Chapter 3

Number-Theoretic Reference Problems

Contents in Brief

3.1 Introduction and overview

The security of many public-key cryptosystems relies on the apparent intractability of the computational problems studied in this chapter. In a cryptographic setting, it is prudent to make the assumption that the adversary is very powerful. Thus, informally speaking, a computational problem is said to be *easy* or *tractable* if it can be solved in (expected)[1] polynomial time, at least for a non-negligible fraction of all possible inputs. In other words, if there is an algorithm which can solve a non-negligible fraction of all instances of a problem in polynomial time, then any cryptosystem whose security is based on that problem must be considered insecure.

The computational problems studied in this chapter are summarized in Table 3.1. The true computational complexities of these problems are not known. That is to say, they are widely believed to be intractable,[2] although no proof of this is known. Generally, the only lower bounds known on the resources required to solve these problems are the trivial linear bounds, which do not provide any evidence of their intractability. It is, therefore, of interest to study their relative difficulties. For this reason, various techniques of reducing one

[1]For simplicity, the remainder of the chapter shall generally not distinguish between deterministic polynomial-time algorithms and randomized algorithms (see §2.3.4) whose *expected* running time is polynomial.

[2]More precisely, these problems are intractable if the problem parameters are carefully chosen.

Problem	Description
FACTORING	*Integer factorization problem*: given a positive integer n, find its prime factorization; that is, write $n = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k}$ where the p_i are pairwise distinct primes and each $e_i \geq 1$.
RSAP	*RSA problem* (also known as *RSA inversion*): given a positive integer n that is a product of two distinct odd primes p and q, a positive integer e such that $\gcd(e, (p-1)(q-1)) = 1$, and an integer c, find an integer m such that $m^e \equiv c \pmod{n}$.
QRP	*Quadratic residuosity problem*: given an odd composite integer n and an integer a having Jacobi symbol $\left(\frac{a}{n}\right) = 1$, decide whether or not a is a quadratic residue modulo n.
SQROOT	*Square roots modulo n*: given a composite integer n and $a \in Q_n$ (the set of quadratic residues modulo n), find a square root of a modulo n; that is, an integer x such that $x^2 \equiv a \pmod{n}$.
DLP	*Discrete logarithm problem*: given a prime p, a generator α of $\mathbb{Z}_p^*$, and an element $\beta \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq p-2$, such that $\alpha^x \equiv \beta \pmod{p}$.
GDLP	*Generalized discrete logarithm problem*: given a finite cyclic group G of order n, a generator α of G, and an element $\beta \in G$, find the integer x, $0 \leq x \leq n-1$, such that $\alpha^x = \beta$.
DHP	*Diffie-Hellman problem*: given a prime p, a generator α of $\mathbb{Z}_p^*$, and elements $\alpha^a \bmod p$ and $\alpha^b \bmod p$, find $\alpha^{ab} \bmod p$.
GDHP	*Generalized Diffie-Hellman problem*: given a finite cyclic group G, a generator α of G, and group elements α^a and α^b, find α^{ab}.
SUBSET-SUM	*Subset sum problem*: given a set of positive integers $\{a_1, a_2, \ldots, a_n\}$ and a positive integer s, determine whether or not there is a subset of the a_j that sums to s.

Table 3.1: *Some computational problems of cryptographic relevance.*

computational problem to another have been devised and studied in the literature. These reductions provide a means for converting any algorithm that solves the second problem into an algorithm for solving the first problem. The following intuitive notion of reducibility (cf. §2.3.3) is used in this chapter.

3.1 Definition Let A and B be two computational problems. A is said to *polytime reduce* to B, written $A \leq_P B$, if there is an algorithm that solves A which uses, as a subroutine, a hypothetical algorithm for solving B, and which runs in polynomial time if the algorithm for B does.[3]

Informally speaking, if A polytime reduces to B, then B is at least as difficult as A; equivalently, A is no harder than B. Consequently, if A is a well-studied computational problem that is widely believed to be intractable, then proving that $A \leq_P B$ provides strong evidence of the intractability of problem B.

3.2 Definition Let A and B be two computational problems. If $A \leq_P B$ and $B \leq_P A$, then A and B are said to be *computationally equivalent*, written $A \equiv_P B$.

[3]In the literature, the hypothetical polynomial-time subroutine for B is sometimes called an *oracle* for B.

Informally speaking, if $A \equiv_P B$ then A and B are either both tractable or both intractable, as the case may be.

Chapter outline

The remainder of the chapter is organized as follows. Algorithms for the integer factorization problem are studied in §3.2. Two problems related to factoring, the RSA problem and the quadratic residuosity problem, are briefly considered in §3.3 and §3.4. Efficient algorithms for computing square roots in $\mathbb{Z}_p$, p a prime, are presented in §3.5, and the equivalence of the problems of finding square roots modulo a composite integer n and factoring n is established. Algorithms for the discrete logarithm problem are studied in §3.6, and the related Diffie-Hellman problem is briefly considered in §3.7. The relation between the problems of factoring a composite integer n and computing discrete logarithms in (cyclic subgroups of) the group $\mathbb{Z}_n^*$ is investigated in §3.8. The tasks of finding partial solutions to the discrete logarithm problem, the RSA problem, and the problem of computing square roots modulo a composite integer n are the topics of §3.9. The L^3-lattice basis reduction algorithm is presented in §3.10, along with algorithms for the subset sum problem and for simultaneous diophantine approximation. Berlekamp's Q-matrix algorithm for factoring polynomials is presented in §3.11. Finally, §3.12 provides references and further chapter notes.

3.2 The integer factorization problem

The security of many cryptographic techniques depends upon the intractability of the integer factorization problem. A partial list of such protocols includes the RSA public-key encryption scheme (§8.2), the RSA signature scheme (§11.3.1), and the Rabin public-key encryption scheme (§8.3). This section summarizes the current knowledge on algorithms for the integer factorization problem.

3.3 Definition The *integer factorization problem* (FACTORING) is the following: given a positive integer n, find its prime factorization; that is, write $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ where the p_i are pairwise distinct primes and each $e_i \geq 1$.

3.4 Remark (*primality testing vs. factoring*) The problem of *deciding* whether an integer is composite or prime seems to be, in general, much easier than the factoring problem. Hence, before attempting to factor an integer, the integer should be tested to make sure that it is indeed composite. Primality tests are a main topic of Chapter 4.

3.5 Remark (*splitting vs. factoring*) A *non-trivial factorization* of n is a factorization of the form $n = ab$ where $1 < a < n$ and $1 < b < n$; a and b are said to be *non-trivial factors* of n. Here a and b are not necessarily prime. To solve the integer factorization problem, it suffices to study algorithms that *split* n, that is, find a non-trivial factorization $n = ab$. Once found, the factors a and b can be tested for primality. The algorithm for splitting integers can then be recursively applied to a and/or b, if either is found to be composite. In this manner, the prime factorization of n can be obtained.

3.6 Note (*testing for perfect powers*) If $n \geq 2$, it can be efficiently checked as follows whether or not n is a *perfect power*, i.e., $n = x^k$ for some integers $x \geq 2$, $k \geq 2$. For each prime

$p \leq \lg n$, an integer approximation x of $n^{1/p}$ is computed. This can be done by performing a binary search for x satisfying $n = x^p$ in the interval $[2, 2^{\lfloor \lg n/p \rfloor + 1}]$. The entire procedure takes $O((\lg^3 n) \lg\lg\lg n)$ bit operations. For the remainder of this section, it will always be assumed that n is not a perfect power. It follows that if n is composite, then n has at least two distinct prime factors.

Some factoring algorithms are tailored to perform better when the integer n being factored is of a special form; these are called *special-purpose* factoring algorithms. The running times of such algorithms typically depend on certain properties of the factors of n. Examples of special-purpose factoring algorithms include trial division (§3.2.1), Pollard's rho algorithm (§3.2.2), Pollard's $p-1$ algorithm (§3.2.3), the elliptic curve algorithm (§3.2.4), and the special number field sieve (§3.2.7). In contrast, the running times of the so-called *general-purpose* factoring algorithms depend solely on the size of n. Examples of general-purpose factoring algorithms include the quadratic sieve (§3.2.6) and the general number field sieve (§3.2.7).

Whenever applicable, special-purpose algorithms should be employed as they will generally be more efficient. A reasonable overall strategy is to attempt to find small factors first, capitalize on any particular special forms an integer may have, and then, if all else fails, bring out the general-purpose algorithms. As an example of a general strategy, one might consider the following.

1. Apply trial division by small primes less than some bound b_1.
2. Next, apply Pollard's rho algorithm, hoping to find any small prime factors smaller than some bound b_2, where $b_2 > b_1$.
3. Apply the elliptic curve factoring algorithm, hoping to find any small factors smaller than some bound b_3, where $b_3 > b_2$.
4. Finally, apply one of the more powerful general-purpose algorithms (quadratic sieve or general number field sieve).

3.2.1 Trial division

Once it is established that an integer n is composite, before expending vast amounts of time with more powerful techniques, the first thing that should be attempted is trial division by all "small" primes. Here, "small" is determined as a function of the size of n. As an extreme case, trial division can be attempted by all primes up to $\sqrt{n}$. If this is done, trial division will completely factor n but the procedure will take roughly $\sqrt{n}$ divisions in the worst case when n is a product of two primes of the same size. In general, if the factors found at each stage are tested for primality, then trial division to factor n completely takes $O(p + \lg n)$ divisions, where p is the second-largest prime factor of n.

Fact 3.7 indicates that if trial division is used to factor a randomly chosen large integer n, then the algorithm can be expected to find some small factors of n relatively quickly, and expend a large amount of time to find the second largest prime factor of n.

3.7 Fact Let n be chosen uniformly at random from the interval $[1, x]$.

(i) If $\frac{1}{2} \leq \alpha \leq 1$, then the probability that the largest prime factor of n is $\leq x^\alpha$ is approximately $1 + \ln \alpha$. Thus, for example, the probability that n has a prime factor $> \sqrt{x}$ is $\ln 2 \approx 0.69$.

(ii) The probability that the second-largest prime factor of n is $\leq x^{0.2117}$ is about $\frac{1}{2}$.

(iii) The expected total number of prime factors of n is $\ln\ln x + O(1)$. (If $n = \prod p_i^{e_i}$, the *total* number of prime factors of n is $\sum e_i$.)

3.2.2 Pollard's rho factoring algorithm

Pollard's rho algorithm is a special-purpose factoring algorithm for finding small factors of a composite integer.

Let $f : S \longrightarrow S$ be a random function, where S is a finite set of cardinality n. Let x_0 be a random element of S, and consider the sequence $x_0, x_1, x_2, \ldots$ defined by $x_{i+1} = f(x_i)$ for $i \geq 0$. Since S is finite, the sequence must eventually cycle, and consists of a *tail* of expected length $\sqrt{\pi n/8}$ followed by an endlessly repeating *cycle* of expected length $\sqrt{\pi n/8}$ (see Fact 2.37). A problem that arises in some cryptanalytic tasks, including integer factorization (Algorithm 3.9) and the discrete logarithm problem (Algorithm 3.60), is of finding distinct indices i and j such that $x_i = x_j$ (a *collision* is then said to have occurred).

An obvious method for finding a collision is to compute and store x_i for $i = 0, 1, 2, \ldots$ and look for duplicates. The expected number of inputs that must be tried before a duplicate is detected is $\sqrt{\pi n/2}$ (Fact 2.27). This method requires $O(\sqrt{n})$ memory and $O(\sqrt{n})$ time, assuming the x_i are stored in a hash table so that new entries can be added in constant time.

3.8 Note (*Floyd's cycle-finding algorithm*) The large storage requirements in the above technique for finding a collision can be eliminated by using *Floyd's cycle-finding algorithm*. In this method, one starts with the pair (x_1, x_2), and iteratively computes (x_i, x_{2i}) from the previous pair (x_{i-1}, x_{2i-2}), until $x_m = x_{2m}$ for some m. If the tail of the sequence has length λ and the cycle has length μ, then the first time that $x_m = x_{2m}$ is when $m = \mu(1 + \lfloor \lambda/\mu \rfloor)$. Note that $\lambda < m \leq \lambda + \mu$, and consequently the expected running time of this method is $O(\sqrt{n})$.

Now, let p be a prime factor of a composite integer n. Pollard's rho algorithm for factoring n attempts to find duplicates in the sequence of integers $x_0, x_1, x_2, \ldots$ defined by $x_0 = 2$, $x_{i+1} = f(x_i) = x_i^2 + 1 \bmod p$ for $i \geq 0$. Floyd's cycle-finding algorithm is utilized to find x_m and x_{2m} such that $x_m \equiv x_{2m} \pmod{p}$. Since p divides n but is unknown, this is done by computing the terms x_i modulo n and testing if $\gcd(x_m - x_{2m}, n) > 1$. If also $\gcd(x_m - x_{2m}, n) < n$, then a non-trivial factor of n is obtained. (The situation $\gcd(x_m - x_{2m}, n) = n$ occurs with negligible probability.)

3.9 Algorithm Pollard's rho algorithm for factoring integers

INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

1. Set $a \leftarrow 2$, $b \leftarrow 2$.
2. For $i = 1, 2, \ldots$ do the following:
   - 2.1 Compute $a \leftarrow a^2 + 1 \bmod n$, $b \leftarrow b^2 + 1 \bmod n$, $b \leftarrow b^2 + 1 \bmod n$.
   - 2.2 Compute $d = \gcd(a - b, n)$.
   - 2.3 If $1 < d < n$ then return(d) and terminate with success.
   - 2.4 If $d = n$ then terminate the algorithm with failure (see Note 3.12).

3.10 Example (*Pollard's rho algorithm for finding a non-trivial factor of* $n = 455459$) The following table lists the values of variables a, b, and d at the end of each iteration of step 2 of Algorithm 3.9.

a	b	d
5	26	1
26	2871	1
677	179685	1
2871	155260	1
44380	416250	1
179685	43670	1
121634	164403	1
155260	247944	1
44567	68343	743

Hence two non-trivial factors of 455459 are 743 and $455459/743 = 613$. □

3.11 Fact Assuming that the function $f(x) = x^2 + 1 \bmod p$ behaves like a random function, the expected time for Pollard's rho algorithm to find a factor p of n is $O(\sqrt{p})$ modular multiplications. This implies that the expected time to find a non-trivial factor of n is $O(n^{1/4})$ modular multiplications.

3.12 Note (*options upon termination with failure*) If Pollard's rho algorithm terminates with failure, one option is to try again with a different polynomial f having integer coefficients instead of $f(x) = x^2 + 1$. For example, the polynomial $f(x) = x^2 + c$ may be used as long as $c \neq 0, -2$.

3.2.3 Pollard's $p-1$ factoring algorithm

Pollard's $p-1$ factoring algorithm is a special-purpose factoring algorithm that can be used to efficiently find any prime factors p of a composite integer n for which $p-1$ is smooth (see Definition 3.13) with respect to some relatively small bound B.

3.13 Definition Let B be a positive integer. An integer n is said to be B-*smooth*, or *smooth with respect to a bound* B, if all its prime factors are $\leq B$.

The idea behind Pollard's $p-1$ algorithm is the following. Let B be a smoothness bound. Let Q be the least common multiple of all powers of primes $\leq B$ that are $\leq n$. If $q^l \leq n$, then $l \ln q \leq \ln n$, and so $l \leq \lfloor \frac{\ln n}{\ln q} \rfloor$. Thus

$$Q = \prod_{q \leq B} q^{\lfloor \ln n / \ln q \rfloor},$$

where the product is over all distinct primes $q \leq B$. If p is a prime factor of n such that $p-1$ is B-smooth, then $p-1|Q$, and consequently for any a satisfying $\gcd(a,p) = 1$, Fermat's theorem (Fact 2.127) implies that $a^Q \equiv 1 \pmod{p}$. Hence if $d = \gcd(a^Q - 1, n)$, then $p|d$. It is possible that $d = n$, in which case the algorithm fails; however, this is unlikely to occur if n has at least two large distinct prime factors.

