], Bach and Shallit [22] and the handbooks [16, 418].

The chapter begins with some remarks about computational problems, algorithms and complexity theory. We then present methods for fast integer and modular arithmetic. Next we present some fundamental algorithms in computational number theory such as Euclid's algorithm, computing Legendre symbols, and taking square roots modulo p. Finally, we discuss polynomial arithmetic, constructing finite fields, and some computational problems in finite fields.

2.1 Algorithms and Complexity

We assume the reader is already familiar with computers, computation and algorithms. General references for this section are Chapter 1 of Cormen, Leiserson, Rivest and Stein [146], Davis and Weyuker [167], Hopcroft and Ullman [293], Section 3.1 of Shoup [556], Sipser [568] and Talbot and Welsh [600].

Rather than using a fully abstract model of computation, such as Turing machines, we consider all algorithms as running on a digital computer with a typical instruction set,

an infinite number of bits of memory and constant-time memory access. This is similar to the random access machine (or register machine) model; see Section 3.6 of [22], [139], Section 2.2 of [146], Section 7.6 of [293] or Section 3.2 of [556]. We think of an algorithm as a sequence of bit operations, though it is more realistic to consider word operations.

A **computational problem** is specified by an input (of a certain form) and an output (satisfying certain properties relative to the input). An **instance** of a computational problem is a specific input. The **input size** of an instance of a computational problem is the number of bits required to represent the instance. The **output size** of an instance of a computational problem is the number of bits necessary to represent the output. A **decision problem** is a computational problem where the output is either "yes" or "no". As an example, we give one of the most important definitions in the book.

Definition 2.1.1. Let G be a group written in multiplicative notation. The **discrete logarithm problem** (**DLP**) is: Given $g, h \in G$ to find a, if it exists, such that $h = g^a$.

In Definition 2.1.1 the input is a description of the group G together with the group elements g and h and the output is a or the failure symbol $\perp$ (to indicate that $h \notin \langle g \rangle$). Typically G is an algebraic group over a finite field and the order of g is assumed to be known. We stress that an instance of the DLP, according to Definition 2.1.1, includes the specification of G, g and h; so one must understand that they are all allowed to vary (note that, in many cryptographic applications one considers the group G and element g as being fixed; we discuss this in Exercise 21.1.2). As explained in Section 2.1.2, a computational problem should be defined with respect to an instance generator; in the absence of any further information it is usual to assume that the instances are chosen uniformly from the space of all possible inputs of a given size. In particular, for the DLP it is usual to denote the order of g by r and to assume that $h = g^a$ where a is chosen uniformly in $\mathbb{Z}/r\mathbb{Z}$. The output is the integer a (e.g., written in binary). The input size depends on the specific group G and the method used to represent it. If h can take all values in $\langle g \rangle$ then one needs at least $\log_2(r)$ bits to specify h from among the r possibilities. Hence, the input size is at least $\log_2(r)$ bits. Similarly, if the output a is uniformly distributed in $\mathbb{Z}/r\mathbb{Z}$ then the output size is at least $\log_2(r)$ bits.

An algorithm to solve a computational problem is called **deterministic** if it does not make use of any randomness. We will study the asymptotic complexity of deterministic algorithms by counting the number of bit operations performed by the algorithm expressed as a function of the input size. Upper bounds on the complexity are presented using "big O" notation. When giving complexity estimates using big O notation we implicitly assume that there is a countably infinite number of possible inputs to the algorithm.

Definition 2.1.2. Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$. Write $f = O(g)$ if there are $c \in \mathbb{R}_{>0}$ and $N \in \mathbb{N}$ such that

$$f(n) \leq cg(n)$$

for all $n \geq N$.

Similarly, if $f(n_1, \ldots, n_m)$ and $g(n_1, \ldots, n_m)$ are functions from $\mathbb{N}^m$ to $\mathbb{R}_{>0}$ then we write $f = O(g)$ if there are $c \in \mathbb{R}_{>0}$ and $N_1, \ldots, N_m \in \mathbb{N}$ such that $f(n_1, \ldots, n_m) \leq cg(n_1, \ldots, n_m)$ for all $(n_1, \ldots, n_m) \in \mathbb{N}^m$ with $n_i \geq N_i$ for all $1 \leq i \leq m$.

Example 2.1.3. $3n^2 + 2n + 1 = O(n^2)$, $n + \sin(n) = O(n)$, $n^{100} + 2^n = O(2^n)$, $\log_{10}(n) = O(\log(n))$.

Exercise 2.1.4. Show that if $f(n) = O(\log(n)^a)$ and $g(n) = O(\log(n)^b)$ then $(f+g)(n) = f(n) + g(n) = O(\log(n)^{\max\{a,b\}})$ and $(fg)(n) = f(n)g(n) = O(\log(n)^{a+b})$. Show that $O(n^c) = O(2^{c\log(n)})$.

We also present the "little o", "soft O", "big Omega" and "big Theta" notation . These will only ever be used in this book for functions of a single argument.

Definition 2.1.5. Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$. Write $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} f(n)/g(n) = 0.$$

Write $f(n) = \tilde{O}(g(n))$ if there is some $m \in \mathbb{N}$ such that $f(n) = O(g(n)\log(g(n))^m)$. Write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$. Write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Exercise 2.1.6. Show that if $g(n) = O(n)$ and $f(n) = \tilde{O}(g(n))$ then there is some $m \in \mathbb{N}$ such that $f(n) = O(n\log(n)^m)$.

Definition 2.1.7. (Worst-case asymptotic complexity.) Let A be a deterministic algorithm and let $t(n)$ be a bound on the running time of A on every problem of input size n bits.

- A is **polynomial-time** if there is an integer $k \in \mathbb{N}$ such that $t(n) = O(n^k)$.
- A is **superpolynomial-time** if $t(n) = \Omega(n^c)$ for all $c \in \mathbb{R}_{>1}$.
- A is **exponential-time** if there is a constant $c_2 > 1$ such that $t(n) = O(c_2^n)$.
- A is **subexponential-time** if $t(n) = O(c^n)$ for all $c \in \mathbb{R}_{>1}$.

Exercise 2.1.8. Show that $n^{a\log(\log(n))}$ and $n^{a\log(n)}$, for some $a \in \mathbb{R}_{>0}$, are functions that are $\Omega(n^c)$ and $O(c^n)$ for all $c \in \mathbb{R}_{>1}$.

For more information about computational complexity, including the definitions of complexity classes such as P and NP, see Chapters 2 to 4 of [600], Chapter 13 of [293], Chapter 15 of [167], Chapter 7 of [568] or Chapter 34 of [146]. Definition 2.1.7 is for **uniform** complexity, as a single algorithm A solves all problem instances. One can also consider **non-uniform complexity**, where one has an algorithm A and, for each $n \in \mathbb{N}$, polynomially sized auxiliary input $h(n)$ (the hint) such that if x is an n-bit instance of the computational problem then $A(x, h(n))$ solves the instance. An alternative definition is a sequence A_n of algorithms, one for each input size $n \in \mathbb{N}$, and such that the description of the algorithm is polynomially bounded. We stress that the hint is not required to be efficiently computable. We refer to Section 4.6 of Talbot and Welsh [600] for details.

Complexity theory is an excellent tool for comparing algorithms, but one should always be aware that the results can be misleading. For example, it can happen that there are several algorithms to solve a computational problem and that the one with the best complexity is slower than the others for the specific problem instance one is interested in (for example, see Remark 2.2.5).

2.1.1 Randomised Algorithms

All our algorithms may be **randomised**, in the sense that they have access to a random number generator. A deterministic algorithm should terminate after a finite number of steps but a randomised algorithm can run forever if an infinite sequence of "unlucky" random choices is made.[1] Also, a randomised algorithm may output an incorrect answer for

[1] In algorithmic number theory it is traditional to allow algorithms that do not necessarily terminate, whereas in cryptography it is traditional to consider algorithms whose running time is bounded (typically by a polynomial in the input size). Indeed, in security reductions it is crucial that an adversary (i.e., randomised algorithm) always terminates. Hence, some of the definitions in this section (e.g., Las Vegas algorithms) mainly arise in the algorithmic number theory literature.

some choices of randomness. A **Las Vegas algorithm** is a randomised algorithm which, if it terminates[2], outputs a correct solution to the problem. A randomised algorithm for a decision problem is a **Monte Carlo algorithm** if it always terminates and if the output is "yes" then it is correct and if the output is "no" then it is correct with "noticeable" probability (see the next section for a formal definition of noticeable success probability).

An example of a Las Vegas algorithm is choosing a random quadratic non-residue modulo p by choosing random integers modulo p and computing the Legendre symbol (see Exercise 2.4.6 in Section 2.4); the algorithm could be extremely unlucky forever. Of course, there is a deterministic algorithm for this problem, but its complexity is worse than the randomised algorithm. An example of a Monte Carlo algorithm is testing primality of an integer N using the Miller-Rabin test (see Section 12.1.2). Many of the algorithms in the book are randomised Monte Carlo or Las Vegas algorithms. We will often omit the words "Las Vegas" or "Monte Carlo".

Deterministic algorithms have a well-defined running time on any given problem instance. For a randomised algorithm the running time for a fixed instance of the problem is not necessarily well-defined. Instead, one considers the expected value of the running time over all choices of the randomness. We usually consider **worst-case complexity**. For a randomised algorithm, the worst-case complexity for input size n is the maximum, over all problem instances of size n, of the expected running time of the algorithm. As always, when considering asymptotic complexity it is necessary that the computational problem have a countably infinite number of problem instances.

A randomised algorithm is **expected polynomial-time** if the worst-case over all problem instances of size n bits of the expected value of the running time is $O(n^c)$ for some $c \in \mathbb{R}_{>0}$. (An expected polynomial-time algorithm can run longer than polynomial-time if it makes many "unlucky" choices.) A randomised algorithm is **expected exponential-time** (respectively, **expected subexponential-time**) if there exists $c \in \mathbb{R}_{>1}$ (respectively, for all $c \in \mathbb{R}_{>1}$) such that the expected value of the running time on problem instances of size n bits is $O(c^n)$.

One can also consider **average-case complexity**, which is the average, over all problem instances of size n, of the expected running time of the algorithm. Equivalently, the average-case complexity is the expected number of bit operations of the algorithm where the expectation is taken over all problem instances of size n as well as all choices of the randomness. For more details see Section 4.2 of Talbot and Welsh [600].

2.1.2 Success Probability of a Randomised Algorithm

Throughout the book we give very simple definitions (like Definition 2.1.1) for computational problems. However, it is more subtle to define what it means for a randomised algorithm A to solve a computational problem. A **perfect algorithm** is one whose output is always correct (i.e., it always succeeds). We also consider algorithms that give the correct answer only for some subset of the problem instances, or for all instances but only with a certain probability.

The issue of whether an algorithm is successful is handled somewhat differently by the two communities whose work is surveyed in this book. In the computational number theory community, algorithms are expected to solve all problem instances with probability of success close to 1. In the cryptography community it is usual to consider algorithms that only solve some noticeable (see Definition 2.1.10) proportion of problem instances, and even then only with some noticeable probability. The motivation for the latter community

[2] An alternative definition is that a Las Vegas algorithm has finite expected running time, and outputs either a correct result or the failure symbol $\perp$.

is that an algorithm to break a cryptosystem is considered devastating even if only a relatively small proportion of ciphertexts are vulnerable to the attack. Two examples of attacks that only apply to a small proportion of ciphertexts are the attack by Boneh, Joux and Nguyen on textbook Elgamal (see Exercise 20.4.9) and the Desmedt-Odlyzko signature forgery method (see Section 24.4.3).

We give general definitions for the success probability of an algorithm in this section, but rarely use the formalism in our later discussion. Instead, for most of the book, we focus on the case of algorithms that always succeed (or, at least, that succeed with probability extremely close to 1). This choice allows shorter and simpler proofs of many facts. In any case, for most computational problems the success probability can be increased by running the algorithm repeatedly, see Section 2.1.4.

The success of an algorithm to solve a computational problem is defined with respect to an **instance generator**, which is a randomised algorithm that takes as input $\kappa \in \mathbb{N}$ (often κ is called the **security parameter**), runs in polynomial-time in the output size, and outputs an instance of the computational problem (or fails to generate an instance with some negligible probability). The output is usually assumed to be $\Theta(\kappa)$ bits,[3] so "polynomial-time" in the previous sentence means $O(\kappa^m)$ bit operations for some $m \in \mathbb{N}$. We give an example of an instance generator for the DLP in Example 2.1.9.

Let A be a randomised algorithm that takes an input $\kappa \in \mathbb{N}$. Write $\mathcal{S}_\kappa$ for the set of possible outputs of $A(\kappa)$. The **output distribution** of A on input κ is the distribution on $\mathcal{S}_\kappa$ such that $\Pr(x)$ for $x \in \mathcal{S}_\kappa$ is the probability, over the random choices made by A, that the output of $A(\kappa)$ is x.

Example 2.1.9. Let a security parameter $\kappa \in \mathbb{N}$ be given. First, generate a random prime number r such that $2^{2\kappa} < r < 2^{2\kappa+1}$ (by choosing uniformly at random $(2\kappa+1)$-bit integers and testing each for primality). Next, try consecutive small[4] integers k until $p = kr + 1$ is prime. Then, choose a random integer $1 < u < p$ and set $g = u^{(p-1)/r}$ and repeat if $g = 1$. It follows that g is a generator of a cyclic subgroup of $G = \mathbb{F}_p^*$ of order r. Finally, choose uniformly at random an integer $0 < a < r$ and set $h = g^a$. Output (p, r, g, h), which can be achieved using $3\lceil \log_2(p) \rceil + \lceil \log_2(r) \rceil$ bits.

One sees that there are finitely many problem instances for a given value of the security parameter κ, but infinitely many instances in total. The output distribution has r uniform among $(2\kappa+1)$-bit primes, p is not at all random (it is essentially determined by r), while the pair (g, h) is uniformly distributed in the set of pairs of elements of order r in $\mathbb{F}_p^*$, but not necessarily well-distributed in $(\mathbb{F}_p^*)^2$.

When considering an algorithm A to solve a computational problem we assume that A has been customised for a particular instance generator. Hence, a problem might be easy with respect to some instance generators and hard for others. Thus it makes no sense to claim that "DLP is a hard problem"; instead, one should conjecture that DLP is hard for certain instance generators.

We now define what is meant by the word negligible.

Definition 2.1.10. A function $\epsilon : \mathbb{N} \to \mathbb{R}_{>0}$ is **negligible** if for every polynomial $p(x) \in \mathbb{R}[x]$ there is some $K \in \mathbb{N}$ such that for all $\kappa > K$ with $p(\kappa) \neq 0$ we have $\epsilon(\kappa) < 1/|p(\kappa)|$.

A function $\epsilon : \mathbb{N} \to \mathbb{R}_{>0}$ is **noticeable** if there exists a polynomial $p(x) \in \mathbb{R}[x]$ and an integer K such that $\epsilon(\kappa) > 1/|p(\kappa)|$ for all $\kappa > K$ with $p(\kappa) \neq 0$.

[3] For problems related to RSA or factoring, we may either take κ to be the bit-length of the modulus, or assume the output is $O(\kappa^3)$ bits.

[4] In practice, to ensure the discrete logarithm problem can't be solved in 2^κ bit operations using index calculus algorithms, one would choose k large enough that $L_p(1/3, 1.5) > 2^{kappa}$.

Let $[0,1] = \{x \in \mathbb{R} : 0 \leq x \leq 1\}$. A function $p : \mathbb{N} \rightarrow [0,1]$ is **overwhelming** if $1 - p(\kappa)$ is negligible.

Note that noticeable is not the logical negation of negligible. There are functions that are neither negligible nor noticeable.

Example 2.1.11. The function $\epsilon(\kappa) = 1/2^{\kappa}$ is negligible.

Exercise 2.1.12. Let $f_1(\kappa)$ and $f_2(\kappa)$ be negligible functions. Prove that $f_1 + f_2$ is a negligible function and that $p(\kappa)f_1(\kappa)$ is a negligible function for any polynomial $p(x) \in \mathbb{R}[x]$ such that $p(x) > 0$ for all sufficiently large x.

Definition 2.1.13. Let A be a randomised algorithm to solve instances of a computational problem generated by a specific instance generator. The **success probability** of the algorithm A is the function $f : \mathbb{N} \rightarrow [0,1]$ such that, for $\kappa \in \mathbb{N}$, $f(\kappa)$ is the probability that A outputs the correct answer, where the probability is taken over the randomness used by A and according to the output distribution of the instance generator on input κ. An algorithm A with respect to an instance generator **succeeds** if its success probability is a noticeable function.

Note that the success probability is taken over both the random choices made by A and the distribution of problem instances. In particular, an algorithm that succeeds does not necessarily solve a specific problem instance even when run repeatedly with different random choices.

Example 2.1.14. Consider an algorithm A for the DLP with respect to the instance generator of Example 2.1.9. Suppose A simply outputs an integer a chosen uniformly at random in the range $0 < a < r$. Since $r > 2^{2\kappa}$ the probability that A is correct is $1/(r-1) \leq 1/2^{2\kappa}$. For any polynomial $p(x)$ there are $X, c \in \mathbb{R}_{>0}$ and $n \in \mathbb{N}$ such that $|p(x)| \leq cx^n$ for $x \geq X$. Similarly, there is some $K \geq X$ such that $cK^n \leq 2^{2K}$. Hence, the success probability of A is negligible.

Certain decision problems (for example, decision Diffie-Hellman) require an algorithm to behave differently when given inputs drawn from different distributions on the same underlying set. In this case, the success probability is not the right concept and one instead uses the **advantage**. We refer to Definition 20.2.4 for an example.

Chapter 7 of Shoup [556] gives further discussion of randomised algorithms and success probabilities.

2.1.3 Reductions

An **oracle** for a computational problem takes one unit of running time, independent of the size of the instance, and returns an output. An oracle that always outputs a correct answer is called a **perfect oracle**. One can consider oracles that only output a correct answer with a certain noticeable probability (or advantage). For simplicity we usually assume that oracles are perfect and leave the details in the general case as an exercise for the reader. We sometimes use the word **reliable** for an oracle whose success probability is overwhelming (i.e., success probability $1 - \epsilon$ where ϵ is negligible) and **unreliable** for an oracle whose success probability is small (but still noticeable).

Note that the behaviour of an oracle is only defined if its input is a valid instance of the computational problem it solves. Similarly, the oracle performs with the stated success probability only if it is given problem instances drawn with the correct distribution from the set of all problem instances.

Definition 2.1.15. A **reduction** from problem A to problem B is a randomised algorithm to solve problem A (running in expected polynomial-time and having noticeable success probability) by making queries to an oracle (which succeeds with noticeable probability) to solve problem B.

If there is a reduction from problem A to problem B then we write[5]

$$\text{A} \leq_R \text{B}.$$

Theorem 2.1.16. *Let A and B be computational problems such that $A \leq_R B$. If there is a polynomial-time randomised algorithm to solve B then there is a polynomial-time randomised algorithm to solve A.*

A reduction between problems A and B therefore explains that "if you can solve B then you can solve A". This means that solving A has been "reduced" to solving problem B and we can infer that problem B is "at least as hard as" problem A or that problem A is "no harder than" problem B.

Since oracle queries take one unit of running time and reductions are polynomial-time algorithms, a reduction makes only polynomially many oracle queries.

Definition 2.1.17. If there is a reduction from A to B and a reduction from B to A then we say that problems A and B are **equivalent** and write $A \equiv_R B$.

Some authors use the phrases **polynomial-time reduction** and **polynomial-time equivalent** in place of reduction and equivalence. However, these terms have a technical meaning in complexity theory that is different from reduction (see Section 34.3 of [146]). Definition 2.1.15 is closer to the notion of Turing reduction, except that we allow randomised algorithms whereas a Turing reduction is a deterministic algorithm. We abuse terminology and define the terms **subexponential-time reduction** and **exponential-time reduction** by relaxing the condition in Definition 2.1.15 that the algorithm be polynomial-time (these terms are used in Section 21.4.3).

2.1.4 Random Self-Reducibility

There are two different ways that an algorithm or oracle can be unreliable: First, it may be randomised and only output the correct answer with some probability; such a situation is relatively easy to deal with by repeatedly running the algorithm/oracle on the same input. The second situation, which is more difficult to handle, is when there is a subset of problem instances for which the algorithm or oracle extremely rarely or never outputs the correct solution; for this situation random self-reducibility is essential. We give a definition only for the special case of computational problems in groups.

Definition 2.1.18. Let P be a computational problem for which every instance of the problem is an n_1-tuple of elements of some cyclic group G of order r and such that the solution is an n_2-tuple of elements of G together with an n_3-tuple of elements of $\mathbb{Z}/r\mathbb{Z}$ (where n_2 or n_3 may be zero).

The computational problem P is **random self-reducible** if there is a polynomial-time algorithm that transforms an instance of the problem (with elements in a group G) into a uniformly random instance of the problem (with elements in the *same* group G) such that the solution to the original problem can be obtained in polynomial-time from the solution to the new instance.

[5]The subscript R denotes the word "reduction" and should also remind the reader that our reductions are randomised algorithms.

We stress that a random self-reduction of a computational problem in a group G gives instances of the same computational problem in the same group. In general there is no way to use information about instances of a computational problem in a group G' to solve computational problems in G if $G' \neq G$ (unless perhaps G' is a subgroup of G or vice versa).

Lemma 2.1.19. *Let G be a group and let $g \in G$ have prime order r. Then the DLP in $\langle g \rangle$ is random self-reducible.*

Proof: First, note that the DLP fits the framework of computational problems in Definition 2.1.18. Denote by $\mathcal{X}$ the set $(\langle g \rangle - \{1\}) \times \langle g \rangle$. Let $(g, h) \in \mathcal{X}$ be an instance of the DLP.

Choose $1 \leq x < r$ and $0 \leq y < r$ uniformly at random and consider the pair $(g^x, h^x g^{xy}) \in \mathcal{X}$. Every pair $(g_1, g_2) \in \mathcal{X}$ arises in this way for exactly one pair (x, y). Hence we have produced a DLP instance uniformly at random.

If a is the solution to the new DLP instance, i.e., $h^x g^{xy} = (g^x)^a$ then the solution to the original instance is

$$a - y \pmod{r}.$$

This completes the proof. □

A useful feature of random self-reducible problems is that if A is an algorithm that solves an instance of the problem in a group G with probability (or advantage) ϵ then one can obtain an algorithm A' that repeatedly calls A and solves any instance in G of the problem with overwhelming probability. This is called **amplifying** the success probability (or advantage). An algorithm to transform an unreliable oracle into a reliable one is sometimes called a **self-corrector**.

Lemma 2.1.20. *Let g have prime order r and let $G = \langle g \rangle$. Let A be an algorithm that solves the DLP in G with probability at least $\epsilon > 0$. Let $\epsilon' > 0$ and define $n = \lceil \log(1/\epsilon')/\epsilon \rceil$ (where* log *denotes the natural logarithm). Then there is an algorithm A' that solves the DLP in G with probability at least $1 - \epsilon'$. The running time of A' is $O(n \log(r))$ group operations plus n times the running time of A.*

Proof: Run A on n random self-reduced versions of the original DLP. One convenient feature of the DLP is that one can check whether a solution is correct (this takes $O(\log(r))$ group operations for each guess for the DLP).

The probability that all n trials are incorrect is at most $(1-\epsilon)^n < (e^{-\epsilon})^{\log(1/\epsilon')/\epsilon} = e^{\log(\epsilon')} = \epsilon'$. Hence A' outputs the correct answer with probability at least $1 - \epsilon'$. □

2.2 Integer Operations

We now begin our survey of efficient computer arithmetic. General references for this topic are Section 9.1 of Crandall and Pomerance [162], Section 3.3 of Shoup [556], Section 4.3.1 of Knuth [343], Chapter 1 of Brent-Zimmermann [100] and von zur Gathen and Gerhard [238].

Integers are represented as a sequence of binary words. Operations like add or multiply may correspond to many bit or word operations. The **length** of an unsigned integer a represented in binary is

$$\text{len}(a) = \begin{cases} \lfloor \log_2(a) \rfloor + 1 & \text{if } a \neq 0, \\ 1 & \text{if } a = 0. \end{cases}$$

For a signed integer we define $\text{len}(a) = \text{len}(|a|) + 1$.

The complexity of algorithms manipulating integers depends on the length of the integers, hence one should express the complexity in terms of the function len. However, it is traditional to just use $\log_2$ or the natural logarithm log.

Exercise 2.2.1. Show that, for $a \in \mathbb{N}$, $\text{len}(a) = O(\log(a))$ and $\log(a) = O(\text{len}(a))$.

Lemma 2.2.2. *Let $a, b \in \mathbb{Z}$ be represented as a sequence of binary words.*

1. *It requires $O(\log(a))$ bit operations to write a out in binary.*
2. *One can compute $a \pm b$ in $O(\max\{\log(a), \log(b)\})$ bit operations.*
3. *One can compute ab in $O(\log(a)\log(b))$ bit operations.*
4. *Suppose $|a| > |b|$. One can compute q and r such that $a = bq + r$ and $0 \leq r < |b|$ in $O(\log(b)\log(q)) = O(\log(b)(\log(a) - \log(b) + 1))$ bit operations.*

Proof: Only the final statement is non-trivial. The school method of long division computes q and r simultaneously and requires $O(\log(q)\log(a))$ bit operations. It is more efficient to compute q first by considering only the most significant $\log_2(q)$ bits of a, and then to compute r as $a - bq$. For more details see Section 4.3.1 of [343], Section 2.4 of [238] or Section 3.3.4 of [556]. □

2.2.1 Faster Integer Multiplication

An important discovery is that it is possible to multiply integers more quickly than the "school method". General references for this subject include Section 9.5 of [162], Section 4.3.3 of [343], Section 3.5 of [556] and Section 1.3 of [100].

Karatsuba multiplication is based on the observation that one can compute $(a_0 + 2^n a_1)(b_0 + 2^n b_1)$, where a_0, a_1, b_0 and b_1 are n-bit integers, in three multiplications of n-bit integers rather than four.

Exercise 2.2.3. Prove that the complexity of Karatsuba multiplication of n bit integers is $O(n^{\log_2(3)}) = O(n^{1.585})$ bit operations.
[Hint: Assume n is a power of 2.]

Toom-Cook multiplication is a generalisation of Karatsuba. Fix a value k and suppose $a = a_0 + a_1 2^n + a_2 2^{2n} + \cdots a_k 2^{kn}$ and similarly for b. One can think of a and b as being polynomials in x of degree k evaluated at 2^n and we want to compute the product $c = ab$, which is a polynomial of degree $2k$ in x evaluated at $x = 2^n$. The idea is to compute the coefficients of the polynomial c using polynomial interpolation and therefore to recover c. The arithmetic is fast if the polynomials are evaluated at small integer values. Hence, we compute $c(1) = a(1)b(1)$, $c(-1) = a(-1)b(-1)$, $c(2) = a(2)b(2)$ etc. The complexity of Toom-Cook multiplication for n-bit integers is $O(n^{\log_{k+1}(2k+1)})$ (e.g., when $k = 3$ the complexity is $O(n^{1.465})$). For more details see Section 9.5.1 of [162].

Exercise 2.2.4.★ Give an algorithm for Toom-Cook multiplication with $k = 3$.

Schönhage-Strassen multiplication multiplies n-bit integers in nearly linear time, namely $O(n\log(n)\log(\log(n)))$ bit operations, using the fast Fourier transform (FFT). The Fürer algorithm is slightly better. These algorithms are not currently used in the implementation of RSA or discrete logarithm cryptosystems so we do not describe them in this book. We refer to Sections 9.5.2 to 9.5.7 of Crandall and Pomerance [162], Chapter 8 of von zur Gathen and Gerhard [238], Chapter 2 of Brent and Zimmermann [100], Turk [611] and Chapter 4 of Borodin and Munro [88] for details.

Another alternative is residue number arithmetic which is based on the Chinese remainder theorem. It reduces large integer operations to modular computations for some set of moduli. This idea may be useful if one can exploit parallel computation (though for any given application there may be more effective uses for parallelism). These methods are not used frequently for cryptography so interested readers are referred to Section II.1.2 of [64], Section 14.5.1 of [418], Remark 10.53(ii) of [16], and Section 4.3.2 of [343].

Remark 2.2.5. In practice, the "school" method is fastest for small numbers. The crossover point (i.e., when Karatsuba becomes faster than the school method) depends on the word size of the processor and many other issues, but seems to be for numbers of around 300-1000 bits (i.e., 90-300 digits) for most computing platforms. For a popular 32 bit processor Zimmermann [642] reports reports that Karatsuba beats the school method for integers of 20 words (640 bits) and Toom-Cook with $k = 3$ beats Karatsuba at 77 words (2464 bits). Bentahar [43] reports crossovers of 23 words (i.e., about 700 bits) and 133 words (approximately 4200 bits) respectively. The crossover point for the FFT methods is much larger. Hence, for elliptic curve cryptography at current security levels the "school" method is usually used, while for RSA cryptography the Karatsuba method is usually used.

Definition 2.2.6. Denote by $M(n)$ the number of bit operations to perform a multiplication of n bit integers.

For the remainder of the book we assume that $M(n) = c_1 n^2$ for some constant c_1 when talking about elliptic curve arithmetic, and that $M(n) = c_2 n^{1.585}$ for some constant c_2 when talking about RSA .

Applications of Newton's Method

Recall that if $F : \mathbb{R} \to \mathbb{R}$ is differentiable and if x_0 is an approximation to a zero of $F(x)$ then one can efficiently get a very close approximation to the zero by running Newton's iteration

$$x_{n+1} = x_n - F(x_n)/F'(x_n).$$

Newton's method has quadratic convergence, in general, so the precision of the approximation roughly doubles at each iteration.

Integer Division

There are a number of fast algorithms to compute $\lfloor a/b \rfloor$ for $a, b \in \mathbb{N}$. This operation has important applications to efficient modular arithmetic (see Section 2.5). Section 10.5 of [16] gives an excellent survey.

We now present an application of Newton's method to this problem. The idea is to to compute a good rational approximation to $1/a$ by finding a root of $F(x) = x^{-1} - a$.

Exercise 2.2.7. Show that the Newton iteration for $F(x) = x^{-1} - a$ is $x_{n+1} = 2x_n - ax_n^2$.

First we recall that a real number α can be represented by a rational approximation $b/2^e$ where $b, e \in \mathbb{Z}$. A key feature of this representation (based on the fact that division by powers of 2 is easy) is that if we know that $|\alpha - b/2^e| < 1/2^k$ (i.e., the result is correct to precision k) then we can renormalise the representation by replacing the approximation $b/2^e$ by $\lfloor b/2^{e-k} \rceil / 2^k$.

Suppose $2^m \leq a < 2^{m+1}$. Then we take $x_0 = b_0/2^{e_0} = 1/2^m$ as the first approximation to $1/a$. In other words, $b_0 = 1$ and $e_0 = m$. The Newton iteration in this case is

$e_{n+1} = 2e_n$ and $b_{n+1} = b_n(2^{e_n+1} - ab_n)$ which requires two integer multiplications. To prevent exponential growth of the numbers b_n one can renormalise the representation according to the expected precision of that step. One can show that the total complexity of getting an approximation to $1/a$ of precision m is $O(M(m))$ bit operations. For details see Section 3.5 of [556] (especially Exercise 3.35), Chapter 9 of [238] or, for a slightly different formulation, Section 9.2.2 of [162]. Applications of this idea to modular arithmetic will be given in Section 2.5.

Integer Approximations to Real Roots of Polynomials

Let $F(x) \in \mathbb{Z}[x]$. Approximations to roots of $F(x)$ in $\mathbb{R}$ can be computed using Newton's method. As a special case, integer square roots of m-bit numbers can be computed in time proportional to the cost of a multiplication of two m-bit numbers. Similarly, other roots (such as cube roots) can be computed in polynomial-time.

Exercise 2.2.8. Show that the Newton iteration for computing a square root of a is $x_{n+1} = (x_n + a/x_n)/2$. Hence, write down an algorithm to compute an integer approximation to the square root of a.

Exercise 2.2.8 can be used to test whether an integer a is a square. An alternative is to compute the Legendre symbol $(\frac{a}{p})$ for some random small primes p. For details see Exercise 2.4.9.

Exercise 2.2.9. Show that if $N = p^e$ where p is prime and $e \geq 1$ then one can factor N in polynomial-time.