3.14 Algorithm Pollard's $p-1$ algorithm for factoring integers

INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

1. Select a smoothness bound B.
2. Select a random integer a, $2 \leq a \leq n-1$, and compute $d = \gcd(a, n)$. If $d \geq 2$ then return(d).
3. For each prime $q \leq B$ do the following:
   3.1 Compute $l = \lfloor \frac{\ln n}{\ln q} \rfloor$.
   3.2 Compute $a \leftarrow a^{q^l} \bmod n$ (using Algorithm 2.143).
4. Compute $d = \gcd(a-1, n)$.
5. If $d = 1$ or $d = n$, then terminate the algorithm with failure. Otherwise, return(d).

3.15 Example (*Pollard's $p-1$ algorithm for finding a non-trivial factor of $n = 19048567$*)

1. Select the smoothness bound $B = 19$.
2. Select the integer $a = 3$ and compute $\gcd(3, n) = 1$.
3. The following table lists the intermediate values of the variables q, l, and a after each iteration of step 3 in Algorithm 3.14:

q	l	a
2	24	2293244
3	15	13555889
5	10	16937223
7	8	15214586
11	6	9685355
13	6	13271154
17	5	11406961
19	5	554506

4. Compute $d = \gcd(554506 - 1, n) = 5281$.
5. Two non-trivial factors of n are $p = 5281$ and $q = n/p = 3607$ (these factors are in fact prime).

Notice that $p - 1 = 5280 = 2^5 \times 3 \times 5 \times 11$, and $q - 1 = 3606 = 2 \times 3 \times 601$. That is, $p-1$ is 19-smooth, while $q-1$ is not 19-smooth. □

3.16 Fact Let n be an integer having a prime factor p such that $p-1$ is B-smooth. The running time of Pollard's $p-1$ algorithm for finding the factor p is $O(B \ln n / \ln B)$ modular multiplications.

3.17 Note (*improvements*) The smoothness bound B in Algorithm 3.14 is selected based on the amount of time one is willing to spend on Pollard's $p-1$ algorithm before moving on to more general techniques. In practice, B may be between 10^5 and 10^6. If the algorithm terminates with $d = 1$, then one might try searching over prime numbers $q_1, q_2, \ldots, q_l$ larger than B by first computing $a \leftarrow a^{q_i} \bmod n$ for $1 \leq i \leq l$, and then computing $d = \gcd(a-1, n)$. Another variant is to start with a large bound B, and repeatedly execute step 3 for a few primes q followed by the gcd computation in step 4. There are numerous other practical improvements of the algorithm (see page 125).

3.2.4 Elliptic curve factoring

The details of the *elliptic curve factoring algorithm* are beyond the scope of this book; nevertheless, a rough outline follows. The success of Pollard's $p-1$ algorithm hinges on $p-1$ being smooth for some prime divisor p of n; if no such p exists, then the algorithm fails. Observe that $p-1$ is the order of the group $\mathbb{Z}_p^*$. The elliptic curve factoring algorithm is a generalization of Pollard's $p-1$ algorithm in the sense that the group $\mathbb{Z}_p^*$ is replaced by a random elliptic curve group over $\mathbb{Z}_p$. The order of such a group is roughly uniformly distributed in the interval $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. If the order of the group chosen is smooth with respect to some pre-selected bound, the elliptic curve algorithm will, with high probability, find a non-trivial factor of n. If the group order is not smooth, then the algorithm will likely fail, but can be repeated with a different choice of elliptic curve group.

The elliptic curve algorithm has an expected running time of $L_p[\frac{1}{2}, \sqrt{2}]$ (see Example 2.61 for definition of L_p) to find a factor p of n. Since this running time depends on the size of the prime factors of n, the algorithm tends to find small such factors first. The elliptic curve algorithm is, therefore, classified as a special-purpose factoring algorithm. It is currently the algorithm of choice for finding t-decimal digit prime factors, for $t \leq 40$, of very large composite integers.

In the hardest case, when n is a product of two primes of roughly the same size, the expected running time of the elliptic curve algorithm is $L_n[\frac{1}{2}, 1]$, which is the same as that of the quadratic sieve (§3.2.6). However, the elliptic curve algorithm is not as efficient as the quadratic sieve in practice for such integers.

3.2.5 Random square factoring methods

The basic idea behind the random square family of methods is the following. Suppose x and y are integers such that $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$. Then n divides $x^2-y^2 = (x-y)(x+y)$ but n does not divide either $(x-y)$ or $(x+y)$. Hence, $\gcd(x-y, n)$ must be a non-trivial factor of n. This result is summarized next.

3.18 Fact Let x, y, and n be integers. If $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$, then $\gcd(x-y, n)$ is a non-trivial factor of n.

The random square methods attempt to find integers x and y at random so that $x^2 \equiv y^2 \pmod{n}$. Then, as shown in Fact 3.19, with probability at least $\frac{1}{2}$ it is the case that $x \not\equiv \pm y \pmod{n}$, whence $\gcd(x-y, n)$ will yield a non-trivial factor of n.

3.19 Fact Let n be an odd composite integer that is divisible by k distinct odd primes. If $a \in \mathbb{Z}_n^*$, then the congruence $x^2 \equiv a^2 \pmod{n}$ has exactly 2^k solutions modulo n, two of which are $x = a$ and $x = -a$.

3.20 Example Let $n = 35$. Then there are four solutions to the congruence $x^2 \equiv 4 \pmod{35}$, namely $x = 2$, 12, 23, and 33. □

A common strategy employed by the random square algorithms for finding x and y at random satisfying $x^2 \equiv y^2 \pmod{n}$ is the following. A set consisting of the first t primes $S = \{p_1, p_2, \ldots, p_t\}$ is chosen; S is called the *factor base*. Proceed to find pairs of integers (a_i, b_i) satisfying

(i) $a_i^2 \equiv b_i \pmod{n}$; and

(ii) $b_i = \prod_{j=1}^{t} p_j^{e_{ij}}$, $e_{ij} \geq 0$; that is, b_i is p_t-smooth.

Next find a subset of the b_i's whose product is a perfect square. Knowing the factorizations of the b_i's, this is possible by selecting a subset of the b_i's such that the power of each prime p_j appearing in their product is even. For this purpose, only the parity of the non-negative integer exponents e_{ij} needs to be considered. Thus, to simplify matters, for each i, associate the binary vector $v_i = (v_{i1}, v_{i2}, \ldots, v_{it})$ with the integer exponent vector $(e_{i1}, e_{i2}, \ldots, e_{it})$ such that $v_{ij} = e_{ij} \bmod 2$. If $t+1$ pairs (a_i, b_i) are obtained, then the t-dimensional vectors $v_1, v_2, \ldots, v_{t+1}$ must be linearly dependent over $\mathbb{Z}_2$. That is, there must exist a non-empty subset $T \subseteq \{1, 2, \ldots, t+1\}$ such that $\sum_{i \in T} v_i = 0$ over $\mathbb{Z}_2$, and hence $\prod_{i \in T} b_i$ is a perfect square. The set T can be found using ordinary linear algebra over $\mathbb{Z}_2$. Clearly, $\prod_{i \in T} a_i^2$ is also a perfect square. Thus setting $x = \prod_{i \in T} a_i$ and y to be the integer square root of $\prod_{i \in T} b_i$ yields a pair of integers (x, y) satisfying $x^2 \equiv y^2 \pmod{n}$. If this pair also satisfies $x \not\equiv \pm y \pmod{n}$, then $\gcd(x - y, n)$ yields a non-trivial factor of n. Otherwise, some of the (a_i, b_i) pairs may be replaced by some new such pairs, and the process is repeated. In practice, there will be several dependencies among the vectors $v_1, v_2, \ldots, v_{t+1}$, and with high probability at least one will yield an (x, y) pair satisfying $x \not\equiv \pm y \pmod{n}$; hence, this last step of generating new (a_i, b_i) pairs does not usually occur.

This description of the random square methods is incomplete for two reasons. Firstly, the optimal choice of t, the size of the factor base, is not specified; this is addressed in Note 3.24. Secondly, a method for efficiently generating the pairs (a_i, b_i) is not specified. Several techniques have been proposed. In the simplest of these, called *Dixon's algorithm*, a_i is chosen at random, and $b_i = a_i^2 \bmod n$ is computed. Next, trial division by elements in the factor base is used to test whether b_i is p_t-smooth. If not, then another integer a_i is chosen at random, and the procedure is repeated.

The more efficient techniques strategically select an a_i such that b_i is relatively small. Since the proportion of p_t-smooth integers in the interval $[2, x]$ becomes larger as x decreases, the probability of such b_i being p_t-smooth is higher. The most efficient of such techniques is the quadratic sieve algorithm, which is described next.

3.2.6 Quadratic sieve factoring

Suppose an integer n is to be factored. Let $m = \lfloor \sqrt{n} \rfloor$, and consider the polynomial $q(x) = (x+m)^2 - n$. Note that

$$q(x) \;=\; x^2 + 2mx + m^2 - n \;\approx\; x^2 + 2mx, \tag{3.1}$$

which is small (relative to n) if x is small in absolute value. The quadratic sieve algorithm selects $a_i = (x+m)$ and tests whether $b_i = (x+m)^2 - n$ is p_t-smooth. Note that $a_i^2 = (x+m)^2 \equiv b_i \pmod{n}$. Note also that if a prime p divides b_i then $(x+m)^2 \equiv n \pmod{p}$, and hence n is a quadratic residue modulo p. Thus the factor base need only contain those primes p for which the Legendre symbol $\left(\frac{n}{p}\right)$ is 1 (Definition 2.145). Furthermore, since b_i may be negative, -1 is included in the factor base. The steps of the quadratic sieve algorithm are summarized in Algorithm 3.21.

3.21 Algorithm Quadratic sieve algorithm for factoring integers

INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

1. Select the factor base $S = \{p_1, p_2, \ldots, p_t\}$, where $p_1 = -1$ and p_j $(j \geq 2)$ is the $(j-1)^{\text{th}}$ prime p for which n is a quadratic residue modulo p.
2. Compute $m = \lfloor\sqrt{n}\rfloor$.
3. (*Collect* $t+1$ *pairs* (a_i, b_i). *The* x *values are chosen in the order* $0, \pm1, \pm2, \ldots$.) Set $i \leftarrow 1$. While $i \leq t+1$ do the following:
   - 3.1 Compute $b = q(x) = (x+m)^2 - n$, and test using trial division (cf. Note 3.23) by elements in S whether b is p_t-smooth. If not, pick a new x and repeat step 3.1.
   - 3.2 If b is p_t-smooth, say $b = \prod_{j=1}^{t} p_j^{e_{ij}}$, then set $a_i \leftarrow (x+m)$, $b_i \leftarrow b$, and $v_i = (v_{i1}, v_{i2}, \ldots, v_{it})$, where $v_{ij} = e_{ij} \bmod 2$ for $1 \leq j \leq t$.
   - 3.3 $i \leftarrow i+1$.
4. Use linear algebra over $\mathbb{Z}_2$ to find a non-empty subset $T \subseteq \{1, 2, \ldots, t+1\}$ such that $\sum_{i \in T} v_i = 0$.
5. Compute $x = \prod_{i \in T} a_i \bmod n$.
6. For each j, $1 \leq j \leq t$, compute $l_j = (\sum_{i \in T} e_{ij})/2$.
7. Compute $y = \prod_{j=1}^{t} p_j^{l_j} \bmod n$.
8. If $x \equiv \pm y \pmod{n}$, then find another non-empty subset $T \subseteq \{1, 2, \ldots, t+1\}$ such that $\sum_{i \in T} v_i = 0$, and go to step 5. (In the unlikely case such a subset T does not exist, replace a few of the (a_i, b_i) pairs with new pairs (step 3), and go to step 4.)
9. Compute $d = \gcd(x - y, n)$ and return(d).

3.22 Example (*quadratic sieve algorithm for finding a non-trivial factor of* $n = 24961$)

1. Select the factor base $S = \{-1, 2, 3, 5, 13, 23\}$ of size $t = 6$. (7, 11, 17 and 19 are omitted from S since $\left(\frac{n}{p}\right) = -1$ for these primes.)
2. Compute $m = \lfloor\sqrt{24961}\rfloor = 157$.
3. Following is the data collected for the first $t+1$ values of x for which $q(x)$ is 23-smooth.

i	x	$q(x)$	factorization of $q(x)$	a_i	v_i
1	0	-312	$-2^3 \cdot 3 \cdot 13$	157	$(1,1,1,0,1,0)$
2	1	3	3	158	$(0,0,1,0,0,0)$
3	-1	-625	-5^4	156	$(1,0,0,0,0,0)$
4	2	320	$2^6 \cdot 5$	159	$(0,0,0,1,0,0)$
5	-2	-936	$-2^3 \cdot 3^2 \cdot 13$	155	$(1,1,0,0,1,0)$
6	4	960	$2^6 \cdot 3 \cdot 5$	161	$(0,0,1,1,0,0)$
7	-6	-2160	$-2^4 \cdot 3^3 \cdot 5$	151	$(1,0,1,1,0,0)$

4. By inspection, $v_1 + v_2 + v_5 = 0$. (In the notation of Algorithm 3.21, $T = \{1, 2, 5\}$.)
5. Compute $x = (a_1 a_2 a_5 \bmod n) = 936$.
6. Compute $l_1 = 1, l_2 = 3, l_3 = 2, l_4 = 0, l_5 = 1, l_6 = 0$.
7. Compute $y = -2^3 \cdot 3^2 \cdot 13 \bmod n = 24025$.
8. Since $936 \equiv -24025 \pmod{n}$, another linear dependency must be found.
9. By inspection, $v_3 + v_6 + v_7 = 0$; thus $T = \{3, 6, 7\}$.
10. Compute $x = (a_3 a_6 a_7 \bmod n) = 23405$.
11. Compute $l_1 = 1, l_2 = 5, l_3 = 2, l_4 = 3, l_5 = 0, l_6 = 0$.

12. Compute $y = (-2^5 \cdot 3^2 \cdot 5^3 \bmod n) = 13922$.
13. Now, $23405 \not\equiv \pm 13922 \pmod{n}$, so compute $\gcd(x-y, n) = \gcd(9483, 24961) = 109$. Hence, two non-trivial factors of 24961 are 109 and 229. □

3.23 Note (*sieving*) Instead of testing smoothness by trial division in step 3.1 of Algorithm 3.21, a more efficient technique known as *sieving* is employed in practice. Observe first that if p is an odd prime in the factor base and p divides $q(x)$, then p also divides $q(x+lp)$ for every integer l. Thus by solving the equation $q(x) \equiv 0 \pmod{p}$ for x (for example, using the algorithms in §3.5.1), one knows either one or two (depending on the number of solutions to the quadratic equation) entire sequences of other values y for which p divides $q(y)$.

The *sieving process* is the following. An array $Q[\,]$ indexed by x, $-M \le x \le M$, is created and the x^{th} entry is initialized to $\lfloor \lg |q(x)| \rfloor$. Let x_1, x_2 be the solutions to $q(x) \equiv 0 \pmod{p}$, where p is an odd prime in the factor base. Then the value $\lfloor \lg p \rfloor$ is subtracted from those entries $Q[x]$ in the array for which $x \equiv x_1$ or $x_2 \pmod{p}$ and $-M \le x \le M$. This is repeated for each odd prime p in the factor base. (The case of $p = 2$ and prime powers can be handled in a similar manner.) After the sieving, the array entries $Q[x]$ with values near 0 are most likely to be p_t-smooth (roundoff errors must be taken into account), and this can be verified by factoring $q(x)$ by trial division.

3.24 Note (*running time of the quadratic sieve*) To optimize the running time of the quadratic sieve, the size of the factor base should be judiciously chosen. The optimal selection of $t \approx L_n[\frac{1}{2}, \frac{1}{2}]$ (see Example 2.61) is derived from knowledge concerning the distribution of smooth integers close to $\sqrt{n}$. With this choice, Algorithm 3.21 with sieving (Note 3.23) has an expected running time of $L_n[\frac{1}{2}, 1]$, independent of the size of the factors of n.