2.3 Euclid's Algorithm

For $a, b \in \mathbb{N}$, Euclid's algorithm computes $d = \gcd(a, b)$. A simple way to express Euclid's algorithm is by the recursive formula

$$\gcd(a,b) = \begin{cases} \gcd(a,0) = a \\ \gcd(b, a \pmod b) & \text{if } b \neq 0. \end{cases}$$

The traditional approach is to work with positive integers a and b throughout the algorithm and to choose $a \pmod b$ to be in the set $\{0, 1, \ldots, b-1\}$. In practice, the algorithm can be used with $a, b \in \mathbb{Z}$ and it runs faster if we choose remainders in the range $\{-\lceil |b|/2 \rceil + 1, \ldots, -1, 0, 1, \ldots, \lceil |b|/2 \rceil\}$. However, for some applications (especially, those related to Diophantine approximation) the version with positive remainders is the desired choice.

In practice we often want to compute integers (s, t) such that $d = as + bt$, in which case we use the extended Euclidean algorithm. This is presented in Algorithm 1, where the integers r_i, s_i, t_i always satisfy $r_i = s_i a + t_i b$.

Theorem 2.3.1. *The complexity of Euclid's algorithm is $O(\log(a)\log(b))$ bit operations.*

Proof: Each iteration of Euclid's algorithm involves computing the quotient and remainder of division of r_{i-2} by r_{i-1} where we may assume $|r_{i-2}| > |r_{i-1}|$ (except maybe for $i = 1$). By Lemma 2.2.2 this requires $\leq c \log(r_{i-1})(\log(r_{i-2}) - \log(r_{i-1}) + 1)$ bit operations for some constant $c \in \mathbb{R}_{>0}$. Hence the total running time is at most

$$c \sum_{i \geq 1} \log(r_{i-1})(\log(r_{i-2}) - \log(r_{i-1}) + 1).$$

Algorithm 1 Extended Euclidean algorithm

INPUT: $a, b \in \mathbb{Z}$
OUTPUT: $d = \gcd(a, b)$ and $s, t \in \mathbb{Z}$ such that $d = sa + tb$
1: $r_{-1} = a,\ s_{-1} = 1,\ t_{-1} = 0$
2: $r_0 = b,\ s_0 = 0,\ t_0 = 1$
3: $i = 0$
4: **while** $(r_i \neq 0)$ **do**
5: $\quad i = i + 1$
6: $\quad$ find $q_i, r_i \in \mathbb{Z}$ such that $-|r_{i-1}|/2 < r_i \leq |r_{i-1}|/2$ and $r_{i-2} = q_i r_{i-1} + r_i$
7: $\quad s_i = s_{i-2} - q_i s_{i-1}$
8: $\quad t_i = t_{i-2} - q_i t_{i-1}$
9: **end while**
10: **return** $r_{i-1},\ s_{i-1},\ t_{i-1}$

Re-arranging terms gives

$$c \log(r_{-1}) \log(r_0) + c \sum_{i \geq 1} \log(r_{i-1})(1 + \log(r_i) - \log(r_{i-1})).$$

Now, $2|r_i| \leq |r_{i-1}|$ so $1 + \log(r_i) \leq \log(r_{i-1})$ hence all the terms in the above sum are ≤ 0. It follows that the algorithm performs $O(\log(a) \log(b))$ bit operations. □

Exercise 2.3.2. Show that the complexity of Algorithm 1 is still $O(\log(a) \log(b))$ bit operations even when the remainders in line 6 are chosen in the range $0 \leq r_i < r_{i-1}$.

A more convenient method for fast computer implementation is the binary Euclidean algorithm (originally due to Stein). This uses bit operations such as division by 2 rather than taking general quotients; see Section 4.5.2 of [343], Section 4.7 of [22], Chapter 3 of [238], Section 9.4.1 of [162] or Section 14.4.3 of [418].

There are subquadratic versions of Euclid's algorithm. One can compute the extended gcd of two n-bit integers in $O(M(n) \log(n))$ bit operations. We refer to Section 9.4 of [162], [583] or Section 11.1 of [238].

The rest of the section gives some results about Diophantine approximation that are used later (for example, in the Wiener attack on RSA, see Section 24.5.1). We assume that $a, b > 0$ and that the extended Euclidean algorithm with positive remainders is used to generate the sequence of values (r_i, s_i, t_i).

The integers s_i and t_i arising from the extended Euclidean algorithm are equal, up to sign, to the convergents of the continued fraction expansion of a/b. To be precise, if the convergents of a/b are denoted h_i/k_i for $i = 0, 1, \ldots$ then, for $i \geq 1$, $s_i = (-1)^{i-1} k_{i-1}$ and $t_i = (-1)^i h_{i-1}$. Therefore, the values (s_i, t_i) satisfy various equations, summarised below, that will be used later in the book. We refer to Chapter 10 of [276] or Chapter 7 of [468] for details on continued fractions.

Lemma 2.3.3. *Let $a, b \in \mathbb{N}$ and let $r_i, s_i, t_i \in \mathbb{Z}$ be the triples generated by running Algorithm 1 in the case of positive remainders $0 \leq r_i < r_{i-1}$.*

1. *For $i \geq 1$, $|s_i| < |s_{i+1}|$ and $|t_i| < |t_{i+1}|$.*
2. *If $a, b > 0$ then $t_i > 0$ when $i \geq 1$ is even and $t_i < 0$ when i is odd (and vice versa for s_i).*
3. *$t_{i+1} s_i - t_i s_{i+1} = (-1)^{i+1}$.*

4. $r_i s_{i-1} - r_{i-1} s_i = (-1)^i b$ *and* $r_i t_{i-1} - r_{i-1} t_i = (-1)^{i-1} a$. *In other words,* $r_i|s_{i-1}| + r_{i-1}|s_i| = b$ *and* $r_i|t_{i-1}| + r_{i-1}|t_i| = a$.
5. $|a/b + t_i/s_i| \leq 1/|s_i s_{i+1}|$.
6. $|r_i s_i| < |r_i s_{i+1}| \leq |b|$ *and* $|r_i t_i| < |r_i t_{i+1}| \leq |a|$.
7. *If* $s, t \in \mathbb{Z}$ *are such that* $|a/b + t/s| < 1/(2s^2)$ *then* (s, t) *is (up to sign) one of the pairs* (s_i, t_i) *computed by Euclid's algorithm.*
8. *If* $r, s, t \in \mathbb{Z}$ *satisfy* $r = as + bt$ *and* $|rs| < |b|/2$ *then* (r, s, t) *is (up to sign) one of the triples* (r_i, s_i, t_i) *computed by Euclid's algorithm.*

Proof: Statements 1, 2 and 3 are proved using the relation $s_i = (-1)^{i-1} k_{i-1}$ and $t_i = (-1)^i h_{i-1}$ where h_i/k_i are the continued fraction convergents to a/b. From Chapter 10 of [276] and Chapter 7 of [468] one knows that $h_m = q_{m+1} h_{m-1} + h_{m-2}$ and $k_m = q_{m+1} k_{m-1} + k_{m-2}$ where q_{m+1} is the quotient in iteration $m + 1$ of Euclid's algorithm. The first statement follows immediately and the third statement follows from the fact that $h_m k_{m-1} - h_{m-1} k_m = (-1)^{m-1}$. The second statement follows since $a, b > 0$ implies $h_i, k_i > 0$.

Statement 4 can be proved by induction, using the fact that $r_{i+1} s_i - r_i s_{i+1} = (r_{i-1} - q_i r_i) s_i - r_i (s_{i-1} - q_i s_i) = -(r_i s_{i-1} - r_{i-1} s_i)$. Statement 5 is the standard result (equation (10.7.7) of [276], Theorem 7.11 of [468]) that the convergents of a/b satisfy $|a/b - h_m/k_m| < 1/|k_m k_{m+1}|$. Statement 6 follows directly from statements 2 and 4. For example, $a = r_i (-1)^{i-1} t_{i-1} + r_{i-1} (-1)^i t_i$ and both terms on the right hand side are positive.

Statement 7 is also a standard result in Diophantine approximation; see Theorem 184 of [276] or Theorem 7.14 of [468].

Finally, to prove statement 8, suppose $r, s, t \in \mathbb{Z}$ are such that $r = as + bt$ and $|rs| < |b|/2$. Then

$$|a/b + t/s| = |(as + bt)/bs| = |r|/|bs| = |rs|/|bs^2| < 1/(2s^2).$$

The result follows from statement 7. □

Example 2.3.4. The first few terms of Euclid's algorithm on $a = 513$ and $b = 311$ give

i	r_i	q_i	s_i	t_i	$\lvert r_i s_i \rvert$	$\lvert r_i t_i \rvert$
-1	513	–	1	0	513	0
0	311	–	0	1	0	311
1	202	1	1	–1	202	202
2	109	1	–1	2	109	218
3	93	1	2	–3	186	279
4	16	1	–3	5	48	80
5	13	5	17	-28	221	364

One can verify that $|r_i s_i| \leq |b|$ and $|r_i t_i| \leq |a|$. Indeed, $|r_i s_{i+1}| \leq |b|$ and $|r_i t_{i+1}| \leq |a|$ as stated in part 6 of Lemma 2.3.3.

Diophantine approximation is the study of approximating real numbers by rationals. Statement 7 in Lemma 2.3.3 is a special case of one of the famous results; namely that the "best" rational approximations to real numbers are given by the convergents in their continued fraction expansion. Lemma 2.3.5 shows how the result can be relaxed slightly, giving "less good" rational approximations in terms of convergents to continued fractions.

Lemma 2.3.5. *Let* $\alpha \in \mathbb{R}$, $c \in \mathbb{R}_{>0}$ *and let* $s, t \in \mathbb{N}$ *be such that* $|\alpha - t/s| < c/s^2$. *Then* $(t, s) = (uh_{n+1} \pm vh_n, uk_{n+1} \pm vk_n)$ *for some* $n, u, v \in \mathbb{Z}_{\geq 0}$ *such that* $uv < 2c$.

Proof: See Theorem 1 and Remark 2 of Dujella [184]. □

We now remark that continued fractions allow one to compute solutions to Pell's equation.

Theorem 2.3.6. *Let $d \in \mathbb{N}$ be square-free. Then the continued fraction expansion of $\sqrt{d}$ is periodic; denote by r the period. Let h_n/k_n be the convergents in the continued fraction expansion of $\sqrt{d}$. Then $h_{nr-1}^2 - dk_{nr-1}^2 = (-1)^{nr}$ for $n \in \mathbb{N}$. Furthermore, every solution of the equation $x^2 - dy^2 = \pm 1$ arises in this way.*

Proof: See Corollary 7.23 of [468]. □

2.4 Computing Legendre and Jacobi Symbols

The Legendre symbol tells us when an integer is a square modulo p. It is a non-trivial group homomorphism from $(\mathbb{Z}/p\mathbb{Z})^*$ to the multiplicative group $\{-1, 1\}$.

Definition 2.4.1. Let p be an odd prime and $a \in \mathbb{Z}$. The **Legendre symbol** $(\frac{a}{p})$ is

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } x^2 \equiv a \pmod{p} \text{ has a solution.} \\ 0 & \text{if } p \mid a \\ -1 & \text{otherwise} \end{cases}$$

If p is prime and $a \in \mathbb{Z}$ satisfies $(\frac{a}{p}) = 1$ then a is a **quadratic residue**, while if $(\frac{a}{p}) = -1$ then a is a **quadratic non-residue**.

Let $n = \prod_i p_i^{e_i}$ be odd. The **Jacobi symbol** is

$$\left(\frac{a}{n}\right) = \prod_i \left(\frac{a}{p_i}\right)^{e_i}.$$

A further generalisation is the **Kronecker symbol** $(\frac{a}{n})$ which allows n to be even. This is defined in equation (25.4), which is the only place in the book that it is used.

Exercise 2.4.2. Show that if p is an odd prime then $(\frac{a}{p}) = 1$ for exactly half the integers $1 \leq a \leq p-1$.

Theorem 2.4.3. *Let $n \in \mathbb{N}$ be odd and $a \in \mathbb{Z}$. The Legendre and Jacobi symbols satisfy the following properties.*

- $(\frac{a}{n}) = (\frac{a \pmod n}{n})$ *and* $(\frac{1}{n}) = 1$.
- *(Euler's criterion) If n is prime then* $(\frac{a}{n}) = a^{(n-1)/2} \pmod n$.
- *(Multiplicative)* $(\frac{ab}{n}) = (\frac{a}{n})(\frac{b}{n})$ *for all* $a, b \in \mathbb{Z}$.
- $(\frac{-1}{n}) = (-1)^{(n-1)/2}$. *In other words*

$$\left(\frac{-1}{n}\right) = \begin{cases} 1 & \text{if } n \equiv 1 \pmod 4, \\ -1 & \text{otherwise} \end{cases}$$

- $(\frac{2}{n}) = (-1)^{(n^2-1)/8}$. *In other words*

$$\left(\frac{2}{n}\right) = \begin{cases} 1 & \text{if } n \equiv 1, 7 \pmod 8, \\ -1 & \text{otherwise} \end{cases}$$

- *(Quadratic reciprocity) Let n and m be odd integers with* $\gcd(m,n)=1$. *Then*

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

In other words, $(\frac{n}{m}) = (\frac{m}{n})$ *unless* $m \equiv n \equiv 3 \pmod 4$.

Proof: See Section II.2 of [348], Sections 3.1, 3.2 and 3.3 of [468] or Chapter 6 of [276]. □

An important fact is that it is not necessary to factor integers to compute the Jacobi symbol.

Exercise 2.4.4. Write down an algorithm to compute Legendre and Jacobi symbols using quadratic reciprocity.

Exercise 2.4.5. Prove that the complexity of computing $(\frac{m}{n})$ is $O(\log(m)\log(n))$ bit operations.

Exercise 2.4.6. Give a randomised algorithm to compute a quadratic non-residue modulo p. What is the expected complexity of this algorithm?

Exercise 2.4.7. Several applications require knowing a quadratic non-residue modulo a prime p. Prove that the values a in the following table satisfy $(\frac{a}{p}) = -1$.

p	a
$p \equiv 3 \pmod 4$	-1
$p \equiv 1 \pmod 4$, $p \equiv 2 \pmod 3$	3
$p \equiv 1 \pmod 4$, $p \not\equiv 1 \pmod 8$	$\sqrt{-1}$
$p \equiv 1 \pmod 8$, $p \not\equiv 1 \pmod{16}$	$(1+\sqrt{-1})/\sqrt{2}$

Remark 2.4.8. The problem of computing quadratic non-residues has several algorithmic implications. One conjectures that the least quadratic non-residue modulo p is $O(\log(p)\log(\log(p)))$. Burgess proved that the least quadratic non-residue modulo p is at most $p^{1/(4\sqrt{e})+o(1)} \approx p^{0.151633+o(1)}$ while Ankeny showed, assuming the extended Riemann hypothesis, that it is $O(\log(p)^2)$. We refer to Section 8.5 of Bach and Shallit [22] for details and references. It follows that one can compute a quadratic non-residue in $O(\log(p)^4)$ bit operations assuming the extended Riemann hypothesis.

Exercise 2.4.9. Give a Las Vegas algorithm to test whether $a \in \mathbb{N}$ is a square by computing $(\frac{a}{p})$ for some random small primes p. What is the complexity of this algorithm?

Exercise 2.4.10. Let p be prime. In Section 2.8 we give algorithms to compute modular exponentiation quickly. Compare the cost of computing $(\frac{a}{p})$ using quadratic reciprocity versus using Euler's criterion.

Remark 2.4.11. An interesting computational problem (considered, for example, by Damgård [164]) is: given a prime p an integer k and the sequence $(\frac{a}{p}), (\frac{a+1}{p}), \ldots, (\frac{a+k-1}{p})$ to output $(\frac{a+k}{p})$. A potentially harder problem is to determine a given the sequence of values. It is known that if k is a little larger than $\log_2(p)$ then a is usually uniquely determined modulo p and so both problems make sense. No efficient algorithms are known to solve either of these problems. One can also consider the natural analogue for Jacobi symbols. We refer to [164] for further details. This is also discussed as Conjecture 2.1 of Boneh and Lipton [83]. The pseudorandomness of the sequence is discussed by Mauduit and Sárközy [401] and Sárközy and Stewart [510].