3.25 Note (*multiple polynomial variant*) In order to collect a sufficient number of (a_i, b_i) pairs, the sieving interval must be quite large. From equation (3.1) it can be seen that $|q(x)|$ increases linearly with $|x|$, and consequently the probability of smoothness decreases. To overcome this problem, a variant (the *multiple polynomial quadratic sieve*) was proposed whereby many appropriately-chosen quadratic polynomials can be used instead of just $q(x)$, each polynomial being sieved over an interval of much smaller length. This variant also has an expected running time of $L_n[\frac{1}{2}, 1]$, and is the method of choice in practice.

3.26 Note (*parallelizing the quadratic sieve*) The multiple polynomial variant of the quadratic sieve is well suited for parallelization. Each node of a parallel computer, or each computer in a network of computers, simply sieves through different collections of polynomials. Any (a_i, b_i) pair found is reported to a central processor. Once sufficient pairs have been collected, the corresponding system of linear equations is solved on a single (possibly parallel) computer.

3.27 Note (*quadratic sieve vs. elliptic curve factoring*) The elliptic curve factoring algorithm (§3.2.4) has the same[4] expected (asymptotic) running time as the quadratic sieve factoring algorithm in the special case when n is the product of two primes of equal size. However, for such numbers, the quadratic sieve is superior in practice because the main steps in the algorithm are single precision operations, compared to the much more computationally intensive multi-precision elliptic curve operations required in the elliptic curve algorithm.

[4]This does not take into account the different $o(1)$ terms in the two expressions $L_n[\frac{1}{2}, 1]$.

3.2.7 Number field sieve factoring

For several years it was believed by some people that a running time of $L_n[\frac{1}{2}, 1]$ was, in fact, the best achievable by any integer factorization algorithm. This barrier was broken in 1990 with the discovery of the *number field sieve*. Like the quadratic sieve, the number field sieve is an algorithm in the random square family of methods (§3.2.5). That is, it attempts to find integers x and y such that $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$. To achieve this goal, two factor bases are used, one consisting of all prime numbers less than some bound, and the other consisting of all prime ideals of norm less than some bound in the ring of integers of a suitably-chosen algebraic number field. The details of the algorithm are quite complicated, and are beyond the scope of this book.

A special version of the algorithm (the *special number field sieve*) applies to integers of the form $n = r^e - s$ for small r and $|s|$, and has an expected running time of $L_n[\frac{1}{3}, c]$, where $c = (32/9)^{1/3} \approx 1.526$.

The general version of the algorithm, sometimes called the *general number field sieve*, applies to all integers and has an expected running time of $L_n[\frac{1}{3}, c]$, where $c = (64/9)^{1/3} \approx 1.923$. This is, asymptotically, the fastest algorithm known for integer factorization. The primary reason why the running time of the number field sieve is smaller than that of the quadratic sieve is that the candidate smooth numbers in the former are much smaller than those in the latter.

The general number field sieve was at first believed to be slower than the quadratic sieve for factoring integers having fewer than 150 decimal digits. However, experiments in 1994–1996 have indicated that the general number field sieve is substantially faster than the quadratic sieve even for numbers in the 115 digit range. This implies that the crossover point between the effectiveness of the quadratic sieve vs. the general number field sieve may be 110–120 digits. For this reason, the general number field sieve is considered the current champion of all general-purpose factoring algorithms.

3.3 The RSA problem

The intractability of the RSA problem forms the basis for the security of the RSA public-key encryption scheme (§8.2) and the RSA signature scheme (§11.3.1).

3.28 Definition The *RSA problem* (RSAP) is the following: given a positive integer n that is a product of two distinct odd primes p and q, a positive integer e such that $\gcd(e, (p-1)(q-1)) = 1$, and an integer c, find an integer m such that $m^e \equiv c \pmod{n}$.

In other words, the RSA problem is that of finding e^{th} roots modulo a composite integer n. The conditions imposed on the problem parameters n and e ensure that for each integer $c \in \{0, 1, \ldots, n-1\}$ there is exactly one $m \in \{0, 1, \ldots, n-1\}$ such that $m^e \equiv c \pmod{n}$. Equivalently, the function $f : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n$ defined as $f(m) = m^e \bmod n$ is a permutation.

3.29 Remark (*SQROOT vs. RSA problems*) Since $p - 1$ is even, it follows that e is odd. In particular, $e \neq 2$, and hence the SQROOT problem (Definition 3.43) is *not* a special case of the RSA problem.

As is shown in §8.2.2(i), if the factors of n are known then the RSA problem can be easily solved. This fact is stated next.

3.30 Fact RSAP $\leq_P$ FACTORING. That is, the RSA problem polytime reduces to the integer factorization problem.

It is widely believed that the RSA and the integer factorization problems are computationally equivalent, although no proof of this is known.

3.4 The quadratic residuosity problem

The security of the Goldwasser-Micali probabilistic public-key encryption scheme (§8.7) and the Blum-Blum-Shub pseudorandom bit generator (§5.5.2) are both based on the apparent intractability of the quadratic residuosity problem.

Recall from §2.4.5 that if $n \geq 3$ is an odd integer, then J_n is the set of all $a \in \mathbb{Z}_n^*$ having Jacobi symbol 1. Recall also that Q_n is the set of quadratic residues modulo n and that the set of pseudosquares modulo n is defined by $\widetilde{Q}_n = J_n - Q_n$.

3.31 Definition The *quadratic residuosity problem* (QRP) is the following: given an odd composite integer n and $a \in J_n$, decide whether or not a is a quadratic residue modulo n.

3.32 Remark (*QRP with a prime modulus*) If n is a prime, then it is easy to decide whether $a \in \mathbb{Z}_n^*$ is a quadratic residue modulo n since, by definition, $a \in Q_n$ if and only if $\left(\frac{a}{n}\right) = 1$, and the Legendre symbol $\left(\frac{a}{n}\right)$ can be efficiently calculated by Algorithm 2.149.

Assume now that n is a product of two distinct odd primes p and q. It follows from Fact 2.137 that if $a \in J_n$, then $a \in Q_n$ if and only if $\left(\frac{a}{p}\right) = 1$. Thus, if the factorization of n is known, then QRP can be solved simply by computing the Legendre symbol $\left(\frac{a}{p}\right)$. This observation can be generalized to all integers n and leads to the following fact.

3.33 Fact QRP $\leq_P$ FACTORING. That is, the QRP polytime reduces to the FACTORING problem.

On the other hand, if the factorization of n is unknown, then there is no efficient procedure known for solving QRP, other than by guessing the answer. If $n = pq$, then the probability of a correct guess is $\frac{1}{2}$ since $|Q_n| = |\widetilde{Q}_n|$ (Fact 2.155). It is believed that the QRP is as difficult as the problem of factoring integers, although no proof of this is known.

3.5 Computing square roots in $\mathbb{Z}_n$

The operations of squaring modulo an integer n and extracting square roots modulo an integer n are frequently used in cryptographic functions. The operation of computing square roots modulo n can be performed efficiently when n is a prime, but is difficult when n is a composite integer whose prime factors are unknown.

3.5.1 Case (i): n prime

Recall from Remark 3.32 that if p is a prime, then it is easy to decide if $a \in \mathbb{Z}_p^*$ is a quadratic residue modulo p. If a is, in fact, a quadratic residue modulo p, then the two square roots of a can be efficiently computed, as demonstrated by Algorithm 3.34.

3.34 Algorithm Finding square roots modulo a prime p

INPUT: an odd prime p and an integer a, $1 \le a \le p-1$.
OUTPUT: the two square roots of a modulo p, provided a is a quadratic residue modulo p.

1. Compute the Legendre symbol $\left(\frac{a}{p}\right)$ using Algorithm 2.149. If $\left(\frac{a}{p}\right) = -1$ then return(a does not have a square root modulo p) and terminate.
2. Select integers b, $1 \le b \le p-1$, at random until one is found with $\left(\frac{b}{p}\right) = -1$. ($b$ is a quadratic non-residue modulo p.)
3. By repeated division by 2, write $p - 1 = 2^s t$, where t is odd.
4. Compute $a^{-1} \bmod p$ by the extended Euclidean algorithm (Algorithm 2.142).
5. Set $c \leftarrow b^t \bmod p$ and $r \leftarrow a^{(t+1)/2} \bmod p$ (Algorithm 2.143).
6. For i from 1 to $s-1$ do the following:
   - 6.1 Compute $d = (r^2 \cdot a^{-1})^{2^{s-i-1}} \bmod p$.
   - 6.2 If $d \equiv -1 \pmod p$ then set $r \leftarrow r \cdot c \bmod p$.
   - 6.3 Set $c \leftarrow c^2 \bmod p$.
7. Return(r, $-r$).

Algorithm 3.34 is a randomized algorithm because of the manner in which the quadratic non-residue b is selected in step 2. No deterministic polynomial-time algorithm for finding a quadratic non-residue modulo a prime p is known (see Remark 2.151).

3.35 Fact Algorithm 3.34 has an expected running time of $O((\lg p)^4)$ bit operations.

This running time is obtained by observing that the dominant step (step 6) is executed $s-1$ times, each iteration involving a modular exponentiation and thus taking $O((\lg p)^3)$ bit operations (Table 2.5). Since in the worst case $s = O(\lg p)$, the running time of $O((\lg p)^4)$ follows. When s is small, the loop in step 6 is executed only a small number of times, and the running time of Algorithm 3.34 is $O((\lg p)^3)$ bit operations. This point is demonstrated next for the special cases $s = 1$ and $s = 2$.

Specializing Algorithm 3.34 to the case $s = 1$ yields the following simple deterministic algorithm for finding square roots when $p \equiv 3 \pmod 4$.

3.36 Algorithm Finding square roots modulo a prime p where $p \equiv 3 \pmod 4$

INPUT: an odd prime p where $p \equiv 3 \pmod 4$, and a square $a \in Q_p$.
OUTPUT: the two square roots of a modulo p.

1. Compute $r = a^{(p+1)/4} \bmod p$ (Algorithm 2.143).
2. Return(r, $-r$).

Specializing Algorithm 3.34 to the case $s = 2$, and using the fact that 2 is a quadratic non-residue modulo p when $p \equiv 5 \pmod 8$, yields the following simple deterministic algorithm for finding square roots when $p \equiv 5 \pmod 8$.

3.37 Algorithm Finding square roots modulo a prime p where $p \equiv 5 \pmod 8$

INPUT: an odd prime p where $p \equiv 5 \pmod 8$, and a square $a \in Q_p$.
OUTPUT: the two square roots of a modulo p.

1. Compute $d = a^{(p-1)/4} \bmod p$ (Algorithm 2.143).
2. If $d = 1$ then compute $r = a^{(p+3)/8} \bmod p$.
3. If $d = p - 1$ then compute $r = 2a(4a)^{(p-5)/8} \bmod p$.
4. Return(r, $-r$).

3.38 Fact Algorithms 3.36 and 3.37 have running times of $O((\lg p)^3)$ bit operations.

Algorithm 3.39 for finding square roots modulo p is preferable to Algorithm 3.34 when $p - 1 = 2^s t$ with s large.

3.39 Algorithm Finding square roots modulo a prime p

INPUT: an odd prime p and a square $a \in Q_p$.
OUTPUT: the two square roots of a modulo p.

1. Choose random $b \in \mathbb{Z}_p$ until $b^2 - 4a$ is a quadratic non-residue modulo p, i.e., $\left(\frac{b^2-4a}{p}\right) = -1$.
2. Let f be the polynomial $x^2 - bx + a$ in $\mathbb{Z}_p[x]$.
3. Compute $r = x^{(p+1)/2} \bmod f$ using Algorithm 2.227. (Note: r will be an integer.)
4. Return(r, $-r$).

3.40 Fact Algorithm 3.39 has an expected running time of $O((\lg p)^3)$ bit operations.

3.41 Note (*computing square roots in a finite field*) Algorithms 3.34, 3.36, 3.37, and 3.39 can be extended in a straightforward manner to find square roots in any finite field $\mathbb{F}_q$ of odd order $q = p^m$, p prime, $m \geq 1$. Square roots in finite fields of even order can also be computed efficiently via Fact 3.42.

3.42 Fact Each element $a \in \mathbb{F}_{2^m}$ has exactly one square root, namely $a^{2^{m-1}}$.

3.5.2 Case (ii): n composite

The discussion in this subsection is restricted to the case of computing square roots modulo n, where n is a product of two distinct odd primes p and q. However, all facts presented here generalize to the case where n is an arbitrary composite integer.

Unlike the case where n is a prime, the problem of deciding whether a given $a \in \mathbb{Z}_n^*$ is a quadratic residue modulo a composite integer n, is believed to be a difficult problem. Certainly, if the Jacobi symbol $\left(\frac{a}{n}\right) = -1$, then a is a quadratic non-residue. On the other hand, if $\left(\frac{a}{n}\right) = 1$, then deciding whether or not a is a quadratic residue is precisely the quadratic residuosity problem, considered in §3.4.

3.43 Definition The *square root modulo n problem* (SQROOT) is the following: given a composite integer n and a quadratic residue a modulo n (i.e. $a \in Q_n$), find a square root of a modulo n.

If the factors p and q of n are known, then the SQROOT problem can be solved efficiently by first finding square roots of a modulo p and modulo q, and then combining them using the Chinese remainder theorem (Fact 2.120) to obtain the square roots of a modulo n. The steps are summarized in Algorithm 3.44, which, in fact, finds all of the four square roots of a modulo n.

3.44 Algorithm Finding square roots modulo n given its prime factors p and q

INPUT: an integer n, its prime factors p and q, and $a \in Q_n$.
OUTPUT: the four square roots of a modulo n.

1. Use Algorithm 3.39 (or Algorithm 3.36 or 3.37, if applicable) to find the two square roots r and $-r$ of a modulo p.
2. Use Algorithm 3.39 (or Algorithm 3.36 or 3.37, if applicable) to find the two square roots s and $-s$ of a modulo q.
3. Use the extended Euclidean algorithm (Algorithm 2.107) to find integers c and d such that $cp + dq = 1$.
4. Set $x \leftarrow (rdq + scp) \bmod n$ and $y \leftarrow (rdq - scp) \bmod n$.
5. Return($\pm x \bmod n$, $\pm y \bmod n$).

3.45 Fact Algorithm 3.44 has an expected running time of $O((\lg p)^3)$ bit operations.

Algorithm 3.44 shows that if one can factor n, then the SQROOT problem is easy. More precisely, SQROOT $\leq_P$ FACTORING. The converse of this statement is also true, as stated in Fact 3.46.

3.46 Fact FACTORING $\leq_P$ SQROOT. That is, the FACTORING problem polytime reduces to the SQROOT problem. Hence, since SQROOT $\leq_P$ FACTORING, the FACTORING and SQROOT problems are computationally equivalent.

Justification. Suppose that one has a polynomial-time algorithm A for solving the SQROOT problem. This algorithm can then be used to factor a given composite integer n as follows. Select an integer x at random with $\gcd(x, n) = 1$, and compute $a = x^2 \bmod n$. Next, algorithm A is run with inputs a and n, and a square root y of a modulo n is returned. If $y \equiv \pm x \pmod{n}$, then the trial fails, and the above procedure is repeated with a new x chosen at random. Otherwise, if $y \not\equiv \pm x \pmod{n}$, then $\gcd(x - y, n)$ is guaranteed to be a non-trivial factor of n (Fact 3.18), namely, p or q. Since a has four square roots modulo n ($\pm x$ and $\pm z$ with $\pm z \not\equiv \pm x \pmod{n}$), the probability of success for each attempt is $\frac{1}{2}$. Hence, the expected number of attempts before a factor of n is obtained is two, and consequently the procedure runs in expected polynomial time. □

3.47 Note (*strengthening of Fact 3.46*) The proof of Fact 3.46 can be easily modified to establish the following stronger result. Let $c \geq 1$ be any constant. If there is an algorithm A which, given n, can find a square root modulo n in polynomial time for a $\frac{1}{(\lg n)^c}$ fraction of all quadratic residues $a \in Q_n$, then the algorithm A can be used to factor n in expected polynomial time. The implication of this statement is that if the problem of factoring n is difficult, then for *almost all* $a \in Q_n$ it is difficult to find square roots modulo n.

The computational equivalence of the SQROOT and FACTORING problems was the basis of the first "provably secure" public-key encryption and signature schemes, presented in §8.3.

3.6 The discrete logarithm problem

The security of many cryptographic techniques depends on the intractability of the discrete logarithm problem. A partial list of these includes Diffie-Hellman key agreement and its derivatives (§12.6), ElGamal encryption (§8.4), and the ElGamal signature scheme and its variants (§11.5). This section summarizes the current knowledge regarding algorithms for solving the discrete logarithm problem.

Unless otherwise specified, algorithms in this section are described in the general setting of a (multiplicatively written) finite cyclic group G of order n with generator α (see Definition 2.167). For a more concrete approach, the reader may find it convenient to think of G as the multiplicative group $\mathbb{Z}_p^*$ of order $p-1$, where the group operation is simply multiplication modulo p.

3.48 Definition Let G be a finite cyclic group of order n. Let α be a generator of G, and let $\beta \in G$. The *discrete logarithm of β to the base α*, denoted $\log_\alpha \beta$, is the unique integer x, $0 \le x \le n-1$, such that $\beta = \alpha^x$.

3.49 Example Let $p = 97$. Then $\mathbb{Z}_{97}^*$ is a cyclic group of order $n = 96$. A generator of $\mathbb{Z}_{97}^*$ is $\alpha = 5$. Since $5^{32} \equiv 35 \pmod{97}$, $\log_5 35 = 32$ in $\mathbb{Z}_{97}^*$. □

The following are some elementary facts about logarithms.

3.50 Fact Let α be a generator of a cyclic group G of order n, and let $\beta, \gamma \in G$. Let s be an integer. Then $\log_\alpha(\beta\gamma) = (\log_\alpha \beta + \log_\alpha \gamma) \bmod n$ and $\log_\alpha(\beta^s) = s \log_\alpha \beta \bmod n$.

The groups of most interest in cryptography are the multiplicative group $\mathbb{F}_q^*$ of the finite field $\mathbb{F}_q$ (§2.6), including the particular cases of the multiplicative group $\mathbb{Z}_p^*$ of the integers modulo a prime p, and the multiplicative group $\mathbb{F}_{2^m}^*$ of the finite field $\mathbb{F}_{2^m}$ of characteristic two. Also of interest are the group of units $\mathbb{Z}_n^*$ where n is a composite integer, the group of points on an elliptic curve defined over a finite field, and the jacobian of a hyperelliptic curve defined over a finite field.

3.51 Definition The *discrete logarithm problem* (DLP) is the following: given a prime p, a generator α of $\mathbb{Z}_p^*$, and an element $\beta \in \mathbb{Z}_p^*$, find the integer x, $0 \le x \le p-2$, such that $\alpha^x \equiv \beta \pmod{p}$.

3.52 Definition The *generalized discrete logarithm problem* (GDLP) is the following: given a finite cyclic group G of order n, a generator α of G, and an element $\beta \in G$, find the integer x, $0 \le x \le n-1$, such that $\alpha^x = \beta$.

The discrete logarithm problem in elliptic curve groups and in the jacobians of hyperelliptic curves are not explicitly considered in this section. The discrete logarithm problem in $\mathbb{Z}_n^*$ is discussed further in §3.8.

3.53 Note (*difficulty of the GDLP is independent of generator*) Let α and γ be two generators of a cyclic group G of order n, and let $\beta \in G$. Let $x = \log_\alpha \beta$, $y = \log_\gamma \beta$, and $z = \log_\alpha \gamma$. Then $\alpha^x = \beta = \gamma^y = (\alpha^z)^y$. Consequently $x = zy \bmod n$, and

$$\log_\gamma \beta = (\log_\alpha \beta)\,(\log_\alpha \gamma)^{-1} \bmod n.$$

This means that any algorithm which computes logarithms to the base α can be used to compute logarithms to any other base γ that is also a generator of G.

3.54 Note (*generalization of GDLP*) A more general formulation of the GDLP is the following: given a finite group G and elements $\alpha, \beta \in G$, find an integer x such that $\alpha^x = \beta$, provided that such an integer exists. In this formulation, it is not required that G be a cyclic group, and, even if it is, it is not required that α be a generator of G. This problem may be harder to solve, in general, than GDLP. However, in the case where G is a cyclic group (for example if G is the multiplicative group of a finite field) and the order of α is known, it can be easily recognized whether an integer x satisfying $\alpha^x = \beta$ exists. This is because of the following fact: if G is a cyclic group, α is an element of order n in G, and $\beta \in G$, then there exists an integer x such that $\alpha^x = \beta$ if and only if $\beta^n = 1$.

3.55 Note (*solving the DLP in a cyclic group G of order n is in essence computing an isomorphism between G and $\mathbb{Z}_n$*) Even though any two cyclic groups of the same order are *isomorphic* (that is, they have the same structure although the elements may be written in different representations), an efficient algorithm for computing logarithms in one group does not necessarily imply an efficient algorithm for the other group. To see this, consider that every cyclic group of order n is isomorphic to the additive cyclic group $\mathbb{Z}_n$, i.e., the set of integers $\{0, 1, 2, \ldots, n-1\}$ where the group operation is addition modulo n. Moreover, the discrete logarithm problem in the latter group, namely, the problem of finding an integer x such that $ax \equiv b \pmod{n}$ given $a, b \in \mathbb{Z}_n$, is easy as shown in the following. First note that there does not exist a solution x if $d = \gcd(a, n)$ does not divide b (Fact 2.119). Otherwise, if d divides b, the extended Euclidean algorithm (Algorithm 2.107) can be used to find integers s and t such that $as + nt = d$. Multiplying both sides of this equation by the integer b/d gives $a(sb/d) + n(tb/d) = b$. Reducing this equation modulo n yields $a(sb/d) \equiv b \pmod{n}$ and hence $x = (sb/d) \bmod n$ is the desired (and easily obtainable) solution.

The known algorithms for the DLP can be categorized as follows:

1. algorithms which work in arbitrary groups, e.g., exhaustive search (§3.6.1), the baby-step giant-step algorithm (§3.6.2), Pollard's rho algorithm (§3.6.3);
2. algorithms which work in arbitrary groups but are especially efficient if the order of the group has only small prime factors, e.g., Pohlig-Hellman algorithm (§3.6.4); and
3. the index-calculus algorithms (§3.6.5) which are efficient only in certain groups.

3.6.1 Exhaustive search

The most obvious algorithm for GDLP (Definition 3.52) is to successively compute α^0, α^1, $\alpha^2, \ldots$ until β is obtained. This method takes $O(n)$ multiplications, where n is the order of α, and is therefore inefficient if n is large (i.e. in cases of cryptographic interest).

3.6.2 Baby-step giant-step algorithm

Let $m = \lceil \sqrt{n} \rceil$, where n is the order of α. The baby-step giant-step algorithm is a time-memory trade-off of the method of exhaustive search and is based on the following observation. If $\beta = \alpha^x$, then one can write $x = im + j$, where $0 \le i, j < m$. Hence, $\alpha^x = \alpha^{im}\alpha^j$, which implies $\beta(\alpha^{-m})^i = \alpha^j$. This suggests the following algorithm for computing x.

3.56 Algorithm Baby-step giant-step algorithm for computing discrete logarithms

INPUT: a generator α of a cyclic group G of order n, and an element $\beta \in G$.
OUTPUT: the discrete logarithm $x = \log_\alpha \beta$.

1. Set $m \leftarrow \lceil \sqrt{n} \rceil$.
2. Construct a table with entries (j, α^j) for $0 \le j < m$. Sort this table by second component. (Alternatively, use conventional hashing on the second component to store the entries in a hash table; placing an entry, and searching for an entry in the table takes constant time.)
3. Compute α^{-m} and set $\gamma \leftarrow \beta$.
4. For i from 0 to $m-1$ do the following:
   4.1 Check if γ is the second component of some entry in the table.
   4.2 If $\gamma = \alpha^j$ then return($x = im + j$).
   4.3 Set $\gamma \leftarrow \gamma \cdot \alpha^{-m}$.

Algorithm 3.56 requires storage for $O(\sqrt{n})$ group elements. The table takes $O(\sqrt{n})$ multiplications to construct, and $O(\sqrt{n} \lg n)$ comparisons to sort. Having constructed this table, step 4 takes $O(\sqrt{n})$ multiplications and $O(\sqrt{n})$ table look-ups. Under the assumption that a group multiplication takes more time than $\lg n$ comparisons, the running time of Algorithm 3.56 can be stated more concisely as follows.

3.57 Fact The running time of the baby-step giant-step algorithm (Algorithm 3.56) is $O(\sqrt{n})$ group multiplications.

3.58 Example (*baby-step giant-step algorithm for logarithms in* $\mathbb{Z}_{113}^*$) Let $p = 113$. The element $\alpha = 3$ is a generator of $\mathbb{Z}_{113}^*$ of order $n = 112$. Consider $\beta = 57$. Then $\log_3 57$ is computed as follows.

1. Set $m \leftarrow \lceil \sqrt{112} \rceil = 11$.
2. Construct a table whose entries are $(j, \alpha^j \bmod p)$ for $0 \le j < 11$:

j	0	1	2	3	4	5	6	7	8	9	10
$3^j \bmod 113$	1	3	9	27	81	17	51	40	7	21	63

and sort the table by second component:

j	0	1	8	2	5	9	3	7	6	10	4
$3^j \bmod 113$	1	3	7	9	17	21	27	40	51	63	81

3. Using Algorithm 2.142, compute $\alpha^{-1} = 3^{-1} \bmod 113 = 38$ and then compute $\alpha^{-m} = 38^{11} \bmod 113 = 58$.
4. Next, $\gamma = \beta\alpha^{-mi} \bmod 113$ for $i = 0, 1, 2, \ldots$ is computed until a value in the second row of the table is obtained. This yields:

i	0	1	2	3	4	5	6	7	8	9
$\gamma = 57 \cdot 58^i \bmod 113$	57	29	100	37	112	55	26	39	2	3

Finally, since $\beta\alpha^{-9m} = 3 = \alpha^1$, $\beta = \alpha^{100}$ and, therefore, $\log_3 57 = 100$. □

3.59 Note (*restricted exponents*) In order to improve performance, some cryptographic protocols which use exponentiation in $\mathbb{Z}_p^*$ select exponents of a special form, e.g. having small Hamming weight. (The *Hamming weight* of an integer is the number of ones in its binary representation.) Suppose that p is a k-bit prime, and only exponents of Hamming weight t are used. The number of such exponents is $\binom{k}{t}$. Algorithm 3.56 can be modified to search the exponent space in roughly $\binom{k}{t/2}$ steps. The algorithm also applies to exponents that are restricted in certain other ways, and extends to all finite groups.

3.6.3 Pollard's rho algorithm for logarithms

Pollard's rho algorithm (Algorithm 3.60) for computing discrete logarithms is a randomized algorithm with the same expected running time as the baby-step giant-step algorithm (Algorithm 3.56), but which requires a negligible amount of storage. For this reason, it is far preferable to Algorithm 3.56 for problems of practical interest. For simplicity, it is assumed in this subsection that G is a cyclic group whose order n is prime.

The group G is partitioned into three sets S_1, S_2, and S_3 of roughly equal size based on some easily testable property. Some care must be exercised in selecting the partition; for example, $1 \notin S_2$. Define a sequence of group elements $x_0, x_1, x_2, \ldots$ by $x_0 = 1$ and

$$x_{i+1} \;=\; f(x_i) \stackrel{\text{def}}{=} \begin{cases} \beta \cdot x_i, & \text{if } x_i \in S_1, \\ x_i^2, & \text{if } x_i \in S_2, \\ \alpha \cdot x_i, & \text{if } x_i \in S_3, \end{cases} \tag{3.2}$$

for $i \geq 0$. This sequence of group elements in turn defines two sequences of integers $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ satisfying $x_i = \alpha^{a_i}\beta^{b_i}$ for $i \geq 0$: $a_0 = 0$, $b_0 = 0$, and for $i \geq 0$,

$$a_{i+1} = \begin{cases} a_i, & \text{if } x_i \in S_1, \\ 2a_i \bmod n, & \text{if } x_i \in S_2, \\ a_i + 1 \bmod n, & \text{if } x_i \in S_3, \end{cases} \tag{3.3}$$

and

$$b_{i+1} = \begin{cases} b_i + 1 \bmod n, & \text{if } x_i \in S_1, \\ 2b_i \bmod n, & \text{if } x_i \in S_2, \\ b_i, & \text{if } x_i \in S_3. \end{cases} \tag{3.4}$$

Floyd's cycle-finding algorithm (Note 3.8) can then be utilized to find two group elements x_i and x_{2i} such that $x_i = x_{2i}$. Hence $\alpha^{a_i}\beta^{b_i} = \alpha^{a_{2i}}\beta^{b_{2i}}$, and so $\beta^{b_i - b_{2i}} = \alpha^{a_{2i} - a_i}$. Taking logarithms to the base α of both sides of this last equation yields

$$(b_i - b_{2i}) \cdot \log_\alpha \beta \equiv (a_{2i} - a_i) \pmod{n}.$$

Provided $b_i \not\equiv b_{2i} \pmod{n}$ (note: $b_i \equiv b_{2i}$ occurs with negligible probability), this equation can then be efficiently solved to determine $\log_\alpha \beta$.

3.60 Algorithm Pollard's rho algorithm for computing discrete logarithms

INPUT: a generator α of a cyclic group G of prime order n, and an element $\beta \in G$.
OUTPUT: the discrete logarithm $x = \log_\alpha \beta$.

1. Set $x_0 \leftarrow 1$, $a_0 \leftarrow 0$, $b_0 \leftarrow 0$.
2. For $i = 1, 2, \ldots$ do the following:
   - 2.1 Using the quantities $x_{i-1}, a_{i-1}, b_{i-1}$, and $x_{2i-2}, a_{2i-2}, b_{2i-2}$ computed previously, compute x_i, a_i, b_i and x_{2i}, a_{2i}, b_{2i} using equations (3.2), (3.3), and (3.4).
   - 2.2 If $x_i = x_{2i}$, then do the following:
     Set $r \leftarrow b_i - b_{2i} \bmod n$.
     If $r = 0$ then terminate the algorithm with failure; otherwise, compute $x = r^{-1}(a_{2i} - a_i) \bmod n$ and return(x).

In the rare case that Algorithm 3.60 terminates with failure, the procedure can be repeated by selecting random integers a_0, b_0 in the interval $[1, n-1]$, and starting with $x_0 = \alpha^{a_0}\beta^{b_0}$. Example 3.61 with artificially small parameters illustrates Pollard's rho algorithm.

3.61 Example (*Pollard's rho algorithm for logarithms in a subgroup of* $\mathbb{Z}^*_{383}$) The element $\alpha = 2$ is a generator of the subgroup G of $\mathbb{Z}^*_{383}$ of order $n = 191$. Suppose $\beta = 228$. Partition the elements of G into three subsets according to the rule $x \in S_1$ if $x \equiv 1 \pmod 3$, $x \in S_2$ if $x \equiv 0 \pmod 3$, and $x \in S_3$ if $x \equiv 2 \pmod 3$. Table 3.2 shows the values of x_i, a_i, b_i, x_{2i}, a_{2i}, and b_{2i} at the end of each iteration of step 2 of Algorithm 3.60. Note that $x_{14} = x_{28} = 144$. Finally, compute $r = b_{14} - b_{28} \bmod 191 = 125$, $r^{-1} = 125^{-1} \bmod 191 = 136$, and $r^{-1}(a_{28} - a_{14}) \bmod 191 = 110$. Hence, $\log_2 228 = 110$. □

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	228	0	1	279	0	2
2	279	0	2	184	1	4
3	92	0	4	14	1	6
4	184	1	4	256	2	7
5	205	1	5	304	3	8
6	14	1	6	121	6	18
7	28	2	6	144	12	38
8	256	2	7	235	48	152
9	152	2	8	72	48	154
10	304	3	8	14	96	118
11	372	3	9	256	97	119
12	121	6	18	304	98	120
13	12	6	19	121	5	51
14	144	12	38	144	10	104

Table 3.2: *Intermediate steps of Pollard's rho algorithm in Example 3.61.*

3.62 Fact Let G be a group of order n, a prime. Assume that the function $f : G \longrightarrow G$ defined by equation (3.2) behaves like a random function. Then the expected running time of Pollard's rho algorithm for discrete logarithms in G is $O(\sqrt{n})$ group operations. Moreover, the algorithm requires negligible storage.