Finally, we remark that one can compute the Legendre or Jacobi symbol of n-bit integers in $O(M(n)\log(n))$ operations using an analogue of fast algorithms for computing gcds. We refer to Exercise 5.52 (also see pages 343-344) of Bach and Shallit [22] or Brent and Zimmermann [101] for the details.

2.5 Modular Arithmetic

In cryptography, modular arithmetic (i.e., arithmetic modulo $n \in \mathbb{N}$) is a fundamental building block. We represent elements of $\mathbb{Z}/n\mathbb{Z}$ as integers from the set $\{0, 1, \ldots, n-1\}$. We first summarise the complexity of standard "school" methods for modular arithmetic.

Lemma 2.5.1. *Let $a, b \in \mathbb{Z}/n\mathbb{Z}$.*

1. *Computing $a \pm b \pmod n$ can be done in $O(\log(n))$ bit operations.*
2. *Computing $ab \pmod n$ can be done in $O(\log(n)^2)$ bit operations.*
3. *Computing $a^{-1} \pmod n$ can be done in $O(\log(n)^2)$ bit operations.*
4. *For $a \in \mathbb{Z}$ computing $a \pmod n$ can be done in $O(\log(n)(\log(a) - \log(n) + 1))$ bit operations.*

Montgomery Multiplication

This method[6] is useful when one needs to perform an operation such as $a^m \pmod n$ when n is odd. It is based on the fact that arithmetic modulo 2^s is easier than arithmetic modulo n. Let $R = 2^s > n$ (where s is typically a multiple of the word size).

Definition 2.5.2. Let $n \in \mathbb{N}$ be odd and $R = 2^s > n$. The **Montgomery representation** of $a \in (\mathbb{Z}/n\mathbb{Z})$ is $\overline{a} = aR \pmod n$ such that $0 \leq \overline{a} < n$.

To transform a into Montgomery representation requires a standard modular multiplication. However, Lemma 2.5.3 shows that transforming back from Montgomery representation to standard representation may be performed more efficiently.

Lemma 2.5.3. *(Montgomery reduction) Let $n \in \mathbb{N}$ be odd and $R = 2^s > n$. Let $n' = -n^{-1} \pmod R$ be such that $1 \leq n' < R$. Let $\overline{a}$ be an element of $(\mathbb{Z}/n\mathbb{Z})$ in Montgomery representation. Let $u = \overline{a}n' \pmod R$. Then $w = (\overline{a} + un)/R$ lies in $\mathbb{Z}$ and satisfies $w \equiv \overline{a}R^{-1} \pmod n$.*

Proof: Write $w' = \overline{a} + un$. Clearly $w' \equiv 0 \pmod R$ so $w \in \mathbb{Z}$. Further, $0 \leq w' \leq (n-1) + (R-1)n = Rn - 1$ and so $w < n$. Finally, it is clear that $w \equiv \overline{a}R^{-1} \pmod n$. □

The reason why this is efficient is that division by R is easy. The computation of n' is also easier than a general modular inversion (see Algorithm II.5 of [64]) and, in many applications, it can be precomputed.

We now sketch the **Montgomery multiplication** algorithm. If $\overline{a}$ and $\overline{b}$ are in Montgomery representation then we want to compute the Montgomery representation of ab, which is $\overline{a}\overline{b}R^{-1} \pmod n$. Compute $x = \overline{a}\overline{b} \in \mathbb{Z}$ so that $0 \leq x < n^2 < nR$, then compute $u = xn' \pmod R$ and $w' = x + nu \in \mathbb{Z}$. As in Lemma 2.5.3 we have $w' \equiv 0 \pmod R$ and can compute $w = w'/R$. It follows that $w \equiv \overline{a}\overline{b}R^{-1} \pmod n$ and $0 \leq w < 2n$ so $\overline{ab}$ is either w or $w - n$.

Lemma 2.5.4. *The complexity of Montgomery multiplication modulo n is $O(M(\log(n)))$ bit operations.*

[6] Credited to Montgomery [435], but apparently a similar idea was used by Hensel.

For further details see Section 9.2.1 of [162], Section II.1.4 of [64], Section 11.1.2.b of [16] or Section 2.2.4 of [274].

Faster Modular Reduction

Using Newton's method to compute $\lfloor a/n \rfloor$ one can compute $a \pmod n$ using only multiplication of integers. If $a = O(n^2)$ then the complexity is $O(M(\log(n)))$. The basic idea is to use Newton's method to compute a rational approximation to $1/a$ of the form $b/2^e$ (see Section 2.2.1) and then compute $q = \lfloor n/a \rfloor = \lfloor nb/2^e \rfloor$ and thus $r = a - nq$ is the remainder. See Exercises 3.35, 3.36 of [556] and Section 9.1 of [238] for details. For large a the cost of computing $a \pmod n$ remains $O(\log(a)\log(n))$ as before. This idea gives rise to Barret reduction; see Section 9.2.2 of [162], Section 2.3.1 of [100], Section 14.3.3 of [418], Section II.1.3 of [64], or Section 10.4.1 of [16].

Special Moduli

For cryptography based on discrete logarithms, especially elliptic curve cryptography, it is recommended to use primes of a special form to speed up arithmetic modulo p. Commonly used primes are of the form $p = 2^k - c$ for some small $c \in \mathbb{N}$ or the **NIST primes** $p = 2^{n_k w} \pm 2^{n_{k-1} w} \pm \cdots \pm 2^{n_1 w} \pm 1$ where $w = 16, 32$ or 64. In these cases it is possible to compute reduction modulo p much more quickly than for general p. See Section 2.2.6 of [274], Section 14.3.4 of [418] or Section 10.4.3 of [16] for examples and details.

Modular Inversion

Suppose that $a, n \in \mathbb{N}$ are such that $\gcd(a, n) = 1$. One can compute $a^{-1} \pmod n$ using the extended Euclidean algorithm: computing integers $s, t \in \mathbb{Z}$ such that $as + nt = 1$ gives $a^{-1} \equiv s \pmod n$. Hence, if $0 < a < n$ then one can compute $a^{-1} \pmod n$ in $O(\log(n)^2)$ bit operations, or faster using subquadratic versions of the extended Euclidean algorithm.

In practice, modular inversion is significantly slower than modular multiplication. For example, when implementing elliptic curve cryptography it is usual to assume that the cost of an inversion in $\mathbb{F}_p$ is between 8 and 50 times slower than the cost of a multiplication in $\mathbb{F}_p$ (the actual figure depends on the platform and algorithms used).

Simultaneous Modular Inversion

One can compute $a_1^{-1} \pmod n, \ldots, a_m^{-1} \pmod n$ with a single inversion modulo n and a number of multiplications modulo n using a trick due to Montgomery. Namely, one computes $b = a_1 \cdots a_m \pmod n$, computes $b^{-1} \pmod n$, and then recovers the individual a_i^{-1}.

Exercise 2.5.5. Give pseudocode for simultaneous modular inversion and show that it requires one inversion and $3(m-1)$ modular multiplications.

2.6 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT), so-called because it was first discovered by the Chinese mathematician Sunzi, states that if $\gcd(m_1, m_2) = 1$ then there is a unique

solution $0 \leq x < m_1 m_2$ to $x \equiv c_i \pmod{m_i}$ for $i = 1, 2$. Computing x can be done in polynomial-time in various ways. One method is to use the formula

$$x = c_1 + (c_2 - c_1)(m_1^{-1} \pmod{m_2}) m_1.$$

This is a special case of Garner's algorithm (see Section 14.5.2 of [418] or Section 10.6.4 of [16]).

Exercise 2.6.1. Suppose $m_1 < m_2$ and $0 \leq c_i < m_i$. What is the input size of the instance of the CRT? What is the complexity of computing the solution?

Exercise 2.6.2. Let $n > 2$ and suppose coprime integers $2 \leq m_1 < \cdots < m_n$ and integers $c_1, \ldots, c_n$ such that $0 \leq c_i < m_i$ for $1 \leq i \leq n$ are given. Let $N = \prod_{i=1}^{n} m_i$. For $1 \leq i \leq n$ define $N_i = N/m_i$ and $u_i = N_i^{-1} \pmod{m_i}$. Show that

$$x = \sum_{i=1}^{n} c_i u_i N_i \tag{2.1}$$

satisfies $x \equiv c_i \pmod{m_i}$ for all $1 \leq i \leq n$.

Show that one can compute the integer x in equation (2.1) in $O(n^2 \log(m_n)^2)$ bit operations.

Exercise 2.6.3. Show that a special case of Exercise 2.6.2 (which is recommended when many computations are required for the same pair of moduli) is to pre-compute the integers $A = u_1 N_1$ and $B = u_2 N_2$ so that $x = c_1 A + c_2 B \pmod{N}$.

Algorithm 10.22 of [238] gives an asymptotically fast CRT algorithm.

Exercise 2.6.4 gives a variant of the Chinese remainder theorem, which seems to originate in work of Montgomery and Silverman, called the **explicit Chinese remainder theorem**. This variant is useful when one wants to compute the large integer x modulo a smaller integer p and one wants to minimise the overall storage. For a more careful complexity analysis see Section 6 of Sutherland [597]; for small p he shows that the explicit CRT can be computed in $O(nM(\log(p)) + M(\log(N) + n\log(n)))$ bit operations.

Exercise 2.6.4. Let the notation be as in Exercise 2.6.2 and let p be coprime to N. The goal is to compute $x \pmod{p}$ where x is an integer such that $x \equiv c_i \pmod{m_i}$ for $1 \leq i \leq n$ and $|x| < N/2$.

Let $z = \sum_{i=1}^{n} c_i u_i / m_i \in \mathbb{Q}$. Show that $2z \notin \mathbb{Z}$ and that $0 \leq z < n m_n$. Show that the solution x to the congruences satisfying $|x| < N/2$ is equal to $Nz - N\lfloor z \rceil$.

Hence, show that

$$x \equiv \sum_{i=1}^{n} (c_i u_i (N_i \pmod{p}) \pmod{p}) - (N \pmod{p})(\lfloor z \rceil \pmod{p}) \pmod{p}. \tag{2.2}$$

Show that one can therefore compute x using equation (2.2) and representing z as a floating point number in a way that does not require knowing more than one of the values c_i at a time. Show that one can precompute $N \pmod{p}$ and $N_i \pmod{p}$ for $1 \leq i \leq n$ in $O(n(\log(m_n)\log(p) + M(\log(p)))$ bit operations. Hence show that the complexity of solving the explicit CRT is (assuming the floating point operations can be ignored) at most $O(n(\log(m_n)\log(p) + M(\log(p))))$ bit operations.

2.7 Linear Algebra

Let A be an $n \times n$ matrix over a field $\Bbbk$. One can perform Gaussian elimination to solve the linear system $A\underline{x} = \underline{b}$ (or determine there are no solutions), to compute $\det(A)$, or to compute A^{-1} in $O(n^3)$ field operations. When working over $\mathbb{R}$ a number of issues arise due to rounding errors, but no such problems arise when working over finite fields. We refer to Section 3.3 of Joux [317] for details.

A matrix is called **sparse** if almost all entries of each row are zero. To make this precise one usually considers the asymptotic complexity of an algorithm on $m \times n$ matrices, as m and/or n tends to infinity, and where the number of non-zero entries in each row is bounded by $O(\log(n))$ or $O(\log(m))$.

One can compute the kernel (i.e., a vector $\underline{x}$ such that $A\underline{x} = \underline{0}$) of an $n \times n$ sparse matrix A over a field in $O(n^2)$ field operations using the algorithms of Wiedemann [629] or Lanczos [363]. We refer to Section 3.4 of [317] or Section 12.4 of [238] for details.

Hermite Normal Form

When working over a ring the Hermite normal form (HNF) is an important tool for solving or simplifying systems of equations. Some properties of the Hermite normal form are mentioned in Section A.11.

Algorithms to compute the HNF of a matrix are given in Section 2.4.2 of Cohen [136], Hafner and McCurley [273], Section 3.3.3 of Joux [317], Algorithm 16.26 of von zur Gathen and Gerhard [238], Section 5.3 of Schrijver [531], Kannan and Bachem [331], Storjohann and Labahn [593], and Micciancio and Warinschi [425]. Naive algorithms to compute the HNF suffer from coefficient explosion, so computing the HNF efficiently in practice, and determining the complexity of the algorithm, is non-trivial. One solution is to work modulo the determinant (or a sub-determinant) of the matrix A (see Section 2.4.2 of [136], [273] or [593] for further details). Let $A = (A_{i,j})$ be an $n \times m$ matrix over $\mathbb{Z}$ and define $\|A\|_\infty = \max_{i,j}\{|A_{i,j}|\}$. The complexity of the HNF algorithm of Storjohann and Labahn on A (using naive integer and matrix multiplication) is $O(nm^4 \log(\|A\|_\infty)^2)$ bit operations.

One can also use lattice reduction to compute the HNF of a matrix. For details see page 74 of [531], Havas, Majewski and Matthews [280], or van der Kallen [328].

2.8 Modular Exponentiation

Exponentiation modulo n can be performed in polynomial-time by the "square-and-multiply" method.[7] This method is presented in Algorithm 2; it is called a "left-to-right" algorithm as it processes the bits of the exponent m starting with the most significant bits. Algorithm 2 can be applied in any group, in which case the complexity is $O(\log(m))$ times the complexity of the group operation. In this section we give some basic techniques to speed-up the algorithm; further tricks are described in Chapter 11.

Lemma 2.8.1. *The complexity of Algorithm 2 using naive modular arithmetic is* $O(\log(m)\log(n)^2)$ *bit operations.*

Exercise 2.8.2. Prove Lemma 2.8.1.

Lemma 2.8.3. *If Montgomery multiplication (see Section 2.5) is used then the complexity of Algorithm 2.5 is* $O(\log(n)^2 + \log(m)M(\log(n)))$.

[7] This algorithm already appears in the *chandah-sūtra* by Pingala.

Algorithm 2 Square-and-multiply algorithm for modular exponentiation

INPUT: $g, n, m \in \mathbb{N}$
OUTPUT: $b \equiv g^m \pmod{n}$
1: $i = \lfloor \log_2(m) \rfloor - 1$
2: Write m in binary as $(1m_i \dots m_1 m_0)_2$
3: $b = g$
4: **while** $(i \geq 0)$ **do**
5: $\quad b = b^2 \pmod{n}$
6: $\quad$ **if** $m_i = 1$ **then**
7: $\quad\quad b = bg \pmod{n}$
8: $\quad$ **end if**
9: $\quad i = i - 1$
10: **end while**
11: **return** b

Proof: Convert g to Montgomery representation $\overline{g}$ in $O(\log(n)^2)$ bit operations. Algorithm 2 then proceeds using Montgomery multiplication in lines 5 and 7, which requires $O(\log(m)M(\log(n)))$ bit operations. Finally Montgomery reduction is used to convert the output to standard form. □

The algorithm using Montgomery multiplication is usually better than the naive version, especially when fast multiplication is available. An application of the above algorithm, where Karatsuba multiplication would be appropriate, is RSA decryption (either the standard method, or using the CRT). Since $\log(m) = \Omega(\log(n))$ in this case, decryption requires $O(\log(n)^{2.585})$ bit operations.

Corollary 2.8.4. *One can compute the Legendre symbol* $(\frac{a}{p})$ *using Euler's criterion in* $O(\log(p)M(\log(p)))$ *bit operations.*

When storage for precomputed group elements is available there are many ways to speed up exponentiation. These methods are particularly appropriate when many exponentiations of a fixed element g are required. The methods fall naturally into two types: those that reduce the number of squarings in Algorithm 2 and those that reduce the number of multiplications. An extreme example of the first type is to precompute and store $u_i = g^{2^i} \pmod{n}$ for $2 \leq i \leq \log(n)$. Given $2^l \leq m < 2^{l+1}$ with binary expansion $(1m_{l-1} \dots m_1 m_0)_2$ one computes $\prod_{i=0:m_i=1}^{l} u_i \pmod{n}$. Obviously this method is not more efficient than Algorithm 2 if g varies. An example of the second type is **sliding window methods** that we now briefly describe. Note that there is a simpler but less efficient "non-sliding" window method, also called the 2^k-ary method, which can be found in many books. Sliding window methods can be useful even in the case where g varies (e.g., Algorithm 3 below).

Given a **window length** w one precomputes $u_i = g^i \pmod{n}$ for all odd integers $1 \leq i < 2^w$. Then one runs a variant of Algorithm 2 where w (or more) squarings are performed followed by one multiplication corresponding to a w-bit sub-string of the binary expansion of m that corresponds to an odd integer. One subtlety is that algorithms based on the "square-and-multiply" idea and which use pre-computation must parse the exponent starting with the most significant bits (i.e., from left to right) whereas to work out sliding windows one needs to parse the exponent from the least significant bits (i.e., right to left).

Example 2.8.5. Let $w = 2$ so that one precomputes $u_1 = g$ and $u_3 = g^3$. Suppose m has binary expansion $(10011011)_2$. By parsing the binary expansion starting with the

least significant bits one obtains the representation 10003003 (we stress that this is still a representation in base 2). One then performs the usual square-and-multiply algorithm by parsing the exponent from left to right; the steps of the sliding window algorithm are (omitting the $\pmod n$ notation)

$$b = u_1,\ b = b^2;\ b = b^2,\ b = b^2,\ b = b^2,\ b = bu_3,\ b = b^2,\ b = b^2,\ b = b^2,\ b = bu_3.$$

Exercise 2.8.6. Write pseudocode for the sliding window method. Show that the precomputation stage requires one squaring and $2^{w-1} - 1$ multiplications.

Exercise 2.8.7. Show that the expected number of squarings between each multiply in the sliding window algorithm is $w + 1$. Hence show that (ignoring the precomputation) exponentiation using sliding windows requires $\log(m)$ squarings and, on average, $\log(m)/(w + 1)$ multiplications.

Exercise 2.8.8. Consider running the sliding window method in a group, with varying g and m (so the powers of g must be computed for every exponentiation) but with unlimited storage. For a given bound on $\text{len}(m)$ one can compute the value for w that minimises the total cost. Verify that the choices in the following table are optimal.

$\text{len}(m)$	80	160	300	800	2000
w	3	4	5	6	7

Exercise 2.8.9. Algorithm 2 processes the bits of the exponent m from left to right. Give pseudocode for a modular exponentiation algorithm that processes the bits of the exponent m from right to left.
[Hint: Have two variables in the main loop; one that stores g^{2^i} in the i-th iteration, and the other that stores the value $g^{\sum_{j=0}^{i} a_j 2^j}$.]

Exercise 2.8.10. Write pseudocode for a right to left sliding window algorithm for computing $g^m \pmod n$, extending Exercise 2.8.9. Explain why this variant is not appropriate when using precomputation (hence, it is not so effective when computing $g^m \pmod n$ for many random m but when g is fixed).

One can also consider the opposite scenario where one is computing $g^m \pmod n$ for a fixed value m and varying g. Again, with some precomputation, and if there is sufficient storage available, one can get an improvement over the naive algorithm. The idea is to determine an efficient **addition chain** for m. This is a sequence of squarings and multiplications, depending on m, that minimises the number of group operations. More precisely, an addition chain of length l for m is a sequence $m_1, m_2, \ldots, m_l$ of integers such that $m_1 = 1$, $m_l = m$ and, for each $2 \leq i \leq l$ we have $m_i = m_j + m_k$ for some $1 \leq j \leq k < i$. One computes each of the intermediate values g^{m_i} for $2 \leq i \leq l$ with one group operation. Note that *all* these intermediate values are stored. The algorithm requires l group operations and l group elements of storage.

It is conjectured by Stolarsky that every integer m has an addition chain of length $\log_2(m) + \log_2(\text{wt}(m))$ where $\text{wt}(m)$ is the **Hamming weight** of m (i.e., the number of ones in the binary expansion of m). There is a vast literature on addition chains, we refer to Section C6 of [272], Section 4.6.3 of [343] and Section 9.2 of [16] for discussion and references.

Exercise 2.8.11. Prove that an addition chain has length at least $\log_2(m)$.

2.9 Square Roots Modulo p

There are a number of situations in this book that require computing square roots modulo a prime. Let p be an odd prime and let $a \in \mathbb{N}$. We have already shown that Legendre symbols can be computed in polynomial-time. Hence, the decision problem "Is a a square modulo p?" is soluble in polynomial-time. But this fact does not imply that the computational problem "Find a square root of a modulo p" is easy.

We present two methods in this section. The Tonelli-Shanks algorithm [549] is the best method in practice. The Cipolla algorithm actually has better asymptotic complexity, but is usually slower than Tonelli-Shanks.

Recall that half the integers $1 \leq a < p$ are squares modulo p and, when a is square, there are two solutions $\pm x$ to the equation $x^2 \equiv a \pmod{p}$.

Lemma 2.9.1. *Let $p \equiv 3 \pmod 4$ be prime and $a \in \mathbb{N}$. If $(\frac{a}{p}) = 1$ then $x = a^{(p+1)/4} \pmod p$ satisfies $x^2 \equiv a \pmod p$.*

This result can be verified directly by computing x^2, but we give a more group-theoretic proof that helps to motivate the general algorithm.

Proof: Since $p \equiv 3 \pmod 4$ it follows that $q = (p-1)/2$ is odd. The assumption $(\frac{a}{p}) = 1$ implies that $a^q = a^{(p-1)/2} \equiv 1 \pmod p$ and so the order of a is odd. Therefore a square root of a is given by

$$x = a^{2^{-1} \pmod q} \pmod p.$$

Now, $2^{-1} \pmod q$ is just $(q+1)/2 = (p+1)/4$. □

Lemma 2.9.2. *Let p be a prime and suppose that a is a square modulo p. Write $p-1 = 2^e q$ where q is odd. Let $w = a^{(q+1)/2} \pmod p$. Then $w^2 \equiv ab \pmod p$ where b has order dividing 2^{e-1}.*

Proof: We have

$$w^2 \equiv a^{q+1} \equiv aa^q \pmod p$$

so $b \equiv a^q \pmod p$. Now a has order dividing $(p-1)/2 = 2^{e-1}q$ so b has order dividing 2^{e-1}. □

The value w is like a "first approximation" to the square root of a modulo p. To complete the computation it is therefore sufficient to compute a square root of b.

Lemma 2.9.3. *Suppose $1 < n < p$ is such that $(\frac{n}{p}) = -1$. Then $y \equiv n^q \pmod p$ has order 2^e.*

Proof: The order of y is a divisor of 2^e. The fact $n^{(p-1)/2} \equiv -1 \pmod p$ implies that y satisfies $y^{2^{e-1}} \equiv -1 \pmod p$. Hence the order of y is equal to 2^e. □

Since $\mathbb{Z}_p^*$ is a cyclic group, it follows that y generates the full subgroup of elements of order dividing 2^e. Hence, $b = y^i \pmod p$ for some $1 \leq i \leq 2^e$. Furthermore, since the order of b divides 2^{e-1} it follows that i is even.

Writing $i = 2j$ and $x = w/y^j \pmod p$ then

$$x^2 \equiv w^2/y^{2j} \equiv ab/b \equiv a \pmod p.$$

Hence, if one can compute i then one can compute the square root of a.

If e is small then the value i can be found by a brute-force search. A more advanced method is to use the Pohlig-Hellman method to solve the discrete logarithm of b to the base y (see Section 13.2 for an explanation of these terms). This idea leads to the Tonelli-Shanks algorithm for computing square roots modulo p (see Section 1.3.3 of [64] or Section 1.5 of [136]).

Algorithm 3 Tonelli-Shanks algorithm

INPUT: a, p such that $(\frac{a}{p}) = 1$
OUTPUT: x such that $x^2 \equiv a \pmod{p}$
1: Write $p - 1 = 2^e q$ where q is odd
2: Choose random integers $1 < n < p$ until $(\frac{n}{p}) = -1$
3: Set $y = n^q \pmod{p}$
4: Set $w = a^{(q+1)/2} \pmod{p}$ and $b = a^q \pmod{p}$
5: Compute an integer j such that $b \equiv y^{2j} \pmod{p}$
6: **return** $w/y^j \pmod{p}$

Exercise 2.9.4. Compute $\sqrt{3}$ modulo 61 using the Tonelli-Shanks algorithm.

Lemma 2.9.5. *The Tonelli-Shanks method is a Las Vegas algorithm with expected running time $O(\log(p)^2 M(\log(p)))$ bit operations.*

Proof: The first step of the algorithm is the requirement to find an integer n such that $(\frac{n}{p}) = -1$. This is Exercise 2.4.6 and it is the only part of the algorithm that is randomised and Las Vegas. The expected number of trials is 2. Since one can compute the Legendre symbol in $O(\log(p)^2)$ bit operations, this gives $O(\log(p)^2)$ expected bit operations, which is less than $O(\log(p)M(\log(p)))$.

The remaining parts of the algorithm amount to exponentiation modulo p, requiring $O(\log(p)M(\log(p)))$ bit operations, and the computation of the index j. Naively, this could require as many as $p - 1$ operations, but using the Pohlig-Hellman method (see Exercise 13.2.6 in Section 13.2) brings the complexity of this stage to $O(\log(p)^2 M(\log(p)))$ bit operations. □

As we will see in Exercise 2.12.6, the worst-case complexity $O(\log(p)^2 M(\log(p)))$ of the Tonelli-Shanks algorithm is actually worse than the cost of factoring quadratic polynomials using general polynomial-factorisation algorithms. But, in most practical situations, the Tonelli-Shanks algorithm is faster than using polynomial factorisation.

Exercise 2.9.6. If one precomputes y for a given prime p then the square root algorithm becomes deterministic. Show that the complexity remains the same.

Exercise 2.9.7. Show, using Remark 2.4.8, that under the extended Riemann hypothesis one can compute square roots modulo p in deterministic $O(\log(p)^4)$ bit operations.

Exercise 2.9.8. Let $r \in \mathbb{N}$. Generalise the Tonelli-Shanks algorithm so that it computes r-th roots in $\mathbb{F}_p$ (the only non-trivial case being when $p \equiv 1 \pmod{r}$).

Exercise 2.9.9. (Atkin) Let $p \equiv 5 \pmod{8}$ be prime and $a \in \mathbb{Z}$ such that $(\frac{a}{p}) = 1$. Let $z = (2a)^{(p-5)/8} \pmod{p}$ and $i = 2az^2 \pmod{p}$. Show that $i^2 = -1 \pmod{p}$ and that $w = az(i - 1)$ satisfies $w^2 \equiv a \pmod{p}$.

If $p - 1$ is highly divisible by 2 then an algorithm due to Cipolla, sketched in Exercise 2.9.10 below, is more suitable (see Section 7.2 of [22] or Section 3.5 of [418]). See Bernstein [44] for further discussion. There is a completely different algorithm due to Schoof that is deterministic and has polynomial-time complexity for fixed a as p tends to infinity.

Exercise 2.9.10. (Cipolla) Let p be prime and $a \in \mathbb{Z}$. Show that if $t \in \mathbb{Z}$ is such that $(\frac{t^2-4a}{p}) = -1$ then $x^{(p+1)/2}$ in $\mathbb{F}_p[x]/(x^2 - tx + a)$ is a square root of a modulo p. Hence write down an algorithm to compute square roots modulo p and show that it has expected running time $O(\log(p)M(\log(p)))$ bit operations.

We remark that, in some applications, one wants to compute a Legendre symbol to test whether an element is a square and, if so, compute the square root. If one computes the Legendre symbol using Euler's criterion as $a^{(p-1)/2} \pmod p$ then one will have already computed $a^q \pmod p$ and so this value can be recycled. This is not usually faster than using quadratic reciprocity for large p, but it is relevant for applications such as Lemma 21.4.9.

A related topic is, given a prime p and an integer $d > 0$, to find integer solutions (x, y), if they exist, to the equation $x^2 + dy^2 = p$. The **Cornacchia algorithm** achieves this. The algorithm is given in Section 2.3.4 of Crandall and Pomerance [162], and a proof of correctness is given in Section 4 of Schoof [530] or Morain and Nicolas [437]. In brief, the algorithm computes $p/2 < x_0 < p$ such that $x_0^2 \equiv -d \pmod p$, then runs the Euclidean algorithm on $2p$ and x_0 stopping at the first remainder $r < \sqrt{p}$, then computes $s = \sqrt{(p - r^2)/d}$ if this is an integer. The output is $(x, y) = (r, s)$. The complexity is dominated by computing the square root modulo p, and so is an expected $O(\log(p)^2 M(\log(p)))$ bit operations. A closely related algorithm finds solutions to $x^2 + dy^2 = 4p$.

2.10 Polynomial Arithmetic

Let R be a commutative ring. A polynomial in $R[x]$ of degree d is represented[8] as a $(d+1)$-tuple over R. A polynomial of degree d over $\mathbb{F}_q$ therefore requires $(d+1)\lceil \log_2(q) \rceil$ bits for its representation. An algorithm on polynomials will be polynomial-time if the number of bit operations is bounded above by a polynomial in $d \log(q)$.

Arithmetic on polynomials is analogous to integer arithmetic (indeed, it is simpler as there are no carries to deal with). We refer to Chapter 2 of [238], Chapter 18 of [556], Section 4.6 of [343] or Section 9.6 of [162] for details.

Lemma 2.10.1. *Let R be a commutative ring and $F_1(x), F_2(x) \in R[x]$.*

1. *One can compute $F_1(x) + F_2(x)$ in $O(\max\{\deg(F_1), \deg(F_2)\})$ additions in R.*
2. *One can compute $F_1(x)F_2(x)$ in $O(\deg(F_1)\deg(F_2))$ additions and multiplications in R.*
3. *If R is a field one can compute the quotient and remainder of division of $F_1(x)$ by $F_2(x)$ in $O(\deg(F_2)(\deg(F_1) - \deg(F_2) + 1))$ operations (i.e., additions, multiplications and inversions) in R.*
4. *If R is a field one can compute $F(x) = \gcd(F_1(x), F_2(x))$ and polynomials $s(x), t(x) \in R[x]$ such that $F(x) = s(x)F_1(x) + t(x)F_2(x)$, using the extended Euclidean algorithm in $O(\deg(F_1)\deg(F_2))$ operations in R.*

Exercise 2.10.2. Prove Lemma 2.10.1.

Exercise 2.10.3.★ Describe the Karatsuba and 3-Toom-Cook algorithms for multiplication of polynomials of degree d in $\mathbb{F}_q[x]$. Show that these algorithms have complexity $O(d^{1.585})$ and $O(d^{1.404})$ multiplications in $\mathbb{F}_q$.

Asymptotically fast multiplication of polynomials, analogous to the algorithms mentioned in Section 2.2, are given in Chapter 8 of [238] or Section 18.6 of [556]. Multiplication of polynomials in $\Bbbk[x]$ of degree bounded by d can be done in $O(M(d))$ multiplications in $\Bbbk$. The methods mentioned in Section 2.5 for efficiently computing remainders

[8] We restrict attention in this and the following section to univariate polynomials. There are alternative representations for sparse and/or multivariate polynomials, but we do not consider this issue further.

$F(x) \pmod{G(x)}$ in $\Bbbk[x]$ can also be used with polynomials; see Section 9.6.2 of [162] or Section 11.1 of [238] for details. Fast variants of algorithms for the extended Euclidean algorithm for polynomials in $\Bbbk[x]$ of degree bounded by d require $O(M(d)\log(d))$ multiplications in $\Bbbk$ and $O(d)$ inversions in $\Bbbk$ (Corollary 11.6 of [238]).

Kronecker substitution is a general technique which transforms polynomial multiplication into integer multiplication. It allows multiplication of two degree d polynomials in $\mathbb{F}_q[x]$ (where q is prime) in $O(M(d(\log(q)+\log(d)))) = O(M(d\log(dq)))$ bit operations; see Section 1.3 of [100], Section 8.4 of [238] or Section 18.6 of [556]. Kronecker substitution can be generalised to bivariate polynomials and hence to polynomials over $\mathbb{F}_q$ where q is a prime power. We write $M(d,q) = M(d\log(dq))$ for the number of bit operations to multiply two degree d polynomials over $\mathbb{F}_q$.

Exercise 2.10.4. Show that Montgomery reduction and multiplication can be implemented for arithmetic modulo a polynomial $F(x) \in \mathbb{F}_q[x]$ of degree d.