3.6.4 Pohlig-Hellman algorithm

Algorithm 3.63 for computing logarithms takes advantage of the factorization of the order n of the group G. Let $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ be the prime factorization of n. If $x = \log_\alpha \beta$, then the approach is to determine $x_i = x \bmod p_i^{e_i}$ for $1 \le i \le r$, and then use Gauss's algorithm (Algorithm 2.121) to recover $x \bmod n$. Each integer x_i is determined by computing the digits $l_0, l_1, \ldots, l_{e_i-1}$ in turn of its p_i-ary representation: $x_i = l_0 + l_1 p_i + \cdots + l_{e_i-1} p_i^{e_i-1}$, where $0 \le l_j \le p_i - 1$.

To see that the output of Algorithm 3.63 is correct, observe first that in step 2.3 the order of $\overline{\alpha}$ is q. Next, at iteration j of step 2.4, $\gamma = \alpha^{l_0 + l_1 q + \cdots + l_{j-1} q^{j-1}}$. Hence,

$$\begin{aligned}
\overline{\beta} &= (\beta/\gamma)^{n/q^{j+1}} = (\alpha^{x - l_0 - l_1 q - \cdots - l_{j-1} q^{j-1}})^{n/q^{j+1}} \\
&= (\alpha^{n/q^{j+1}})^{x_i - l_0 - l_1 q - \cdots - l_{j-1} q^{j-1}} \\
&= (\alpha^{n/q^{j+1}})^{l_j q^j + \cdots + l_{e-1} q^{e-1}} \\
&= (\alpha^{n/q})^{l_j + \cdots + l_{e-1} q^{e-1-j}} = (\overline{\alpha})^{l_j},
\end{aligned}$$

the last equality being true because $\overline{\alpha}$ has order q. Hence, $\log_{\overline{\alpha}} \overline{\beta}$ is indeed equal to l_j.

3.63 Algorithm Pohlig-Hellman algorithm for computing discrete logarithms

INPUT: a generator α of a cyclic group G of order n, and an element $\beta \in G$.
OUTPUT: the discrete logarithm $x = \log_\alpha \beta$.

1. Find the prime factorization of n: $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, where $e_i \geq 1$.
2. For i from 1 to r do the following:
   (*Compute* $x_i = l_0 + l_1 p_i + \cdots + l_{e_i - 1} p_i^{e_i - 1}$, *where* $x_i = x \bmod p_i^{e_i}$)
   2.1 (*Simplify the notation*) Set $q \leftarrow p_i$ and $e \leftarrow e_i$.
   2.2 Set $\gamma \leftarrow 1$ and $l_{-1} \leftarrow 0$.
   2.3 Compute $\overline{\alpha} \leftarrow \alpha^{n/q}$.
   2.4 (*Compute the* l_j) For j from 0 to $e-1$ do the following:
   Compute $\gamma \leftarrow \gamma \alpha^{l_{j-1} q^{j-1}}$ and $\overline{\beta} \leftarrow (\beta \gamma^{-1})^{n/q^{j+1}}$.
   Compute $l_j \leftarrow \log_{\overline{\alpha}} \overline{\beta}$ (e.g., using Algorithm 3.56; see Note 3.67(iii)).
   2.5 Set $x_i \leftarrow l_0 + l_1 q + \cdots + l_{e-1} q^{e-1}$.
3. Use Gauss's algorithm (Algorithm 2.121) to compute the integer x, $0 \leq x \leq n-1$, such that $x \equiv x_i \pmod{p_i^{e_i}}$ for $1 \leq i \leq r$.
4. Return(x).

Example 3.64 illustrates Algorithm 3.63 with artificially small parameters.

3.64 Example (*Pohlig-Hellman algorithm for logarithms in* $\mathbb{Z}_{251}^*$) Let $p = 251$. The element $\alpha = 71$ is a generator of $\mathbb{Z}_{251}^*$ of order $n = 250$. Consider $\beta = 210$. Then $x = \log_{71} 210$ is computed as follows.

1. The prime factorization of n is $250 = 2 \cdot 5^3$.
2. (a) (Compute $x_1 = x \bmod 2$)
   Compute $\overline{\alpha} = \alpha^{n/2} \bmod p = 250$ and $\overline{\beta} = \beta^{n/2} \bmod p = 250$. Then $x_1 = \log_{250} 250 = 1$.
   (b) (Compute $x_2 = x \bmod 5^3 = l_0 + l_1 5 + l_2 5^2$)
   i. Compute $\overline{\alpha} = \alpha^{n/5} \bmod p = 20$.
   ii. Compute $\gamma = 1$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/5} \bmod p = 149$. Using exhaustive search,[5] compute $l_0 = \log_{20} 149 = 2$.
   iii. Compute $\gamma = \gamma\alpha^2 \bmod p = 21$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/25} \bmod p = 113$. Using exhaustive search, compute $l_1 = \log_{20} 113 = 4$.
   iv. Compute $\gamma = \gamma\alpha^{4 \cdot 5} \bmod p = 115$ and $\overline{\beta} = (\beta\gamma^{-1})^{(p-1)/125} \bmod p = 149$. Using exhaustive search, compute $l_2 = \log_{20} 149 = 2$.

   Hence, $x_2 = 2 + 4 \cdot 5 + 2 \cdot 5^2 = 72$.
3. Finally, solve the pair of congruences $x \equiv 1 \pmod 2$, $x \equiv 72 \pmod{125}$ to get $x = \log_{71} 210 = 197$. □

3.65 Fact Given the factorization of n, the running time of the Pohlig-Hellman algorithm (Algorithm 3.63) is $O(\sum_{i=1}^r e_i(\lg n + \sqrt{p_i}))$ group multiplications.

3.66 Note (*effectiveness of Pohlig-Hellman*) Fact 3.65 implies that the Pohlig-Hellman algorithm is efficient only if each prime divisor p_i of n is relatively small; that is, if n is a smooth

[5] Exhaustive search is preferable to Algorithm 3.56 when the group is very small (here the order of $\overline{\alpha}$ is 5).

integer (Definition 3.13). An example of a group in which the Pohlig-Hellman algorithm is effective follows. Consider the multiplicative group $\mathbb{Z}_p^*$ where p is the 107-digit prime:

$$p = 227088231986781039743145181950291021585250524967592855\\ 96453269189798311427475159776411276642277139650833937.$$

The order of $\mathbb{Z}_p^*$ is $n = p - 1 = 2^4 \cdot 104729^8 \cdot 224737^8 \cdot 350377^4$. Since the largest prime divisor of $p-1$ is only 350377, it is relatively easy to compute logarithms in this group using the Pohlig-Hellman algorithm.

3.67 Note (*miscellaneous*)

(i) If n is a prime, then Algorithm 3.63 (Pohlig-Hellman) is the same as baby-step giant-step (Algorithm 3.56).
(ii) In step 1 of Algorithm 3.63, a factoring algorithm which finds small factors first (e.g., Algorithm 3.9) should be employed; if the order n is not a smooth integer, then Algorithm 3.63 is inefficient anyway.
(iii) The storage required for Algorithm 3.56 in step 2.4 can be eliminated by using instead Pollard's rho algorithm (Algorithm 3.60).

3.6.5 Index-calculus algorithm

The index-calculus algorithm is the most powerful method known for computing discrete logarithms. The technique employed does not apply to all groups, but when it does, it often gives a subexponential-time algorithm. The algorithm is first described in the general setting of a cyclic group G (Algorithm 3.68). Two examples are then presented to illustrate how the index-calculus algorithm works in two kinds of groups that are used in practical applications, namely $\mathbb{Z}_p^*$ (Example 3.69) and $\mathbb{F}_{2^m}^*$ (Example 3.70).

The index-calculus algorithm requires the selection of a relatively small subset S of elements of G, called the *factor base*, in such a way that a significant fraction of elements of G can be efficiently expressed as products of elements from S. Algorithm 3.68 proceeds to precompute a database containing the logarithms of all the elements in S, and then reuses this database each time the logarithm of a particular group element is required.

The description of Algorithm 3.68 is incomplete for two reasons. Firstly, a technique for selecting the factor base S is not specified. Secondly, a method for efficiently generating relations of the form (3.5) and (3.7) is not specified. The factor base S must be a subset of G that is small (so that the system of equations to be solved in step 3 is not too large), but not too small (so that the expected number of trials to generate a relation (3.5) or (3.7) is not too large). Suitable factor bases and techniques for generating relations are known for some cyclic groups including $\mathbb{Z}_p^*$ (see §3.6.5(i)) and $\mathbb{F}_{2^m}^*$ (see §3.6.5(ii)), and, moreover, the multiplicative group $\mathbb{F}_q^*$ of a general finite field $\mathbb{F}_q$.

3.68 Algorithm Index-calculus algorithm for discrete logarithms in cyclic groups

INPUT: a generator α of a cyclic group G of order n, and an element $\beta \in G$.
OUTPUT: the discrete logarithm $y = \log_\alpha \beta$.

1. (*Select a factor base S*) Choose a subset $S = \{p_1, p_2, \ldots, p_t\}$ of G such that a "significant proportion" of all elements in G can be efficiently expressed as a product of elements from S.
2. (*Collect linear relations involving logarithms of elements in S*)

2.1 Select a random integer k, $0 \le k \le n-1$, and compute α^k.
2.2 Try to write α^k as a product of elements in S:

$$\alpha^k = \prod_{i=1}^{t} p_i^{c_i}, \quad c_i \ge 0. \tag{3.5}$$

If successful, take logarithms of both sides of equation (3.5) to obtain a linear relation

$$k \equiv \sum_{i=1}^{t} c_i \log_\alpha p_i \pmod{n}. \tag{3.6}$$

2.3 Repeat steps 2.1 and 2.2 until $t+c$ relations of the form (3.6) are obtained (c is a small positive integer, e.g. $c = 10$, such that the system of equations given by the $t+c$ relations has a unique solution with high probability).

3. (*Find the logarithms of elements in* S) Working modulo n, solve the linear system of $t+c$ equations (in t unknowns) of the form (3.6) collected in step 2 to obtain the values of $\log_\alpha p_i$, $1 \le i \le t$.
4. (*Compute* y)

4.1 Select a random integer k, $0 \le k \le n-1$, and compute $\beta \cdot \alpha^k$.
4.2 Try to write $\beta \cdot \alpha^k$ as a product of elements in S:

$$\beta \cdot \alpha^k = \prod_{i=1}^{t} p_i^{d_i}, \quad d_i \ge 0. \tag{3.7}$$

If the attempt is unsuccessful then repeat step 4.1. Otherwise, taking logarithms of both sides of equation (3.7) yields $\log_\alpha \beta = (\sum_{i=1}^{t} d_i \log_\alpha p_i - k) \bmod n$; thus, compute $y = (\sum_{i=1}^{t} d_i \log_\alpha p_i - k) \bmod n$ and return(y).

(i) Index-calculus algorithm in $\mathbb{Z}_p^*$

For the field $\mathbb{Z}_p$, p a prime, the factor base S can be chosen as the first t prime numbers. A relation (3.5) is generated by computing $\alpha^k \bmod p$ and then using trial division to check whether this integer is a product of primes in S. Example 3.69 illustrates Algorithm 3.68 in $\mathbb{Z}_p^*$ on a problem with artificially small parameters.

3.69 Example (*Algorithm 3.68 for logarithms in* $\mathbb{Z}_{229}^*$) Let $p = 229$. The element $\alpha = 6$ is a generator of $\mathbb{Z}_{229}^*$ of order $n = 228$. Consider $\beta = 13$. Then $\log_6 13$ is computed as follows, using the index-calculus technique.

1. The factor base is chosen to be the first 5 primes: $S = \{2, 3, 5, 7, 11\}$.
2. The following six relations involving elements of the factor base are obtained (unsuccessful attempts are not shown):

$$\begin{aligned}
6^{100} \bmod 229 &= 180 = 2^2 \cdot 3^2 \cdot 5 \\
6^{18} \bmod 229 &= 176 = 2^4 \cdot 11 \\
6^{12} \bmod 229 &= 165 = 3 \cdot 5 \cdot 11 \\
6^{62} \bmod 229 &= 154 = 2 \cdot 7 \cdot 11 \\
6^{143} \bmod 229 &= 198 = 2 \cdot 3^2 \cdot 11 \\
6^{206} \bmod 229 &= 210 = 2 \cdot 3 \cdot 5 \cdot 7.
\end{aligned}$$

These relations yield the following six equations involving the logarithms of elements in the factor base:

$$\begin{aligned}
100 &\equiv 2\log_6 2 + 2\log_6 3 + \log_6 5 \pmod{228}\\
18 &\equiv 4\log_6 2 + \log_6 11 \pmod{228}\\
12 &\equiv \log_6 3 + \log_6 5 + \log_6 11 \pmod{228}\\
62 &\equiv \log_6 2 + \log_6 7 + \log_6 11 \pmod{228}\\
143 &\equiv \log_6 2 + 2\log_6 3 + \log_6 11 \pmod{228}\\
206 &\equiv \log_6 2 + \log_6 3 + \log_6 5 + \log_6 7 \pmod{228}.
\end{aligned}$$

3. Solving the linear system of six equations in five unknowns (the logarithms $x_i = \log_6 p_i$) yields the solutions $\log_6 2 = 21$, $\log_6 3 = 208$, $\log_6 5 = 98$, $\log_6 7 = 107$, and $\log_6 11 = 162$.
4. Suppose that the integer $k = 77$ is selected. Since $\beta \cdot \alpha^k = 13 \cdot 6^{77} \bmod 229 = 147 = 3 \cdot 7^2$, it follows that

$$\log_6 13 = (\log_6 3 + 2\log_6 7 - 77) \bmod 228 = 117. \qquad \square$$

(ii) Index-calculus algorithm in $\mathbb{F}_{2^m}^*$

The elements of the finite field $\mathbb{F}_{2^m}$ are represented as polynomials in $\mathbb{Z}_2[x]$ of degree at most $m-1$, where multiplication is performed modulo a fixed irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_2[x]$ (see §2.6). The factor base S can be chosen as the set of all irreducible polynomials in $\mathbb{Z}_2[x]$ of degree at most some prescribed bound b. A relation (3.5) is generated by computing $\alpha^k \bmod f(x)$ and then using trial division to check whether this polynomial is a product of polynomials in S. Example 3.70 illustrates Algorithm 3.68 in $\mathbb{F}_{2^m}^*$ on a problem with artificially small parameters.

3.70 Example (*Algorithm 3.68 for logarithms in* $\mathbb{F}_{2^7}^*$) The polynomial $f(x) = x^7 + x + 1$ is irreducible over $\mathbb{Z}_2$. Hence, the elements of the finite field $\mathbb{F}_{2^7}$ of order 128 can be represented as the set of all polynomials in $\mathbb{Z}_2[x]$ of degree at most 6, where multiplication is performed modulo $f(x)$. The order of $\mathbb{F}_{2^7}^*$ is $n = 2^7 - 1 = 127$, and $\alpha = x$ is a generator of $\mathbb{F}_{2^7}^*$. Suppose $\beta = x^4 + x^3 + x^2 + x + 1$. Then $y = \log_x \beta$ can be computed as follows, using the index-calculus technique.