Exercise 2.10.5. One can evaluate a polynomial $F(x) \in R[x]$ at a value $a \in R$ efficiently using **Horner's rule**. More precisely, if $F(x) = \sum_{i=0}^{d} F_i x^i$ then one computes $F(a)$ as $(\cdots((F_d a)a + F_{d-1})a + \cdots + F_1)a + F_0$. Write pseudocode for Horner's rule and show that the method requires d additions and d multiplications if $\deg(F(x)) = d$.

2.11 Arithmetic in Finite Fields

Efficient algorithms for arithmetic modulo p have been presented, but we now consider arithmetic in finite fields $\mathbb{F}_{p^m}$ when $m > 1$. We assume $\mathbb{F}_{p^m}$ is represented using either a polynomial basis (i.e., as $\mathbb{F}_p[x]/(F(x))$) or a normal basis. Our main focus is when either p is large and m is small, or vice versa. Optimal asymptotic complexities for the case when both p and m grow large require some care.

Exercise 2.11.1. Show that addition and subtraction of elements in $\mathbb{F}_{p^m}$ requires $O(m)$ additions in $\mathbb{F}_p$. Show that multiplication in $\mathbb{F}_{p^m}$, represented by a polynomial basis and using naive methods, requires $O(m^2)$ multiplications modulo p and $O(m)$ reductions modulo p.

If p is constant and m grows then multiplication in $\mathbb{F}_{p^m}$ requires $O(m^2)$ bit operations or, using fast polynomial arithmetic, $O(M(m))$ bit operations. If m is fixed and p goes to infinity then the complexity is $O(M(\log(p)))$ bit operations.

Inversion of elements in $\mathbb{F}_{p^m} = \mathbb{F}_p[x]/(F(x))$ can be done using the extended Euclidean algorithm in $O(m^2)$ operations in $\mathbb{F}_p$. If p is fixed and m grows then one can invert elements in $\mathbb{F}_{p^m}$ in $O(M(m)\log(m))$ bit operations.

Alternatively, for any vector space basis $\{\theta_1, \ldots, \theta_m\}$ for $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ there is an $m \times m$ matrix M over $\mathbb{F}_q$ such that the product ab for $a, b \in \mathbb{F}_{q^m}$ is given by

$$(a_1, \ldots, a_m)M(b_1, \ldots, b_m)^t = \sum_{i=1}^{m}\sum_{j=1}^{m} M_{i,j}a_i b_j$$

where $(a_1, \ldots, a_m)$ and $(b_1, \ldots, b_m)$ are the coefficient vectors for the representation of a and b with respect to the basis.

In particular, if $\mathbb{F}_{q^m}$ is represented by a normal basis $\{\theta, \theta^q, \ldots, \theta^{q^{m-1}}\}$ then multiplication of elements in normal basis representation is given by

$$\left(\sum_{i=0}^{m-1} a_i\theta^{q^i}\right)\left(\sum_{j=0}^{m-1} b_j\theta^{q^j}\right) = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_i b_j \theta^{q^i+q^j}$$

so it is necessary to precompute the representation of each term $M_{i,j} = \theta^{q^i+q^j}$ over the normal basis. Multiplication in $\mathbb{F}_{p^m}$ using a normal basis representation is typically slower than multiplication with a polynomial basis; indeed, the complexity can be as bad as $O(m^3)$ operations in $\mathbb{F}_p$. An **optimal normal basis** is a normal basis for which the number of non-zero coefficients in the product is minimal (see Section II.2.2 of [64] for the case of $\mathbb{F}_{2^m}$). Much work has been done on speeding up multiplication with optimal normal bases; for example see Bernstein and Lange [54] for discussion and references.

Raising an element of $\mathbb{F}_{q^m}$ to the q-th power is always fast since it is a linear operation. Taking q-th powers (respectively, q-th roots) is especially fast for normal bases as it is a rotation; this is the main motivation for considering normal bases. This fact has a number of important applications, see for example, Exercise 14.4.7.

The **Itoh-Tsujii inversion algorithm** [307] is appropriate when using a normal basis representation, though it can be used to compute inverses in any finite field. First we present a simple inversion algorithm for any field and later we sketch the actual Itoh-Tsujii algorithm. Let $g \in \mathbb{F}_{q^m}^*$. The idea is to exploit the fact that the norm $\mathrm{N}_{\mathbb{F}_{q^m}/\mathbb{F}_q}(g) = g^{1+q+q^2+\cdots+q^{m-1}}$ lies in $\mathbb{F}_q$ and is therefore easier to invert than an element of $\mathbb{F}_{q^m}$.

Lemma 2.11.2. *Let $g \in \mathbb{F}_{q^m}^*$. Then*

$$g^{-1} = \mathrm{N}_{\mathbb{F}_{q^m}/\mathbb{F}_q}(g)^{-1} \prod_{i=1}^{m-1} g^{q^i}.$$

Exercise 2.11.3. Prove Lemma 2.11.2.

A simple inversion algorithm is to compute $h_1 = \prod_{i=1}^{m-1} g^{q^i}$ and $h_2 = gh$ using m multiplications in $\mathbb{F}_{q^m}$, then h_2^{-1} using one inversion in $\mathbb{F}_q$ and finally $g^{-1} = h_2^{-1}h_1$ with an $\mathbb{F}_q \times \mathbb{F}_{q^m}$ multiplication. The complexity of the algorithm is $O(m^3 \log(q)^2)$ bit operations using naive arithmetic, or $O(mM(m)\log(q)^2)$ using fast arithmetic when m is large and q is small. This is worse than the complexity $O(m^2 \log(q)^2)$ of the extended Euclidean algorithm.

In the case where $q = 2$ we know that $\mathrm{N}_{\mathbb{F}_{q^m}/\mathbb{F}_q}(g) = 1$ and the algorithm simply computes g^{-1} as

$$g^{2+2^2+\cdots+2^{m-1}} = \prod_{i=1}^{m-1} g^{2^i}.$$

This formula can be derived directly using the fact $g^{2^m-1} = 1$ as

$$g^{-1} = g^{2^m-1-1} = g^{2(2^{m-1}-1)} = g^{2(1+2+2^2+\cdots+2^{m-2})}.$$

The Itoh-Tsujii algorithm then follows from a further idea, which is that one can compute $g^{2^{m-1}-1}$ in fewer than m multiplications using an appropriate addition chain. We give the details only in the special case where $m = 2^k + 1$. Since $2^{m-1} - 1 = 2^{2^k} - 1 = (2^{2^{k-1}} - 1)2^{2^{k-1}} + (2^{2^{k-1}} - 1)$ it is sufficient to compute the sequence $g^{2^{2^i}-1}$ iteratively for $i = 0, 1, \ldots, k$, each step taking some shifts and one multiplication in the field. In other words, the complexity in this case is $O(km^2 \log(q)^2) = O(\log(m)m^2 \log(q)^2)$ field operations. For details of the general case, and further discussion we refer to [307, 270].

See, for example, Fong, Hankerson, Lopez and Menezes [207] for more discussion about inversion for the fields relevant for elliptic curve cryptography.

Finally we remark that, for some computational devices, it is convenient to use finite fields $\mathbb{F}_{p^m}$ where $p \approx 2^{32}$ or $p \approx 2^{64}$. These are called **optimal extension fields** and we refer to Section 2.4 of [274] for details.

2.12 Factoring Polynomials over Finite Fields

There is a large literature on polynomial factorisation and we only give a very brief sketch of the main concepts. The basic ideas go back to Berlekamp and others. For full discussion, historical background, and extensive references see Chapter 7 of Bach and Shallit [22] or Chapter 14 of von zur Gathen and Gerhard [238]. One should be aware that for polynomials over fields of small characteristic the algorithm by Niederreiter [466] can be useful.

Let $F(x) \in \mathbb{F}_q[x]$ have degree d. If there exists $G(x) \in \mathbb{F}_q[x]$ such that $G(x)^2 \mid F(x)$ then $G(x) \mid F'(x)$ where $F'(x)$ is the derivative of $F(x)$. A polynomial is **square-free** if it has no repeated factor. It follows that $F(x)$ is square-free if $F'(x) \neq 0$ and $\gcd(F(x), F'(x)) = 1$. If $F(x) \in \mathbb{F}_q[x]$ and $S(x) = \gcd(F(x), F'(x))$ then $F(x)/S(x)$ is square-free.

Exercise 2.12.1. Determine the complexity of testing whether a polynomial $F(x) \in \mathbb{F}_q[x]$ is square-free.

Exercise 2.12.2. Show that one can reduce polynomial factorisation over finite fields to the case of factoring square-free polynomials.

Finding Roots of Polynomials in Finite Fields

Let $F(x) \in \mathbb{F}_q[x]$ have degree d. The roots of $F(x)$ in $\mathbb{F}_q$ are precisely the roots of

$$R_1(x) = \gcd(F(x), x^q - x). \tag{2.3}$$

If q is much larger than d then the efficient way to compute $R_1(x)$ is to compute $x^q \pmod{F(x)}$ using a square-and-multiply algorithm and then run Euclid's algorithm.

Exercise 2.12.3. Determine the complexity of computing $R_1(x)$ in equation (2.3). Hence explain why the decision problem "Does $F(x)$ have a root in $\mathbb{F}_q$?" has a polynomial-time solution.

The basic idea of root-finding algorithms is to note that, if q is odd, $x^q - x = x(x^{(q-1)/2} + 1)(x^{(q-1)/2} - 1)$. Hence, one can try to split[9] $R_1(x)$ by computing

$$\gcd(R_1(x), x^{(q-1)/2} - 1). \tag{2.4}$$

Similar ideas can be used when q is even (see Section 2.14.2).

Exercise 2.12.4. Show that the roots of the polynomial in equation (2.4) are precisely the $\alpha \in \mathbb{F}_q$ such that $F(\alpha) = 0$ and α is a square in $\mathbb{F}_q^*$.

To obtain a randomised (Las Vegas) algorithm to factor $R_1(x)$ completely when q is odd one simply chooses a random polynomial $u(x) \in \mathbb{F}_q[x]$ of degree $< d$ and computes

$$\gcd(R_1(x), u(x)^{(q-1)/2} - 1).$$

This computation selects those roots α of $R_1(x)$ such that $u(\alpha)$ is a square in $\mathbb{F}_q$. In practice it suffices to choose $u(x)$ to be linear. Performing this computation sufficiently many times on the resulting factors of $R_1(x)$ and taking gcds eventually leads to the complete factorisation of $R_1(x)$.

[9] We reserve the word "factor" for giving the full decomposition into irreducibles, whereas we use the word "split" to mean breaking into two pieces.

Exercise 2.12.5. Write down pseudocode for the above root finding algorithm and show that its expected complexity (without using a fast Euclidean algorithm) is bounded by $O(\log(d)(\log(q)M(d) + d^2)) = O(\log(q)\log(d)d^2)$ field operations.

Exercise 2.12.6. Let q be an odd prime power and $R(x) = x^2 + ax + b \in \mathbb{F}_q[x]$. Show that the expected complexity of finding roots of $R(x)$ using polynomial factorisation is $O(\log(q)M(\log(q)))$ bit operations.

Exercise 2.12.7.★ Show, using Kronecker substitution, fast versions of Euclid's algorithm and other tricks, that one can compute one root in $\mathbb{F}_q$ (if any exist) of a degree d polynomial in $\mathbb{F}_q[x]$ in an expected $O(\log(qd)M(d, q))$ bit operations.

When q is even (i.e., $q = 2^m$) then, instead of $x^{(q-1)/2}$, one considers the **trace polynomial** $T(x) = \sum_{i=0}^{m-1} x^{2^i}$. (A similar idea can be used over any field of small characteristic.)

Exercise 2.12.8. Show that the roots of the polynomial $\gcd(R_1(x), T(x))$ are precisely the $\alpha \in \mathbb{F}_q$ such that $R_1(\alpha) = 0$ and $\mathrm{Tr}_{\mathbb{F}_{2^m}/\mathbb{F}_2}(\alpha) = 0$.

Taking random $u(x) \in \mathbb{F}_{2^m}[x]$ of degree $< d$ and then computing $\gcd(R_1(x), T(u(x)))$ gives a Las Vegas root finding algorithm as before. See Section 21.3.2 of [556] for details.

Higher Degree Factors

Having found the roots in $\mathbb{F}_q$ one can try to find factors of larger degree. The same ideas can be used. Let

$$R_2(x) = \gcd(F(x)/R_1(x), x^{q^2} - x),\ R_3(x) = \gcd(F(x)/(R_1(x)R_2(x)), x^{q^3} - x),\ \ldots.$$

Exercise 2.12.9. Show that all irreducible factors of $R_i(x)$ over $\mathbb{F}_q[x]$ have degree i.

Exercise 2.12.10. Give an algorithm to test whether a polynomial $F(x) \in \mathbb{F}_q[x]$ of degree d is irreducible. What is the complexity?

When q is odd one can factor $R_i(x)$ using similar ideas to the above, i.e., by computing

$$\gcd(R_i(x), u(x)^{(q^i-1)/2} - 1).$$

These techniques lead to the **Cantor-Zassenhaus algorithm**. It factors polynomials of degree d over $\mathbb{F}_q$ in an expected $O(d\log(d)\log(q)M(d))$ field operations. For many more details about polynomial factorisation see Section 7.4 of [22], Sections 21.3 and 21.4 of [556], Chapter 14 of [238], [370], Chapter 4 of [388, 389] or Section 4.6.2 of [343].

Exercise 2.12.11. Let $d \in \mathbb{N}$ and $F(x) \in \mathbb{F}_q[x]$ of degree d. Given $1 < b < n$ suppose we wish to output all irreducible factors of $F(x)$ of degree at most b. Show that the expected complexity is $O(b\log(d)\log(q)M(d))$ operations in $\mathbb{F}_q$. Hence, one can factor $F(x)$ completely in $O(d\log(d)\log(q)M(d))$ operations in $\mathbb{F}_q$.

Exercise 2.12.12.★ Using the same methods as Exercise 2.12.7, show that one can find an irreducible factor of degree $1 < b < d$ of a degree d polynomial in $\mathbb{F}_q[x]$ in an expected $O(b\log(dq)M(d, q))$ bit operations.

2.13 Hensel Lifting

Hensel lifting is a tool for solving polynomial equations of the form $F(x) \equiv 0 \pmod{p^e}$ where p is prime and $e \in \mathbb{N}_{>1}$. One application of Hensel lifting is the Takagi variant of RSA, see Example 24.1.6. The key idea is given in the following Lemma.

Lemma 2.13.1. *Let $F(x) \in \mathbb{Z}[x]$ be a polynomial and p a prime. Let $x_k \in \mathbb{Z}$ satisfy $F(x_k) \equiv 0 \pmod{p^k}$ where $k \in \mathbb{N}$. Suppose $F'(x_k) \not\equiv 0 \pmod{p}$. Then one can compute $x_{k+1} \in \mathbb{Z}$ in polynomial-time such that $F(x_{k+1}) \equiv 0 \pmod{p^{k+1}}$.*

Proof: Write $x_{k+1} = x_k + p^k z$ where z is a variable. Note that $F(x_{k+1}) \equiv 0 \pmod{p^k}$. One has

$$F(x_{k+1}) \equiv F(x_k) + p^k F'(x_k) z \pmod{p^{k+1}}.$$

Setting $z = -(F(x_k)/p^k)F'(x_k)^{-1} \pmod{p}$ gives $F(x_{k+1}) \equiv 0 \pmod{p^{k+1}}$. □

Example 2.13.2. We solve the equation

$$x^2 \equiv 7 \pmod{3^3}.$$

Let $f(x) = x^2 - 7$. First, the equation $f(x) \equiv x^2 - 1 \pmod{3}$ has solutions $x \equiv 1, 2 \pmod{3}$. We take $x_1 = 1$. Since $f'(1) = 2 \not\equiv 0 \pmod{3}$ we can "lift" this to a solution modulo 3^2. Write $x_2 = 1 + 3z$. Then

$$f(x_2) = x_2^2 - 7 \equiv -6 + 6z \pmod{3^2}$$

or, in other words, $1 - z \equiv 0 \pmod{3}$. This has the solution $z = 1$ giving $x_2 = 4$.

Now lift to a solution modulo 3^3. Write $x_3 = 4 + 9z$. Then $f(x_3) \equiv 9 + 72z \pmod{3^3}$ and dividing by 9 yields $1 - z \equiv 0 \pmod{3}$. This has solution $z = 1$ giving $x_3 = 13$ as one solution to the original equation.