1. The factor base is chosen to be the set of all irreducible polynomials in $\mathbb{Z}_2[x]$ of degree at most 3: $S = \{x, x+1, x^2+x+1, x^3+x+1, x^3+x^2+1\}$.
2. The following five relations involving elements of the factor base are obtained (unsuccessful attempts are not shown):

$$\begin{aligned}
x^{18} \bmod f(x) &= x^6 + x^4 && = x^4(x+1)^2\\
x^{105} \bmod f(x) &= x^6 + x^5 + x^4 + x && = x(x+1)^2(x^3+x^2+1)\\
x^{72} \bmod f(x) &= x^6 + x^5 + x^3 + x^2 && = x^2(x+1)^2(x^2+x+1)\\
x^{45} \bmod f(x) &= x^5 + x^2 + x + 1 && = (x+1)^2(x^3+x+1)\\
x^{121} \bmod f(x) &= x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 && = (x^3+x+1)(x^3+x^2+1).
\end{aligned}$$

These relations yield the following five equations involving the logarithms of elements in the factor base (for convenience of notation, let $p_1 = \log_x x$, $p_2 = \log_x(x+$

1), $p_3 = \log_x(x^2+x+1)$, $p_4 = \log_x(x^3+x+1)$, and $p_5 = \log_x(x^3+x^2+1)$):

$$\begin{aligned} 18 &\equiv 4p_1 + 2p_2 \pmod{127} \\ 105 &\equiv p_1 + 2p_2 + p_5 \pmod{127} \\ 72 &\equiv 2p_1 + 2p_2 + p_3 \pmod{127} \\ 45 &\equiv 2p_2 + p_4 \pmod{127} \\ 121 &\equiv p_4 + p_5 \pmod{127}. \end{aligned}$$

3. Solving the linear system of five equations in five unknowns yields the values $p_1 = 1$, $p_2 = 7$, $p_3 = 56$, $p_4 = 31$, and $p_5 = 90$.
4. Suppose $k = 66$ is selected. Since

$$\beta\alpha^k = (x^4+x^3+x^2+x+1)x^{66} \bmod f(x) = x^5+x^3+x = x(x^2+x+1)^2,$$

it follows that

$$\log_x(x^4+x^3+x^2+x+1) = (p_1 + 2p_3 - 66) \bmod 127 = 47. \qquad \square$$

3.71 Note (*running time of Algorithm 3.68*) To optimize the running time of the index-calculus algorithm, the size t of the factor base should be judiciously chosen. The optimal selection relies on knowledge concerning the distribution of smooth integers in the interval $[1, p-1]$ for the case of $\mathbb{Z}_p^*$, and for the case of $\mathbb{F}_{2^m}^*$ on the distribution of *smooth polynomials* (that is, polynomials all of whose irreducible factors have relatively small degrees) among polynomials in $\mathbb{F}_2[x]$ of degree less than m. With an optimal choice of t, the index-calculus algorithm as described above for $\mathbb{Z}_p^*$ and $\mathbb{F}_{2^m}^*$ has an expected running time of $L_q[\frac{1}{2}, c]$ where $q = p$ or $q = 2^m$, and $c > 0$ is a constant.

3.72 Note (*fastest algorithms known for discrete logarithms in $\mathbb{Z}_p^*$ and $\mathbb{F}_{2^m}^*$*) Currently, the best algorithm known for computing logarithms in $\mathbb{F}_{2^m}^*$ is a variation of the index-calculus algorithm called *Coppersmith's algorithm*, with an expected running time of $L_{2^m}[\frac{1}{3}, c]$ for some constant $c < 1.587$. The best algorithm known for computing logarithms in $\mathbb{Z}_p^*$ is a variation of the index-calculus algorithm called the *number field sieve*, with an expected running time of $L_p[\frac{1}{3}, 1.923]$. The latest efforts in these directions are surveyed in the Notes section (§3.12).

3.73 Note (*parallelization of the index-calculus algorithm*)

(i) For the optimal choice of parameters, the most time-consuming phase of the index-calculus algorithm is usually the generation of relations involving factor base logarithms (step 2 of Algorithm 3.68). The work for this stage can be easily distributed among a network of processors by simply having the processors search for relations independently of each other. The relations generated are collected by a central processor. When enough relations have been generated, the corresponding system of linear equations can be solved (step 3 of Algorithm 3.68) on a single (possibly parallel) computer.

(ii) The database of factor base logarithms need only be computed once for a given finite field. Relative to this, the computation of individual logarithms (step 4 of Algorithm 3.68) is considerably faster.

3.6.6 Discrete logarithm problem in subgroups of $\mathbb{Z}_p^*$

The discrete logarithm problem in subgroups of $\mathbb{Z}_p^*$ has special interest because its presumed intractability is the basis for the security of the U.S. Government NIST Digital Signature Algorithm (§11.5.1), among other cryptographic techniques.

Let p be a prime and q a prime divisor of $p-1$. Let G be the unique cyclic subgroup of $\mathbb{Z}_p^*$ of order q, and let α be a generator of G. Then the discrete logarithm problem in G is the following: given p, q, α, and $\beta \in G$, find the unique integer x, $0 \leq x \leq q-1$, such that $\alpha^x \equiv \beta \pmod p$. The powerful index-calculus algorithms do not appear to apply directly in G. That is, one needs to apply the index-calculus algorithm in the group $\mathbb{Z}_p^*$ itself in order to compute logarithms in the smaller group G. Consequently, there are two approaches one could take to computing logarithms in G:

1. Use a "square-root" algorithm directly in G, such as Pollard's rho algorithm (Algorithm 3.60). The running time of this approach is $O(\sqrt{q})$.
2. Let γ be a generator of $\mathbb{Z}_p^*$, and let $l = (p-1)/q$. Use an index-calculus algorithm in $\mathbb{Z}_p^*$ to find integers y and z such that $\alpha = \gamma^y$ and $\beta = \gamma^z$. Then $x = \log_\alpha \beta = (z/l)(y/l)^{-1} \bmod q$. (Since y and z are both divisible by l, y/l and z/l are indeed integers.) The running time of this approach is $L_p[\frac{1}{3}, c]$ if the number field sieve is used.

Which of the two approaches is faster depends on the relative size of $\sqrt{q}$ and $L_p[\frac{1}{3}, c]$.

3.7 The Diffie-Hellman problem

The Diffie-Hellman problem is closely related to the well-studied discrete logarithm problem (DLP) of §3.6. It is of significance to public-key cryptography because its apparent intractability forms the basis for the security of many cryptographic schemes including Diffie-Hellman key agreement and its derivatives (§12.6), and ElGamal public-key encryption (§8.4).

3.74 Definition The *Diffie-Hellman problem (DHP)* is the following: given a prime p, a generator α of $\mathbb{Z}_p^*$, and elements $\alpha^a \bmod p$ and $\alpha^b \bmod p$, find $\alpha^{ab} \bmod p$.

3.75 Definition The *generalized Diffie-Hellman problem (GDHP)* is the following: given a finite cyclic group G, a generator α of G, and group elements α^a and α^b, find α^{ab}.

Suppose that the discrete logarithm problem in $\mathbb{Z}_p^*$ could be efficiently solved. Then given α, p, $\alpha^a \bmod p$ and $\alpha^b \bmod p$, one could first find a from α, p, and $\alpha^a \bmod p$ by solving a discrete logarithm problem, and then compute $(\alpha^b)^a = \alpha^{ab} \bmod p$. This establishes the following relation between the Diffie-Hellman problem and the discrete logarithm problem.

3.76 Fact DHP $\leq_P$ DLP. That is, DHP polytime reduces to the DLP. More generally, GDHP $\leq_P$ GDLP.

The question then remains whether the GDLP and GDHP are computationally equivalent. This remains unknown; however, some recent progress in this regard is summarized in Fact 3.77. Recall that ϕ is the Euler phi function (Definition 2.100), and an integer is B-smooth if all its prime factors are $\leq B$ (Definition 3.13).

3.77 Fact (*known equivalences between GDHP and GDLP*)

(i) Let p be a prime where the factorization of $p-1$ is known. Suppose also that $\phi(p-1)$ is B-smooth, where $B = O((\ln p)^c)$ for some constant c. Then the DHP and DLP in $\mathbb{Z}_p^*$ are computationally equivalent.

(ii) More generally, let G be a finite cyclic group of order n where the factorization of n is known. Suppose also that $\phi(n)$ is B-smooth, where $B = O((\ln n)^c)$ for some constant c. Then the GDHP and GDLP in G are computationally equivalent.

(iii) Let G be a finite cyclic group of order n where the factorization of n is known. If for each prime divisor p of n either $p-1$ or $p+1$ is B-smooth, where $B = O((\ln n)^c)$ for some constant c, then the GDHP and GDLP in G are computationally equivalent.

3.8 Composite moduli

The group of units of $\mathbb{Z}_n$, namely $\mathbb{Z}_n^*$, has been proposed for use in several cryptographic mechanisms, including the key agreement protocols of Yacobi and McCurley (see §12.6 notes on page 538) and the identification scheme of Girault (see §10.4 notes on page 423). There are connections of cryptographic interest between the discrete logarithm and Diffie-Hellman problems in (cyclic subgroups of) $\mathbb{Z}_n^*$, and the problem of factoring n. This section summarizes the results known along these lines.

3.78 Fact Let n be a composite integer. If the discrete logarithm problem in $\mathbb{Z}_n^*$ can be solved in polynomial time, then n can be factored in expected polynomial time.

In other words, the discrete logarithm problem in $\mathbb{Z}_n^*$ is at least as difficult as the problem of factoring n. Fact 3.79 is a partial converse to Fact 3.78 and states that the discrete logarithm in $\mathbb{Z}_n^*$ is no harder than the combination of the problems of factoring n and computing discrete logarithms in $\mathbb{Z}_p^*$ for each prime factor p of n.

3.79 Fact Let n be a composite integer. The discrete logarithm problem in $\mathbb{Z}_n^*$ polytime reduces to the combination of the integer factorization problem and the discrete logarithm problem in $\mathbb{Z}_p^*$ for each prime factor p of n.

Fact 3.80 states that the Diffie-Hellman problem in $\mathbb{Z}_n^*$ is at least as difficult as the problem of factoring n.

3.80 Fact Let $n = pq$ where p and q are odd primes. If the Diffie-Hellman problem in $\mathbb{Z}_n^*$ can be solved in polynomial time for a non-negligible proportion of all bases $\alpha \in \mathbb{Z}_n^*$, then n can be factored in expected polynomial time.

3.9 Computing individual bits

While the discrete logarithm problem in $\mathbb{Z}_p^*$ (§3.6), the RSA problem (§3.3), and the problem of computing square roots modulo a composite integer n (§3.5.2) appear to be intractable, when the problem parameters are carefully selected, it remains possible that it is much easier to compute some partial information about the solution, for example, its least significant bit. It turns out that while some bits of the solution to these problems are indeed easy to compute, other bits are equally difficult to compute as the entire solution. This section summarizes the results known along these lines. The results have applications to the construction of probabilistic public-key encryption schemes (§8.7) and pseudorandom bit generation (§5.5).

Recall (Definition 1.12) that a function f is called a one-way function if $f(x)$ is easy to compute for all x in its domain, but for essentially all y in the range of f, it is computationally infeasible to find any x such that $f(x) = y$.

Three (candidate) one-way functions

Although no proof is known for the existence of a one-way function, it is widely believed that one-way functions do exist (cf. Remark 9.12). The following are candidate one-way functions (in fact, one-way permutations) since they are easy to compute, but their inversion requires the solution of the discrete logarithm problem in $\mathbb{Z}_p^*$, the RSA problem, or the problem of computing square roots modulo n, respectively:

1. *exponentiation modulo p*. Let p be a prime and let α be a generator of $\mathbb{Z}_p^*$. The function is $f : \mathbb{Z}_p^* \longrightarrow \mathbb{Z}_p^*$ defined as $f(x) = \alpha^x \bmod p$.
2. *RSA function*. Let p and q be distinct odd primes, $n = pq$, and let e be an integer such that $\gcd(e, (p-1)(q-1)) = 1$. The function is $f : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n$ defined as $f(x) = x^e \bmod n$.
3. *Rabin function*. Let $n = pq$, where p and q are distinct primes each congruent to 3 modulo 4. The function is $f : Q_n \longrightarrow Q_n$ defined as $f(x) = x^2 \bmod n$. (Recall from Fact 2.160 that f is a permutation, and from Fact 3.46 that inverting f, i.e., computing principal square roots, is difficult assuming integer factorization is intractable.)

The following definitions are used in §3.9.1, 3.9.2, and 3.9.3.

3.81 Definition Let $f : S \longrightarrow S$ be a one-way function, where S is a finite set. A Boolean predicate $B : S \longrightarrow \{0, 1\}$ is said to be a *hard predicate* for f if:

(i) $B(x)$ is easy to compute given $x \in S$; and
(ii) an oracle which computes $B(x)$ correctly with non-negligible advantage[6] given only $f(x)$ (where $x \in S$) can be used to invert f easily.

Informally, B is a hard predicate for the one-way function f if determining the single bit $B(x)$ of information about x, given only $f(x)$, is as difficult as inverting f itself.

3.82 Definition Let $f : S \longrightarrow S$ be a one-way function, where S is a finite set. A k-bit predicate $B^{(k)} : S \longrightarrow \{0, 1\}^k$ is said to be a *hard k-bit predicate* for f if:

(i) $B^{(k)}(x)$ is easy to compute given $x \in S$; and
(ii) for every Boolean predicate $B : \{0, 1\}^k \longrightarrow \{0, 1\}$, an oracle which computes $B(B^{(k)}(x))$ correctly with non-negligible advantage given only $f(x)$ (where $x \in S$) can be used to invert f easily.

If such a $B^{(k)}$ exists, then f is said to *hide k* bits, or the k bits are said to be *simultaneously secure*.

Informally, $B^{(k)}$ is a hard k-bit predicate for the one-way function f if determining any partial information whatsoever about $B^{(k)}(x)$, given only $f(x)$, is as difficult as inverting f itself.

[6] In Definitions 3.81 and 3.82, the probability is taken over all choices of $x \in S$ and random coin tosses of the oracle.

3.9.1 The discrete logarithm problem in $\mathbb{Z}_p^*$ — individual bits

Let p be an odd prime and α a generator of $\mathbb{Z}_p^*$. Assume that the discrete logarithm problem in $\mathbb{Z}_p^*$ is intractable. Let $\beta \in \mathbb{Z}_p^*$, and let $x = \log_\alpha \beta$. Recall from Fact 2.135 that β is a quadratic residue modulo p if and only if x is even. Hence, the least significant bit of x is equal to $(1 - \left(\frac{\beta}{p}\right))/2$, where the Legendre symbol $\left(\frac{\beta}{p}\right)$ can be efficiently computed (Algorithm 2.149). More generally, the following is true.

3.83 Fact Let p be an odd prime, and let α be a generator of $\mathbb{Z}_p^*$. Suppose that $p - 1 = 2^s t$, where t is odd. Then there is an efficient algorithm which, given $\beta \in \mathbb{Z}_p^*$, computes the s least significant bits of $x = \log_\alpha \beta$.

3.84 Fact Let p be a prime and α a generator of $\mathbb{Z}_p^*$. Define the predicate $B : \mathbb{Z}_p^* \longrightarrow \{0, 1\}$ by

$$B(x) = \begin{cases} 0, & \text{if } 1 \le x \le (p-1)/2, \\ 1, & \text{if } (p-1)/2 < x \le p-1. \end{cases}$$

Then B is a hard predicate for the function of exponentiation modulo p. In other words, given p, α, and β, computing the single bit $B(x)$ of the discrete logarithm $x = \log_\alpha \beta$ is as difficult as computing the entire discrete logarithm.

3.85 Fact Let p be a prime and α a generator of $\mathbb{Z}_p^*$. Let $k = O(\lg \lg p)$ be an integer. Let the interval $[1, p-1]$ be partitioned into 2^k intervals $I_0, I_1, \ldots, I_{2^k-1}$ of roughly equal lengths. Define the k-bit predicate $B^{(k)} : \mathbb{Z}_p^* \longrightarrow \{0, 1\}^k$ by $B^{(k)}(x) = j$ if $x \in I_j$. Then $B^{(k)}$ is a hard k-bit predicate for the function of exponentiation modulo p.

3.9.2 The RSA problem — individual bits

Let n be a product of two distinct odd primes p and q, and let e be an integer such that $\gcd(e, (p-1)(q-1)) = 1$. Given n, e, and $c = x^e \bmod n$ (for some $x \in \mathbb{Z}_n$), some information about x is easily obtainable. For example, since e is an odd integer,

$$\left(\frac{c}{n}\right) = \left(\frac{x^e}{n}\right) = \left(\frac{x}{n}\right)^e = \left(\frac{x}{n}\right),$$

and hence the single bit of information $\left(\frac{x}{n}\right)$ can be obtained simply by computing the Jacobi symbol $\left(\frac{c}{n}\right)$ (Algorithm 2.149). There are, however, other bits of information about x that are difficult to compute, as the next two results show.

3.86 Fact Define the predicate $B : \mathbb{Z}_n \longrightarrow \{0, 1\}$ by $B(x) = x \bmod 2$; that is, $B(x)$ is the least significant bit of x. Then B is a hard predicate for the RSA function (see page 115).

3.87 Fact Let $k = O(\lg \lg n)$ be an integer. Define the k-bit predicate $B^{(k)} : \mathbb{Z}_n \longrightarrow \{0, 1\}^k$ by $B^{(k)}(x) = x \bmod 2^k$. That is, $B^{(k)}(x)$ consists of the k least significant bits of x. Then $B^{(k)}$ is a hard k-bit predicate for the RSA function.

Thus the RSA function has $\lg \lg n$ simultaneously secure bits.

3.9.3 The Rabin problem — individual bits

Let $n = pq$, where p and q are distinct primes each congruent to 3 modulo 4.

3.88 Fact Define the predicate $B : Q_n \longrightarrow \{0,1\}$ by $B(x) = x \bmod 2$; that is, $B(x)$ is the least significant bit of the quadratic residue x. Then B is a hard predicate for the Rabin function (see page 115).

3.89 Fact Let $k = O(\lg\lg n)$ be an integer. Define the k-bit predicate $B^{(k)} : Q_n \longrightarrow \{0,1\}^k$ by $B^{(k)}(x) = x \bmod 2^k$. That is, $B^{(k)}(x)$ consists of the k least significant bits of the quadratic residue x. Then $B^{(k)}$ is a hard k-bit predicate for the Rabin function.

Thus the Rabin function has $\lg\lg n$ simultaneously secure bits.

3.10 The subset sum problem

The difficulty of the subset sum problem was the basis for the (presumed) security of the first public-key encryption scheme, called the Merkle-Hellman knapsack scheme (§8.6.1).

3.90 Definition The *subset sum problem* (SUBSET-SUM) is the following: given a set $\{a_1, a_2, \ldots, a_n\}$ of positive integers, called a *knapsack set*, and a positive integer s, determine whether or not there is a subset of the a_j that sum to s. Equivalently, determine whether or not there exist $x_i \in \{0,1\}$, $1 \le i \le n$, such that $\sum_{i=1}^{n} a_i x_i = s$.

The subset sum problem above is stated as a decision problem. It can be shown that the problem is computationally equivalent to its computational version which is to actually determine the x_i such that $\sum_{i=1}^{n} a_i x_i = s$, provided that such x_i exist. Fact 3.91 provides evidence of the intractability of the subset sum problem.

3.91 Fact The subset sum problem is **NP**-complete. The computational version of the subset sum problem is **NP**-hard (see Example 2.74).

Algorithms 3.92 and 3.94 give two methods for solving the computational version of the subset sum problem; both are exponential-time algorithms. Algorithm 3.94 is the fastest method known for the general subset sum problem.

3.92 Algorithm Naive algorithm for subset sum problem

INPUT: a set of positive integers $\{a_1, a_2, \ldots, a_n\}$ and a positive integer s.
OUTPUT: $x_i \in \{0,1\}$, $1 \le i \le n$, such that $\sum_{i=1}^{n} a_i x_i = s$, provided such x_i exist.

1. For each possible vector $(x_1, x_2, \ldots, x_n) \in (\mathbb{Z}_2)^n$ do the following:
   1.1 Compute $l = \sum_{i=1}^{n} a_i x_i$.
   1.2 If $l = s$ then return(a solution is $(x_1, x_2, \ldots, x_n)$).
2. Return(no solution exists).

3.93 Fact Algorithm 3.92 takes $O(2^n)$ steps and, hence, is inefficient.

3.94 Algorithm Meet-in-the-middle algorithm for subset sum problem

INPUT: a set of positive integers $\{a_1, a_2, \dots, a_n\}$ and a positive integer s.
OUTPUT: $x_i \in \{0, 1\}$, $1 \le i \le n$, such that $\sum_{i=1}^{n} a_i x_i = s$, provided such x_i exist.

1. Set $t \leftarrow \lfloor n/2 \rfloor$.
2. Construct a table with entries $(\sum_{i=1}^{t} a_i x_i, (x_1, x_2, \dots, x_t))$ for $(x_1, x_2, \dots, x_t) \in (\mathbb{Z}_2)^t$. Sort this table by first component.
3. For each $(x_{t+1}, x_{t+2}, \dots, x_n) \in (\mathbb{Z}_2)^{n-t}$, do the following:
   3.1 Compute $l = s - \sum_{i=t+1}^{n} a_i x_i$ and check, using a binary search, whether l is the first component of some entry in the table.
   3.2 If $l = \sum_{i=1}^{t} a_i x_i$ then return(a solution is $(x_1, x_2, \dots, x_n)$).
4. Return(no solution exists).

3.95 Fact Algorithm 3.94 takes $O(n2^{n/2})$ steps and, hence, is inefficient.

3.10.1 The L^3-lattice basis reduction algorithm

The L^3-lattice basis reduction algorithm is a crucial component in many number-theoretic algorithms. It is useful for solving certain subset sum problems, and has been used for cryptanalyzing public-key encryption schemes which are based on the subset sum problem.

3.96 Definition Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ be two vectors in $\mathbb{R}^n$. The *inner product* of x and y is the real number

$$< x, y > = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

3.97 Definition Let $y = (y_1, y_2, \dots, y_n)$ be a vector in $\mathbb{R}^n$. The *length* of y is the real number

$$\|y\| = \sqrt{< y, y >} = \sqrt{y_1^2 + y_2^2 + \cdots + y_n^2}.$$

3.98 Definition Let $B = \{b_1, b_2, \dots, b_m\}$ be a set of linearly independent vectors in $\mathbb{R}^n$ (so that $m \le n$). The set L of all integer linear combinations of $b_1, b_2, \dots, b_m$ is called a *lattice* of *dimension* m; that is, $L = \mathbb{Z}b_1 + \mathbb{Z}b_2 + \cdots + \mathbb{Z}b_m$. The set B is called a *basis* for the lattice L.

A lattice can have many different bases. A basis consisting of vectors of relatively small lengths is called *reduced*. The following definition provides a useful notion of a reduced basis, and is based on the Gram-Schmidt orthogonalization process.

3.99 Definition Let $B = \{b_1, b_2, \dots, b_n\}$ be a basis for a lattice $L \subset \mathbb{R}^n$. Define the vectors b_i^* $(1 \le i \le n)$ and the real numbers $\mu_{i,j}$ $(1 \le j < i \le n)$ inductively by

$$\mu_{i,j} = \frac{< b_i, b_j^* >}{< b_j^*, b_j^* >}, \quad 1 \le j < i \le n, \tag{3.8}$$

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*, \quad 1 \le i \le n. \tag{3.9}$$

The basis B is said to be *reduced* (more precisely, *Lovász-reduced*) if

$$|\mu_{i,j}| \le \frac{1}{2}, \quad \text{for } 1 \le j < i \le n$$

(where $|\mu_{i,j}|$ denotes the absolute value of $\mu_{i,j}$), and

$$\|b_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right) \|b_{i-1}^*\|^2, \quad \text{for } 1 < i \leq n. \tag{3.10}$$

Fact 3.100 explains the sense in which the vectors in a reduced basis are relatively short.

3.100 Fact Let $L \subset \mathbb{R}^n$ be a lattice with a reduced basis $\{b_1, b_2, \ldots, b_n\}$.

(i) For every non-zero $x \in L$, $\|b_1\| \leq 2^{(n-1)/2}\|x\|$.

(ii) More generally, for any set $\{a_1, a_2, \ldots, a_t\}$ of linearly independent vectors in L,

$$\|b_j\| \leq 2^{(n-1)/2} \max(\|a_1\|, \|a_2\|, \ldots, \|a_t\|), \quad \text{for } 1 \leq j \leq t.$$

The L^3-lattice basis reduction algorithm (Algorithm 3.101) is a polynomial-time algorithm (Fact 3.103) for finding a reduced basis, given a basis for a lattice.

3.101 Algorithm L^3-lattice basis reduction algorithm

INPUT: a basis $(b_1, b_2, \ldots, b_n)$ for a lattice L in $\mathbb{R}^m$, $m \geq n$.
OUTPUT: a reduced basis for L.

1. $b_1^* \leftarrow b_1$, $B_1 \leftarrow\ < b_1^*, b_1^* >$.
2. For i from 2 to n do the following:
   - 2.1 $b_i^* \leftarrow b_i$.
   - 2.2 For j from 1 to $i-1$, set $\mu_{i,j} \leftarrow\ < b_i, b_j^* > / B_j$ and $b_i^* \leftarrow b_i^* - \mu_{i,j} b_j^*$.
   - 2.3 $B_i \leftarrow\ < b_i^*, b_i^* >$.
3. $k \leftarrow 2$.
4. Execute subroutine RED(k,$k-1$) to possibly update some $\mu_{i,j}$.
5. If $B_k < (\frac{3}{4} - \mu_{k,k-1}^2) B_{k-1}$ then do the following:
   - 5.1 Set $\mu \leftarrow \mu_{k,k-1}$, $B \leftarrow B_k + \mu^2 B_{k-1}$, $\mu_{k,k-1} \leftarrow \mu B_{k-1}/B$, $B_k \leftarrow B_{k-1} B_k / B$, and $B_{k-1} \leftarrow B$.
   - 5.2 Exchange b_k and b_{k-1}.
   - 5.3 If $k > 2$ then exchange $\mu_{k,j}$ and $\mu_{k-1,j}$ for $j = 1, 2, \ldots, k-2$.
   - 5.4 For $i = k+1, k+2, \ldots, n$:
     Set $t \leftarrow \mu_{i,k}$, $\mu_{i,k} \leftarrow \mu_{i,k-1} - \mu t$, and $\mu_{i,k-1} \leftarrow t + \mu_{k,k-1} \mu_{i,k}$.
   - 5.5 $k \leftarrow \max(2, k-1)$.
   - 5.6 Go to step 4.

   Otherwise, for $l = k-2, k-3, \ldots, 1$, execute RED(k,l), and finally set $k \leftarrow k+1$.
6. If $k \leq n$ then go to step 4. Otherwise, return($b_1, b_2, \ldots, b_n$).

RED(k,l) If $|\mu_{k,l}| > \frac{1}{2}$ then do the following:

1. $r \leftarrow \lfloor 0.5 + \mu_{k,l} \rfloor$, $b_k \leftarrow b_k - r b_l$.
2. For j from 1 to $l-1$, set $\mu_{k,j} \leftarrow \mu_{k,j} - r \mu_{l,j}$.
3. $\mu_{k,l} \leftarrow \mu_{k,l} - r$.

3.102 Note (*explanation of selected steps of Algorithm 3.101*)

(i) Steps 1 and 2 initialize the algorithm by computing b_i^* $(1 \leq i \leq n)$ and $\mu_{i,j}$ $(1 \leq j < i \leq n)$ as defined in equations (3.9) and (3.8), and also $B_i =< b_i^*, b_i^* >$ $(1 \leq i \leq n)$.

(ii) k is a variable such that the vectors $b_1, b_2, \ldots, b_{k-1}$ are reduced (initially $k = 2$ in step 3). The algorithm then attempts to modify b_k, so that $b_1, b_2, \ldots, b_k$ are reduced.

(iii) In step 4, the vector b_k is modified appropriately so that $|\mu_{k,k-1}| \le \frac{1}{2}$, and the $\mu_{k,j}$ are updated for $1 \le j < k-1$.

(iv) In step 5, if the condition of equation (3.10) is violated for $i = k$, then vectors b_k and b_{k-1} are exchanged and their corresponding parameters are updated. Also, k is decremented by 1 since then it is only guaranteed that $b_1, b_2, \dots, b_{k-2}$ are reduced. Otherwise, b_k is modified appropriately so that $|\mu_{k,j}| \le \frac{1}{2}$ for $j = 1, 2, \dots, k-2$, while keeping (3.10) satisfied. k is then incremented because now $b_1, b_2, \dots, b_k$ are reduced.

It can be proven that the L^3-algorithm terminates after a finite number of iterations. Note that if L is an integer lattice, i.e. $L \subset \mathbb{Z}^n$, then the L^3-algorithm only operates on rational numbers. The precise running time is given next.

3.103 Fact Let $L \subset \mathbb{Z}^n$ be a lattice with basis $\{b_1, b_2, \dots, b_n\}$, and let $C \in \mathbb{R}$, $C \ge 2$, be such that $\|b_i\|^2 \le C$ for $i = 1, 2, \dots, n$. Then the number of arithmetic operations needed by Algorithm 3.101 is $O(n^4 \log C)$, on integers of size $O(n \log C)$ bits.

3.10.2 Solving subset sum problems of low density

The density of a knapsack set, as defined below, provides a measure of the size of the knapsack elements.

3.104 Definition Let $S = \{a_1, a_2, \dots, a_n\}$ be a knapsack set. The *density* of S is defined to be

$$d = \frac{n}{\max\{\lg a_i \mid 1 \le i \le n\}}.$$

Algorithm 3.105 reduces the subset sum problem to one of finding a particular short vector in a lattice. By Fact 3.100, the reduced basis produced by the L^3-algorithm includes a vector of length which is guaranteed to be within a factor of $2^{(n-1)/2}$ of the shortest non-zero vector of the lattice. In practice, however, the L^3-algorithm usually finds a vector which is much shorter than what is guaranteed by Fact 3.100. Hence, the L^3-algorithm can be expected to find the short vector which yields a solution to the subset sum problem, provided that this vector is shorter than most of the non-zero vectors in the lattice.

3.105 Algorithm Solving subset sum problems using L^3-algorithm

INPUT: a set of positive integers $\{a_1, a_2, \dots, a_n\}$ and an integer s.
OUTPUT: $x_i \in \{0, 1\}$, $1 \le i \le n$, such that $\sum_{i=1}^n a_i x_i = s$, provided such x_i exist.

1. Let $m = \lceil \frac{1}{2}\sqrt{n} \rceil$.
2. Form an $(n+1)$-dimensional lattice L with basis consisting of the rows of the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & ma_1 \\ 0 & 1 & 0 & \cdots & 0 & ma_2 \\ 0 & 0 & 1 & \cdots & 0 & ma_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & ma_n \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \cdots & \frac{1}{2} & ms \end{pmatrix}$$

3. Find a reduced basis B of L (use Algorithm 3.101).
4. For each vector $y = (y_1, y_2, \dots, y_{n+1})$ in B, do the following:

4.1 If $y_{n+1} = 0$ and $y_i \in \{-\frac{1}{2}, \frac{1}{2}\}$ for all $i = 1, 2, \ldots, n$, then do the following:
For $i = 1, 2, \ldots, n$, set $x_i \leftarrow y_i + \frac{1}{2}$.
If $\sum_{i=1}^{n} a_i x_i = s$, then return(a solution is $(x_1, x_2, \ldots, x_n)$).
For $i = 1, 2, \ldots, n$, set $x_i \leftarrow -y_i + \frac{1}{2}$.
If $\sum_{i=1}^{n} a_i x_i = s$, then return(a solution is $(x_1, x_2, \ldots, x_n)$).