Exercise 2.13.3. The equation $x^3 \equiv 3 \pmod{5}$ has the solution $x \equiv 2 \pmod{5}$. Use Hensel lifting to find a solution to the equation $x^3 \equiv 3 \pmod{5^3}$.

Exercise 2.13.4. Let $F(x) \in \mathbb{Z}[x]$ be a polynomial and p a prime. Let $x_k \in \mathbb{Z}$ satisfy $F(x_k) \equiv 0 \pmod{p^k}$ and $F'(x_k) \not\equiv 0 \pmod{p}$. Show that the Hensel iteration can be written in the form

$$x_{k+1} = x_k - \frac{F(x_k)}{F'(x_k)}$$

just like Newton iteration. Show that Hensel lifting has quadratic convergence in this case (i.e., if $F(x_k) \equiv 0 \pmod{p^k}$ then $F(x_{k+1}) \equiv 0 \pmod{p^{2k}}$).

2.14 Algorithms in Finite Fields

We present some algorithms for constructing finite fields $\mathbb{F}_{p^m}$ when $m > 1$, solving equations in them, and transforming between different representations of them.

2.14.1 Constructing Finite Fields

Lemma A.5.1 implies a randomly chosen monic polynomial in $\mathbb{F}_q[x]$ of degree m is irreducible with probability $\geq 1/(2m)$. Hence, using the algorithm of Exercise 2.12.10 one can generate a random irreducible polynomial $F(x) \in \mathbb{F}_q[x]$ of degree m, using naive arithmetic, in $O(m^4 \log(q))$ operations in $\mathbb{F}_q$. In other words, one can construct a polynomial basis for $\mathbb{F}_{q^m}$ in $O(m^4 \log(q))$ operations in $\mathbb{F}_q$. This complexity is not the best known.

Constructing a Normal Basis

We briefly survey the literature on constructing normal bases for finite fields. We assume that a polynomial basis for $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ has already been computed.

The simplest randomised algorithm is to choose $\theta \in \mathbb{F}_{q^m}$ at random and test whether the set $\{\theta, \theta^q, \ldots, \theta^{q^{m-1}}\}$ is linearly independent over $\mathbb{F}_q$. Corollary 3.6 of von zur Gathen and Giesbrecht [239] (also see Theorem 3.73 and Exercise 3.76 of [388, 389]) shows that a randomly chosen θ is normal with probability at least $1/34$ if $m < q^4$ and probability at least $1/(16\log_q(m))$ if $m \geq q^4$.

Exercise 2.14.1. Determine the complexity of constructing a normal basis by randomly choosing θ.

When $q > m(m-1)$ there is a better randomised algorithm based on the following result.

Theorem 2.14.2. *Let $F(x) \in \mathbb{F}_q[x]$ be irreducible of degree m and let $\alpha \in \mathbb{F}_{q^m}$ be any root of $F(x)$. Define $G(x) = F(x)/((x-\alpha)F'(\alpha)) \in \mathbb{F}_{q^m}[x]$. Then there are $q - m(m-1)$ elements $u \in \mathbb{F}_q$ such that $\theta = G(u)$ generates a normal basis.*

Proof: See Theorem 28 of Section II.N of Artin [14] or Section 3.1 of Gao [236]. □

Deterministic algorithms for constructing a normal basis have been given by Lüneburg [399] and Lenstra [379] (also see Gao [236]).

2.14.2 Solving Quadratic Equations in Finite Fields

This section is about solving quadratic equations $x^2 + ax + b = 0$ over $\mathbb{F}_q$. One can apply any of the algorithms for polynomial factorisation mentioned earlier. As we saw in Exercise 2.12.6, when q is odd, the basic method computes roots in $O(\log(q)M(\log(q)))$ bit operations. When q is odd it is also natural to use the quadratic formula and a square-roots algorithm (see Section 2.9).

Exercise 2.14.3. Generalise the Tonelli-Shanks algorithm from Section 2.9 to work for any finite field $\mathbb{F}_q$ where q is odd. Show that the complexity remains an expected $O(\log(q)^2 M(\log(q)))$ bit operations.

Exercise 2.14.4. Suppose $\mathbb{F}_{q^2}$ is represented as $\mathbb{F}_q(\theta)$ where $\theta^2 \in \mathbb{F}_q$. Show that one can compute square roots in $\mathbb{F}_{q^2}$ using two square roots in $\mathbb{F}_q$ and a small number of multiplications in $\mathbb{F}_q$.

Since squaring in $\mathbb{F}_{2^m}$ is a linear operation one can take square roots in $\mathbb{F}_{2^m}$ using linear algebra in $O(m^3)$ bit operations. The following exercise gives a method that is more efficient.

Exercise 2.14.5. Suppose one represents $\mathbb{F}_{2^m}$ using a polynomial basis $\mathbb{F}_2[x]/(F(x))$. Precompute $\sqrt{x}$ as a polynomial in x. Let $g = \sum_{i=0}^{m-1} a_i x^i$. To compute $\sqrt{g}$ write (assuming m is odd; the case of m even is similar)

$$g = \left(a_0 + a_2x^2 + \cdots + a_{m-1}x^{m-1}\right) + x\left(a_1 + a_3x^2 + \cdots + a_{m-2}x^{m-3}\right).$$

Show that

$$\sqrt{g} = \left(a_0 + a_2x + \cdots x_{m-1}x^{(m-1)/2}\right) + \sqrt{x}\left(a_1 + a_3x + \cdots + a_{m-2}x^{(m-3)/2}\right).$$

Show that this computation takes roughly half the cost of one field multiplication, and hence $O(m^2)$ bit operations.

Exercise 2.14.6. Generalise Exercise 2.14.5 to computing p-th roots in $\mathbb{F}_{p^m}$. Show that the method requires $(p-1)$ multiplications in $\mathbb{F}_{p^m}$.

We now consider how to solve quadratic equations of the form

$$x^2 + x = \alpha \tag{2.5}$$

where $\alpha \in \mathbb{F}_{2^m}$.

Exercise 2.14.7.★ Prove that the equation $x^2 + x = \alpha$ has a solution $x \in \mathbb{F}_{2^m}$ if and only if $\mathrm{Tr}_{\mathbb{F}_{2^m}/\mathbb{F}_2}(\alpha) = 0$.

Lemma 2.14.8. *If m is odd (we refer to Section II.2.4 of [64] for the case where m is even) then a solution to equation (2.5) is given by the* ***half trace***

$$x = \sum_{i=0}^{(m-1)/2} \alpha^{2^{2i}}. \tag{2.6}$$

Exercise 2.14.9. Prove Lemma 2.14.8. Show that the complexity of solving quadratic equations in $\mathbb{F}_q$ when $q = 2^m$ and m is odd is an expected $O(m^3)$ bit operations (or $O(m^2)$ bit operations when a normal basis is being used).

The expected complexity of solving quadratic equations in $\mathbb{F}_{2^m}$ when m is even is $O(m^4)$ bit operations, or $O(m^3)$ when a normal basis is being used. Hence, we can make the statement that the complexity of solving a quadratic equation over any field $\mathbb{F}_q$ is an expected $O(\log(q)^4)$ bit operations.

2.14.3 Isomorphisms Between Finite Fields

Computing the Minimal Polynomial of an Element

Given $g \in \mathbb{F}_{q^m}$ one can compute the minimal polynomial $F(x)$ of g over $\mathbb{F}_q$ using linear algebra. To do this one considers the set $S_n = \{1, g, g^2, \ldots, g^n\}$ for $n = 1, \ldots, m$. Let n be the smallest integer such that S_n is linearly dependent over $\mathbb{F}_q$. Then there are $a_0, \ldots, a_n \in \mathbb{F}_q$ such that $\sum_{i=0}^{n} a_i g^i = 0$. Since S_{n-1} is linearly independent it follows that $F(x) = \sum_{i=0}^{n} a_i x^i$ is the minimal polynomial for g.

Exercise 2.14.10. Show that the above algorithm requires $O(m^3)$ operations in $\mathbb{F}_q$.

Computing a Polynomial Basis for a Finite Field

Suppose $\mathbb{F}_{q^m}$ is given by some basis that is not a polynomial basis. We now give a method to compute a polynomial basis for $\mathbb{F}_{q^m}$.

If $g \in \mathbb{F}_{q^m}$ is chosen uniformly at random then, by Lemma A.8.4, with probability at least $1/q$ the element g does not lie in a subfield of $\mathbb{F}_{q^m}$ that contains $\mathbb{F}_q$. Hence the minimal polynomal $F(x)$ of g over $\mathbb{F}_q$ has degree m and the algorithm of the previous subsection computes $F(x)$. One therefore has a polynomial basis $\{1, x, \ldots, x^{m-1}\}$ for $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$.

Exercise 2.14.11. Determine the complexity of this algorithm.

Computing Isomorphisms Between Finite Fields

Suppose one has two representations for $\mathbb{F}_{q^m}$ as a vector space over $\mathbb{F}_q$ and wants to compute an isomorphism between them. We do this in two stages: first we compute an isomorphism from any representation to a polynomial basis, and second we compute isomorphisms between any two polynomial bases. We assume that one already has an isomorphism between the corresponding representations of the subfield $\mathbb{F}_q$.

Let $\{\theta_1, \ldots, \theta_m\}$ be the vector space basis over $\mathbb{F}_q$ for one of the representations of $\mathbb{F}_{q^m}$. The first task is to compute an isomorphism from this representation to a polynomial representation. To do this one computes a polynomial basis for $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ using the method above. One now has a monic irreducible polynomial $F(x) \in \mathbb{F}_q[x]$ of degree m and a representation $x = \sum_{i=1}^m a_i\theta_i$ for a root of $F(x)$ in $\mathbb{F}_{q^m}$. Determine the representations of $x^2, x^3, \ldots, x^m$ over the basis $\{\theta_1, \ldots, \theta_m\}$. This gives an isomorphism from $\mathbb{F}_q[x]/(F(x))$ to the original representation of $\mathbb{F}_{q^m}$. By solving a system of linear equations, one can express each of $\theta_1, \ldots, \theta_m$ with respect to the polynomial basis; this gives the isomorphism from the original representation to $\mathbb{F}_q[x]/(F(x))$. The above ideas appear in a special case in the work of Zierler [641].

Exercise 2.14.12. Determine the complexity of the above algorithm to give an isomorphism between an arbitrary vector space representation of $\mathbb{F}_{q^m}$ and a polynomial basis for $\mathbb{F}_{q^m}$.

Finally, it remains to compute an isomorphism between any two polynomial representations $\mathbb{F}_q[x]/(F_1(x))$ and $\mathbb{F}_q[y]/(F_2(y))$ for $\mathbb{F}_{q^m}$. This is done by finding a root $a(y) \in \mathbb{F}_q[y]/(F_2(y))$ of the polynomial $F_1(x)$. The function $x \mapsto a(y)$ extends to a field isomorphism from $\mathbb{F}_q[x]/(F_1(x))$ to $\mathbb{F}_q[y]/(F_2(y))$. The inverse to this isomorphism is computed by linear algebra.

Exercise 2.14.13. Determine the complexity of the above algorithm to give an isomorphism between an arbitrary vector space representation of $\mathbb{F}_{q^m}$ and a polynomial basis for $\mathbb{F}_{q^m}$.

See Lenstra [379] for deterministic algorithms to solve this problem.

Random Sampling of Finite Fields

Let $\mathbb{F}_{p^m}$ be represented as a vector space over $\mathbb{F}_p$ with basis $\{\theta_1, \ldots, \theta_m\}$. Generating an element $g \in \mathbb{F}_{p^m}$ uniformly at random can be done by selecting m integers $a_1, \ldots, a_m$ uniformly at random in the range $0 \leq a_i < p$ and taking $g = \sum_{i=1}^m a_i\theta_i$. Section 11.4 mentions some methods to get random integers modulo p from random bits.

To sample uniformly from $\mathbb{F}_{p^m}^*$ one can use the above method, repeating the process if $a_i = 0$ for all $1 \leq i \leq m$. This is much more efficient than choosing $0 \leq a < p^m - 1$ uniformly at random and computing $g = \gamma^a$ where γ is a primitive root.

2.15 Computing Orders of Elements and Primitive Roots

We first consider how to determine the order of an element $g \in \mathbb{F}_q^*$. Assume the factorisation $q - 1 = \prod_{i=1}^m l_i^{e_i}$ is known.[10] Then it is sufficient to determine, for each i, the

[10] As far as I am aware, it has not been proved that computing the order of an element in $\mathbb{F}_q^*$ is equivalent to factoring $q-1$; or even that computing the order of an element in $\mathbb{Z}_N^*$ is equivalent to factoring $\varphi(N)$. Yet it seems to be impossible to correctly determine the order of every $g \in \mathbb{F}_q^*$ without knowing the factorisation of $q - 1$.

smallest $0 \leq f \leq e_i$ such that

$$g^{(q-1)/l_i^f} = 1.$$

This leads to a simple algorithm for computing the order of g that requires $O(\log(q)^4)$ bit operations.

Exercise 2.15.1. Write pseudocode for the basic algorithm for determining the order of g and determine the complexity.

The next subsection gives an algorithm (also used in other parts of the book) that leads to an improvement of the basic algorithm.

2.15.1 Sets of Exponentials of Products

We now explain how to compute sets of the form $\{g^{(q-1)/l} : l \mid (q-1)\}$ efficiently. We generalise the problem as follows. Let $N_1, \ldots, N_m \in \mathbb{N}$ and $N = \prod_{i=1}^m N_i$ (typically the integers N_i will be coprime, but it is not necessary to assume this). Let $k = \lceil \log_2(m) \rceil$ and, for $m < i \leq 2^k$ set $N_i = 1$. Let G be a group and $g \in G$ (where g typically has order $\geq N$). The goal is to efficiently compute

$$\{g^{N/N_i} : 1 \leq i \leq m\}.$$

The naive approach (computing each term separately and not using any window methods etc) requires at least

$$\sum_{i=1}^{m} \log(N/N_i) = m\log(N) - \sum_{i=1}^{m} \log(N_i) = (m-1)\log(N)$$

operations in G and at most $2m\log(N)$ operations in G.

For the improved solution one re-uses intermediate values. The basic idea can be seen in the following example. Computing products in such a way is often called using a **product tree**.

Example 2.15.2. Let $N = N_1N_2N_3N_4$ and suppose one needs to compute

$$g^{N_1N_2N_3}, g^{N_1N_2N_4}, g^{N_1N_3N_4}, g^{N_2N_3N_4}.$$

We first compute

$$h_{1,1} = g^{N_3N_4} \qquad \text{and} \qquad h_{1,2} = g^{N_1N_2}$$

in $\leq 2(\log_2(N_1N_2) + \log_2(N_3N_4)) = 2\log_2(N)$ operations. One can then compute the result

$$g^{N_1N_2N_3} = h_{1,2}^{N_3},\ g^{N_1N_2N_4} = h_{1,2}^{N_4},\ g^{N_1N_3N_4} = h_{1,1}^{N_1},\ g^{N_2N_3N_4} = h_{1,1}^{N_2}.$$

This final step requires at most $2(\log_2(N_3)+\log_2(N_4)+\log_2(N_1)+\log_2(N_2)) = 2\log_2(N)$ operations. The total complexity is at most $4\log_2(N)$ operations in the group.

The algorithm has a compact recursive description. Let F be the function that on input $(g, m, N_1, \ldots, N_m)$ outputs the list of m values g^{N/N_i} for $1 \leq i \leq m$ where $N = N_1 \cdots N_m$. Then $F(g, 1, N_1) = g$. For $m > 1$ one computes $F(g, m, N_1, \ldots, N_m)$ as follows: Let $l = \lfloor m/2 \rfloor$ and let $h_1 = g^{N_1 \cdots N_l}$ and $h_2 = g^{N_{l+1} \cdots N_m}$. Then $F(g, m, N_1, \ldots, N_m)$ is equal to the concatenation of $F(h_1, (m-l), N_{l+1}, \ldots, N_m)$ and $F(h_2, l, N_1, \ldots, N_l)$.

We introduce some notation to express the algorithm in a non-recursive format.

Definition 2.15.3. Define $S = \{1, 2, 3, \ldots, 2^k\}$. For $1 \leq l \leq k$ and $1 \leq j \leq 2^l$ define

$$S_{l,j} = \{i \in S : (j-1)2^{k-l} + 1 \leq i \leq j2^{k-l}\}$$

Lemma 2.15.4. *Let $1 \leq l \leq k$ and $1 \leq j \leq 2^l$. The sets $S_{l,j}$ satisfy:*

1. $\#S_{l,j} = 2^{k-l}$;
2. $S_{l,j} \cap S_{l,j'} = \varnothing$ *if* $j \neq j'$;
3. $\cup_{j=1}^{2^l} S_{l,j} = S$ *for all* $1 \leq l \leq k$;
4. *If* $l \geq 2$ *and* $1 \leq j \leq 2^{l-1}$ *then* $S_{l-1,j} = S_{l,2j-1} \cup S_{l,2j}$;
5. $S_{k,j} = \{j\}$ *for* $1 \leq j \leq 2^k$.

Exercise 2.15.5. Prove Lemma 2.15.4.

Definition 2.15.6. For $1 \leq l \leq k$ and $1 \leq j \leq 2^l$ define

$$h_{l,j} = g^{\prod_{j \in S - S_{l,j}} N_i}.$$

To compute $\{h_{k,j} : 1 \leq j \leq m\}$ efficiently one notes that if $l \geq 2$ and $1 \leq j \leq 2^l$ then, writing $j_1 = \lceil j/2 \rceil$,

$$h_{l,j} = h_{l-1,j_1}^{\prod_{i \in S_{l-1,j_1} - S_{l,j}} N_i}.$$

This leads to Algorithm 4.

Algorithm 4 Computing Set of Exponentials of Products

INPUT: $N_1, \ldots, N_m$
OUTPUT: $\{g^{N/N_i} : 1 \leq i \leq m\}$
1: $k = \lceil \log_2(m) \rceil$
2: $h_{1,1} = g^{N_{2^{k-1}+1} \cdots N_{2^k}}$, $h_{1,2} = g^{N_1 \cdots N_{2^{k-1}}}$
3: **for** $l = 2$ to k **do**
4: **for** $j = 1$ to 2^l **do**
5: $j_1 = \lceil j/2 \rceil$
6: $h_{l,j} = h_{l-1,j_1}^{\prod_{i \in S_{l-1,j_1} - S_{l,j}} N_i}$
7: **end for**
8: **end for**
9: **return** $\{h_{k,1}, \ldots, h_{k,m}\}$

Lemma 2.15.7. *Algorithm 4 is correct and requires $\leq 2\lceil \log_2(m) \rceil \log(N)$ group operations.*

Proof: Almost everything is left as an exercise. The important observation is that lines 4 to 7 involve raising to the power N_i for all $i \in S$. Hence the cost for each iteration of the loop in line 3 is at most $2 \sum_{i=1}^{2^k} \log_2(N_i) = 2 \log_2(N)$. □

This method works efficiently in all cases (i.e., it doesn't require m to be large). However, Exercise 2.15.8 shows that for small values of m there may be more efficient solutions.

Exercise 2.15.8. Let $N = N_1 N_2 N_3$ where $N_i \approx N^{1/3}$ for $1 \leq i \leq 3$. One can compute g^{N/N_i} for $1 \leq i \leq 3$ using Algorithm 4 or in the "naive" way. Suppose one uses the

standard square-and-multiply method for exponentiation and assume that each of N_1, N_2 and N_3 has Hamming weight about half their bit-length.

Note that the exponentiations in the naive solution are all with respect to the fixed base g. A simple optimisation is therefore to precompute all g^{2^j} for $1 \leq j \leq \log_2(N^{2/3})$. Determine the number of group operations for each algorithm if this optimisation is performed. Which is better?

Remark 2.15.9. Sutherland gives an improved algorithm (which he calls the **snowball algorithm**) as Algorithm 7.4 of [596]. Proposition 7.3 of [596] states that the complexity is

$$O(\log(N)\log(m)/\log(\log(m))) \tag{2.7}$$

group operations.

2.15.2 Computing the Order of a Group Element

We can now return to the original problem of computing the order of an element in a finite field.

Theorem 2.15.10. *Let $g \in \mathbb{F}_q^*$ and assume that the factorisation $q-1 = \prod_{i=1}^m l_i^{e_i}$ is known. Then one can determine the order of g in $O(\log(q)\log\log(q))$ multiplications in $\mathbb{F}_q$.*

Proof: The idea is to use Algorithm 4 to compute all $h_i = g^{(q-1)/l_i^{e_i}}$. Since $m = O(\log(q))$ this requires $O(\log(q)\log\log(q))$ multiplications in $\mathbb{F}_q$. One can then compute all $h_i^{l_i^f}$ for $1 \leq f < e_i$ and, since $\prod_{i=1}^m l_i^{e_i} = q-1$ this requires a further $O(\log(q))$ multiplications. □

The complexity in Theorem 2.15.10 cannot be improved to $O(\log(q)\log\log(q)/\log(\log(\log(q))))$ using the result of equation (2.7) because the value m is not always $\Theta(\log(q))$.

2.15.3 Computing Primitive Roots

Recall that $\mathbb{F}_q^*$ is a cyclic group and that a primitive root in $\mathbb{F}_q^*$ is an element of order $q-1$. We assume in this section that the factorisation of $q-1$ is known.

One algorithm to generate primitive roots is to choose $g \in \mathbb{F}_q^*$ uniformly at random and to compute the order of g using the method of Theorem 2.15.10 until an element of order $q-1$ is found. The probability that a random $g \in \mathbb{F}_q^*$ is a primitive root is $\varphi(q-1)/(q-1)$. Using Theorem A.3.1 this probability is at least $1/(6\log(\log(q)))$. Hence this gives an algorithm that requires $O(\log(q)(\log(\log(q)))^2)$ field multiplications in $\mathbb{F}_q$.

We now present a better algorithm for this problem, which works by considering the prime powers dividing $q-1$ individually. See Exercise 11.2 of Section 11.1 of [556] for further details.

Theorem 2.15.11. *Algorithm 5 outputs a primitive root. The complexity of the algorithm is $O(\log(q)\log\log(q))$ multiplications in $\mathbb{F}_q$.*

Proof: The values g_i are elements of order dividing $l_i^{e_i}$. If $g_i^{l_i^{e_i-1}} \neq 1$ then g_i has order exactly $l_i^{e_i}$. One completion of the while loop the value t is the product of m elements of maximal coprime orders $l_i^{e_i}$. Hence t is a primitive root.

Each iteration of the while loop requires $O(\log(q)\log\log(q))$ multiplications in $\mathbb{F}_q$. It remains to bound the number of iterations of the loop. First note that, by the Chinese remainder theorem, the g_i are independent and uniformly at random in subgroups of

Algorithm 5 Computing a primitive root in $\mathbb{F}_q^*$

INPUT: $q, m, \{(l_i, e_i)\}$ such that $q-1 = \prod_{i=1}^{m} l_i^{e_i}$ and the l_i are distinct primes
OUTPUT: primitive root g

```
1: Let S = {1, ..., m}
2: t = 1
3: while S ≠ ∅ do
4:     Choose g ∈ F_q^* uniformly at random
5:     Compute g_i = g^{(q-1)/l_i^{e_i}} for 1 ≤ i ≤ m using Algorithm 4
6:     for i ∈ S do
7:         if g_i^{l_i^{e_i-1}} ≠ 1 then
8:             t = t g_i
9:             Remove i from S
10:        end if
11:    end for
12: end while
13: return t
```

order $l_i^{e_i}$. Hence, the probability that $g_i^{l_i^{e_i-1}} = 1$ is $1/l_i \leq 1/2$ and the expected number of trials for any given value g_i less than or equal to 2. Hence, the expected number of iterations of the while loop is less than or equal to 2. This completes the proof. □

2.16 Fast Evaluation of Polynomials at Multiple Points

We have seen that one can evaluate a univariate polynomial at a field element efficiently using Horner's rule. For some applications, for example the attack on small CRT exponents for RSA in Section 24.5.2, one must evaluate a fixed polynomial repeatedly at lots of field elements. Naively repeating Horner's rule n times would give a total cost of n^2 multiplications. This section shows one can solve this problem more efficiently than the naive method.

Theorem 2.16.1. *Let $F(x) \in \Bbbk[x]$ have degree n and let $x_1, \ldots, x_n \in \Bbbk$. Then one can compute $\{F(x_1), \ldots, F(x_n)\}$ in $O(M(n)\log(n))$ field operations. The storage requirement is $O(n\log(n))$ elements of $\Bbbk$.*

Proof: (Sketch) Let $t = \lceil \log_2(n) \rceil$ and set $x_i = 0$ for $n < i \leq 2^t$. For $0 \leq i \leq t$ and $1 \leq j \leq 2^{t-i}$ define

$$G_{i,j}(x) = \prod_{k=(j-1)2^i+1}^{j2^i} (x - x_k).$$

One computes the $G_{i,j}(x)$ for $i = 0, 1, \ldots, t$ using the formula $G_{i,j}(x) = G_{i-1,2j-1}(x)G_{i-1,2j}(x)$. (This is essentially the same trick as Section 2.15.1.) For each i one needs to store n elements of $\Bbbk$ to represent all the polynomials $G_{i,j}(x)$. Hence, the total storage is $n\log(n)$ elements of $\Bbbk$.

Once all the $G_{i,j}(x)$ have been computed one defines, for $0 \leq i \leq t$, $1 \leq j \leq 2^{t-i}$ the polynomials $F_{i,j}(x) = F(x) \pmod{G_{i,j}(x)}$. One computes $F_{t,0}(x) = F(x) \pmod{G_{t,0}(x)}$ and then computes $F_{i,j}(x)$ efficiently as $F_{i+1,\lfloor (j+1)/2 \rfloor}(x) \pmod{G_{i,j}(x)}$ for $i = t-1$ downto 0. Note that $F_{0,j}(x) = F(x) \pmod{(x - x_j)} = F(x_j)$ as required.

One can show that the complexity is $O(M(n)\log(n))$ operations in $\Bbbk$. For details see Theorem 4 of [611], Section 10.1 of [238] or Corollary 4.5.4 of [88]. □

Exercise 2.16.2. Show that Theorem 2.16.1 also holds when the field $\Bbbk$ is replaced by a ring.

The inverse problem (namely, determining $F(x)$ from the n pairs $(x_j, F(x_j))$) can also be solved in $O(M(n)\log(n))$ field operations; see Section 10.2 of [238].

2.17 Pseudorandom Generation

Many of the above algorithms, and also many cryptographic systems, require generation of random or pseudorandom numbers. The precise definitions for random and pseudorandom are out of the scope of this book, as is a full discussion of methods to extract almost perfect randomness from the environment and methods to generate pseudorandom sequences from a short random seed.

There are pseudorandom number generators related to RSA (the Blum-Blum-Shub generator) and discrete logarithms. Readers interest to learn more about this topic should consult Chapter 5 of [418], Chapter 3 of [343], Chapter 30 of [16], or [398].

2.18 Summary

Table 2.18 gives a brief summary of the complexities for the algorithms discussed in this chapter. The notation used in the table is $n \in \mathbb{N}$, $a, b \in \mathbb{Z}$, p is a prime, q is a prime power and $\Bbbk$ is a field. Recall that $M(m)$ is the number of bit operations to multiply two m-bit integers (which is also the number of operations in $\Bbbk$ to multiply two degree-m polynomials over a field $\Bbbk$). Similarly, $M(d, q)$ is the number of bit operations to multiply two degree-d polynomials in $\mathbb{F}_q[x]$.

Table 2.18 gives the asymptotic complexity for the algorithms that are used in cryptographic applications (i.e., for integers of, say, at most 10,000 bits). Many of the algorithms are randomised and so the complexity in those cases is the expected complexity. The reader is warned that the best possible asymptotic complexity may be different: sometimes it is sufficient to replace $M(m)$ by $m\log(m)\log(\log(m))$ to get the best complexity, but in other cases (such as constructing a polynomial basis for $\mathbb{F}_{q^m}$) there are totally different methods that have better asymptotic complexity. In cryptographic applications $M(m)$ typically behaves as $M(m) = O(m^2)$ or $M(m) = O(m^{1.585})$.

The words "$\Bbbk$-operations" includes additions, multiplications and inversions in $\Bbbk$. If inversions in $\Bbbk$ are not required in the algorithm then we say "$\Bbbk$ multiplications".

Table 2.1: Expected complexity of basic algorithms for numbers of size relevant for cryptography and related applications. The symbol $*$ indicates that better asymptotic complexities are known.

Computational problem	Expected complexity for cryptography
Multiplication of m-bit integers, $M(m)$	$O(m^2)$ or $O(m^{1.585})$ bit operations
Compute $\lfloor a/n \rfloor, a \pmod n$	$O((\log(\lvert a\rvert) - \log(n))\log(n))$ or $O(M(\log(\lvert a\rvert))$ bit operations
Compute $\lfloor \sqrt{\lvert a\rvert} \rfloor$	$O(M(\log(\lvert a\rvert)))$ bit operations
Extended $\gcd(a, b)$ where a and b are m-bit integers	$O(m^2)$ or $O(M(m)\log(m))$ bit operations
Legendre/Jacobi symbol $(\frac{a}{n})$, $\lvert a\rvert < n$	$O(\log(n)^2)$ or $O(M(\log(n))\log(\log(n)))$ bit operations
Multiplication modulo n	$O(M(\log(n)))$ bit operations
Inversion modulo n	$O(\log(n)^2)$ or $O(M(\log(n))\log(n))$ bit operations
Compute $g^m \pmod n$	$O(\log(m)M(\log(n)))$ bit operations
Compute square roots in $\mathbb{F}_q^*$ (q odd)	$O(\log(q)M(\log(q)))$ bit operations
Multiplication of two degree d polys in $\Bbbk[x]$	$O(M(d))$ $\Bbbk$-multiplications
Multiplication of two degree d polys in $\mathbb{F}_q[x]$, $M(d, q)$	$O(M(d\log(dq)))$ bit operations
Inversion in $\Bbbk[x]/(F(x))$ where $\deg(F(x)) = d$	$O(d^2)$ or $O(M(d)\log(d))$ $\Bbbk$-operations
Multiplication in $\mathbb{F}_{q^m}$	$O(M(m))$ operations in $\mathbb{F}_q$ $*$
Evaluate degree d polynomial at $\alpha \in \Bbbk$	$O(d)$ $\Bbbk$-operations
Find all roots in $\mathbb{F}_q$ of a degree d polynomial in $\mathbb{F}_q[x]$	$O(\log(d)\log(q)d^2)$ $\mathbb{F}_q$-operations $*$
Find one root in $\mathbb{F}_q$ of a degree d polynomial in $\mathbb{F}_q[x]$	$O(\log(dq)M(d, q))$ bit operations
Determine if degree d poly over $\mathbb{F}_q$ is irreducible	$O(d^3\log(q))$ $\mathbb{F}_q$-operations $*$
Factor degree d polynomial over $\mathbb{F}_q$	$O(d^3\log(q))$ $\mathbb{F}_q$-operations $*$
Construct polynomial basis for $\mathbb{F}_{q^m}$	$O(m^4\log(q))$ $\mathbb{F}_q$-operations $*$
Construct normal basis for $\mathbb{F}_{q^m}$ given a poly basis	$O(m^3\log_q(m))$ $\mathbb{F}_q$-operations $*$
Solve quadratic equations in $\mathbb{F}_q$	$O(\log(q)^4)$ bit operations $*$
Compute the minimal poly over $\mathbb{F}_q$ of $\alpha \in \mathbb{F}_{q^m}$	$O(m^3)$ $\mathbb{F}_q$-operations
Compute an isomorphism between repns of $\mathbb{F}_{q^m}$	$O(m^3)$ $\mathbb{F}_q$-operations
Compute order of $\alpha \in \mathbb{F}_q^*$ given factorisation of $q-1$	$O(\log(q)\log(\log(q)))$ $\mathbb{F}_q$-multiplications
Compute primitive root in $\mathbb{F}_q^*$ given factorisation of $q-1$	$O(\log(q)\log(\log(q)))$ $\mathbb{F}_q$-multiplications
Compute $f(\alpha_j) \in \Bbbk$ for $f \in \Bbbk[x]$ of degree n and $\alpha_1, \ldots, \alpha_n \in \Bbbk$	$O(M(n)\log(n))$ $\Bbbk$-multiplications