5. Return(FAILURE). (Either no solution exists, or the algorithm has failed to find one.)

Justification. Let the rows of the matrix A be $b_1, b_2, \ldots, b_{n+1}$, and let L be the $(n+1)$-dimensional lattice generated by these vectors. If $(x_1, x_2, \ldots, x_n)$ is a solution to the subset sum problem, the vector $y = \sum_{i=1}^{n} x_i b_i - b_{n+1}$ is in L. Note that $y_i \in \{-\frac{1}{2}, \frac{1}{2}\}$ for $i = 1, 2, \ldots, n$ and $y_{n+1} = 0$. Since $\|y\| = \sqrt{y_1^2 + y_2^2 + \cdots + y_{n+1}^2}$ the vector y is a vector of short length in L. If the density of the knapsack set is small, i.e. the a_i are large, then most vectors in L will have relatively large lengths, and hence y may be the unique shortest non-zero vector in L. If this is indeed the case, then there is good possibility of the L^3-algorithm finding a basis which includes this vector.

Algorithm 3.105 is not guaranteed to succeed. Assuming that the L^3-algorithm always produces a basis which includes the shortest non-zero lattice vector, Algorithm 3.105 succeeds with high probability if the density of the knapsack set is less than 0.9408.

3.10.3 Simultaneous diophantine approximation

Simultaneous diophantine approximation is concerned with approximating a vector $(\frac{q_1}{q}, \frac{q_2}{q}, \ldots, \frac{q_n}{q})$ of rational numbers (more generally, a vector $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of real numbers) by a vector $(\frac{p_1}{p}, \frac{p_2}{p}, \ldots, \frac{p_n}{p})$ of rational numbers with a smaller denominator p. Algorithms for finding simultaneous diophantine approximation have been used to break some knapsack public-key encryption schemes (§8.6).

3.106 Definition Let δ be a real number. The vector $(\frac{p_1}{p}, \frac{p_2}{p}, \ldots, \frac{p_n}{p})$ of rational numbers is said to be a *simultaneous diophantine approximation of δ-quality* to the vector $(\frac{q_1}{q}, \frac{q_2}{q}, \ldots, \frac{q_n}{q})$ of rational numbers if $p < q$ and

$$\left| p\frac{q_i}{q} - p_i \right| \le q^{-\delta} \text{ for } i = 1, 2, \ldots, n.$$

(The larger δ is, the better is the approximation.) Furthermore, it is an *unusually good simultaneous diophantine approximation* (UGSDA) if $\delta > \frac{1}{n}$.

Fact 3.107 shows that an UGSDA is indeed unusual.

3.107 Fact For $n \ge 2$, the set

$$S_n(q) = \left\{ \left(\frac{q_1}{q}, \frac{q_2}{q}, \ldots, \frac{q_n}{q} \right) \mid 0 \le q_i < q, \ \ \gcd(q_1, q_2, \ldots, q_n, q) = 1 \right\}$$

has at least $\frac{1}{2}q^n$ members. Of these, at most $O(q^{n(1-\delta)+1})$ members have at least one δ-quality simultaneous diophantine approximation. Hence, for any fixed $\delta > \frac{1}{n}$, the fraction of members of $S_n(q)$ having at least one UGSDA approaches 0 as $q \to \infty$.

Algorithm 3.108 reduces the problem of finding a δ-quality simultaneous diophantine approximation, and hence also a UGSDA, to the problem of finding a short vector in a lattice. The latter problem can (usually) be solved using the L^3-lattice basis reduction.

3.108 Algorithm Finding a δ-quality simultaneous diophantine approximation

INPUT: a vector $w = (\frac{q_1}{q}, \frac{q_2}{q}, \dots, \frac{q_n}{q})$ of rational numbers, and a rational number $\delta > 0$.
OUTPUT: a δ-quality simultaneous diophantine approximation $(\frac{p_1}{p}, \frac{p_2}{p}, \dots, \frac{p_n}{p})$ of w.

1. Choose an integer $\lambda \approx q^\delta$.
2. Use Algorithm 3.101 to find a reduced basis B for the $(n+1)$-dimensional lattice L which is generated by the rows of the matrix

$$A = \begin{pmatrix} \lambda q & 0 & 0 & \cdots & 0 & 0 \\ 0 & \lambda q & 0 & \cdots & 0 & 0 \\ 0 & 0 & \lambda q & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda q & 0 \\ -\lambda q_1 & -\lambda q_2 & -\lambda q_3 & \cdots & -\lambda q_n & 1 \end{pmatrix}$$

3. For each $v = (v_1, v_2, \dots, v_n, v_{n+1})$ in B such that $v_{n+1} \neq q$, do the following:
   3.1 $p \leftarrow v_{n+1}$.
   3.2 For i from 1 to n, set $p_i \leftarrow \frac{1}{q}\left(\frac{v_i}{\lambda} + pq_i\right)$.
   3.3 If $|p\frac{q_i}{q} - p_i| \leq q^{-\delta}$ for each i, $1 \leq i \leq n$, then return$(\frac{p_1}{p}, \frac{p_2}{p}, \dots, \frac{p_n}{p})$.
4. Return(FAILURE). (Either no δ-quality simultaneous diophantine approximation exists, or the algorithm has failed to find one.)

Justification. Let the rows of the matrix A be denoted by $b_1, b_2, \dots, b_{n+1}$. Suppose that $(\frac{q_1}{q}, \frac{q_2}{q}, \dots, \frac{q_n}{q})$ has a δ-quality approximation $(\frac{p_1}{p}, \frac{p_2}{p}, \dots, \frac{p_n}{p})$. Then the vector

$$\begin{aligned} x &= p_1 b_1 + p_2 b_2 + \cdots + p_n b_n + p b_{n+1} \\ &= (\lambda(p_1 q - pq_1), \lambda(p_2 q - pq_2), \dots, \lambda(p_n q - pq_n), p) \end{aligned}$$

is in L and has length less than approximately $(\sqrt{n+1})q$. Thus x is short compared to the original basis vectors, which are of length roughly $q^{1+\delta}$. Also, if $v = (v_1, v_2, \dots, v_{n+1})$ is a vector in L of length less than q, then the vector $(\frac{p_1}{p}, \frac{p_2}{p}, \dots, \frac{p_n}{p})$ defined in step 3 is a δ-quality approximation. Hence there is a good possibility that the L^3-algorithm will produce a reduced basis which includes a vector v that corresponds to a δ-quality approximation.

3.11 Factoring polynomials over finite fields

The problem considered in this section is the following: given a polynomial $f(x) \in \mathbb{F}_q[x]$, with $q = p^m$, find its factorization $f(x) = f_1(x)^{e_1} f_2(x)^{e_2} \cdots f_t(x)^{e_t}$, where each $f_i(x)$ is an irreducible polynomial in $\mathbb{F}_q[x]$ and each $e_i \geq 1$. (e_i is called the *multiplicity* of the factor $f_i(x)$.) Several situations call for the factoring of polynomials over finite fields, such as index-calculus algorithms in $\mathbb{F}^*_{2^m}$ (Example 3.70) and Chor-Rivest public-key encryption (§8.6.2). This section presents an algorithm for square-free factorization, and Berlekamp's classical deterministic algorithm for factoring polynomials which is efficient if the underlying field is small. Efficient randomized algorithms are known for the case of large q; references are provided on page 132.

3.11.1 Square-free factorization

Observe first that $f(x)$ may be divided by its leading coefficient. Thus, it may be assumed that $f(x)$ is monic (see Definition 2.187). This section shows how the problem of factoring a monic polynomial $f(x)$ may then be reduced to the problem of factoring one or more monic square-free polynomials.

3.109 Definition Let $f(x) \in \mathbb{F}_q[x]$. Then $f(x)$ is *square-free* if it has no repeated factors, i.e., there is no polynomial $g(x)$ with $\deg g(x) \geq 1$ such that $g(x)^2$ divides $f(x)$. The *square-free factorization* of $f(x)$ is $f(x) = \prod_{i=1}^{k} f_i(x)^i$, where each $f_i(x)$ is a square-free polynomial and $\gcd(f_i(x), f_j(x)) = 1$ for $i \neq j$. (Some of the $f_i(x)$ in the square-free factorization of $f(x)$ may be 1.)

Let $f(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree $n \geq 1$. The (formal) *derivative* of $f(x)$ is the polynomial $f'(x) = \sum_{i=0}^{n-1} a_{i+1}(i+1)x^i$. If $f'(x) = 0$, then, because p is the characteristic of $\mathbb{F}_q$, in each term $a_i x^i$ of $f(x)$ for which $a_i \neq 0$, the exponent of x must be a multiple of p. Hence, $f(x)$ has the form $f(x) = a(x)^p$, where $a(x) = \sum_{i=0}^{n/p} a_{ip}^{q/p} x^i$, and the problem of finding the square-free factorization of $f(x)$ is reduced to finding that of $a(x)$. Now, it is possible that $a'(x) = 0$, but repeating this process as necessary, it may be assumed that $f'(x) \neq 0$.

Next, let $g(x) = \gcd(f(x), f'(x))$. Noting that an irreducible factor of multiplicity k in $f(x)$ will have multiplicity $k-1$ in $f'(x)$ if $\gcd(k,p) = 1$, and will retain multiplicity k in $f'(x)$ otherwise, the following conclusions may be drawn. If $g(x) = 1$, then $f(x)$ has no repeated factors; and if $g(x)$ has positive degree, then $g(x)$ is a non-trivial factor of $f(x)$, and $f(x)/g(x)$ has no repeated factors. Note, however, the possibility of $g(x)$ having repeated factors, and, indeed, the possibility that $g'(x) = 0$. Nonetheless, $g(x)$ can be refined further as above. The steps are summarized in Algorithm 3.110. In the algorithm, F denotes the square-free factorization of a factor of $f(x)$ in factored form.

3.110 Algorithm Square-free factorization

SQUARE-FREE($f(x)$)

INPUT: a monic polynomial $f(x) \in \mathbb{F}_q[x]$ of degree ≥ 1, where $\mathbb{F}_q$ has characteristic p.

OUTPUT: the square-free factorization of $f(x)$.

1. Set $i \leftarrow 1$, $F \leftarrow 1$, and compute $f'(x)$.
2. If $f'(x) = 0$ then set $f(x) \leftarrow f(x)^{1/p}$ and $F \leftarrow (\text{SQUARE-FREE}(f(x)))^p$.
   Otherwise (i.e. $f'(x) \neq 0$) do the following:
   2.1 Compute $g(x) \leftarrow \gcd(f(x), f'(x))$ and $h(x) \leftarrow f(x)/g(x)$.
   2.2 While $h(x) \neq 1$ do the following:
   Compute $\overline{h}(x) \leftarrow \gcd(h(x), g(x))$ and $l(x) \leftarrow h(x)/\overline{h}(x)$.
   Set $F \leftarrow F \cdot l(x)^i$, $i \leftarrow i+1$, $h(x) \leftarrow \overline{h}(x)$, and $g(x) \leftarrow g(x)/\overline{h}(x)$.
   2.3 If $g(x) \neq 1$ then set $g(x) \leftarrow g(x)^{1/p}$ and $F \leftarrow F \cdot (\text{SQUARE-FREE}(g(x)))^p$.
3. Return(F).

Once the square-free factorization $f(x) = \prod_{i=1}^{k} f_i(x)^i$ is found, the square-free polynomials $f_1(x), f_2(x), \ldots, f_k(x)$ need to be factored in order to obtain the complete factorization of $f(x)$.

3.11.2 Berlekamp's Q-matrix algorithm

Let $f(x) = \prod_{i=1}^{t} f_i(x)$ be a monic polynomial in $\mathbb{F}_q[x]$ of degree n having distinct irreducible factors $f_i(x)$, $1 \le i \le t$. Berlekamp's Q-matrix algorithm (Algorithm 3.111) for factoring $f(x)$ is based on the following facts. The set of polynomials

$$\mathcal{B} = \{b(x) \in \mathbb{F}_q[x]/(f(x)) \mid b(x)^q \equiv b(x) \pmod{f(x)}\}$$

is a vector space of dimension t over $\mathbb{F}_q$. $\mathcal{B}$ consists of precisely those vectors in the null space of the matrix $Q - I_n$, where Q is the $n \times n$ matrix with (i,j)-entry q_{ij} specified by

$$x^{iq} \bmod f(x) = \sum_{j=0}^{n-1} q_{ij} x^j, \quad 0 \le i \le n-1,$$

and where I_n is the $n \times n$ identity matrix. A basis $B = \{v_1(x), v_2(x), \dots, v_t(x)\}$ for $\mathcal{B}$ can thus be found by standard techniques from linear algebra. Finally, for each pair of distinct factors $f_i(x)$ and $f_j(x)$ of $f(x)$ there exists some $v_k(x) \in B$ and some $\alpha \in \mathbb{F}_q$ such that $f_i(x)$ divides $v_k(x) - \alpha$ but $f_j(x)$ does not divide $v_k(x) - \alpha$; these two factors can thus be split by computing $\gcd(f(x), v_k(x) - \alpha)$. In Algorithm 3.111, a vector $w = (w_0, w_1, \dots, w_{n-1})$ is identified with the polynomial $w(x) = \sum_{i=0}^{n-1} w_i x^i$.

3.111 Algorithm Berlekamp's Q-matrix algorithm for factoring polynomials over finite fields

INPUT: a square-free monic polynomial $f(x)$ of degree n in $\mathbb{F}_q[x]$.
OUTPUT: the factorization of $f(x)$ into monic irreducible polynomials.

1. For each i, $0 \le i \le n-1$, compute the polynomial
$$x^{iq} \bmod f(x) \;=\; \sum_{j=0}^{n-1} q_{ij} x^j.$$
Note that each q_{ij} is an element of $\mathbb{F}_q$.
2. Form the $n \times n$ matrix Q whose (i,j)-entry is q_{ij}.
3. Determine a basis $v_1, v_2, \dots, v_t$ for the null space of the matrix $(Q - I_n)$, where I_n is the $n \times n$ identity matrix. The number of irreducible factors of $f(x)$ is precisely t.
4. Set $F \leftarrow \{f(x)\}$. (F is the set of factors of $f(x)$ found so far; their product is equal to $f(x)$.)
5. For i from 1 to t do the following:
   - 5.1 For each polynomial $h(x) \in F$ such that $\deg h(x) > 1$ do the following: compute $\gcd(h(x), v_i(x) - \alpha)$ for each $\alpha \in \mathbb{F}_q$, and replace $h(x)$ in F by all those polynomials in the gcd computations whose degrees are ≥ 1.
6. Return(the polynomials in F are the irreducible factors of $f(x)$).

3.112 Fact The running time of Algorithm 3.111 for factoring a square-free polynomial of degree n over $\mathbb{F}_q$ is $O(n^3 + tqn^2)$ $\mathbb{F}_q$-operations, where t is the number of irreducible factors of $f(x)$. The method is efficient only when q is small.

3.12 Notes and further references

§3.1

Many of the topics discussed in this chapter lie in the realm of algorithmic number theory. Excellent references on this subject include the books by Bach and Shallit [70], Cohen [263], and Pomerance [993]. Adleman and McCurley [15] give an extensive survey of the important open problems in algorithmic number theory. Two other recommended surveys are by Bach [65] and Lenstra and Lenstra [748]. Woll [1253] gives an overview of the reductions among thirteen of these problems.

§3.2

A survey of the integer factorization problem is given by Pomerance [994]. See also Chapters 8 and 10 of Cohen [263], and the books by Bressoud [198] and Koblitz [697]. Brillhart et al. [211] provide extensive listings of factorizations of integers of the form $b^n \pm 1$ for "small" n and $b = 2, 3, 5, 6, 7, 10, 11, 12$.

Bach and Sorenson [71] presented some algorithms for recognizing perfect powers (cf. Note 3.6), one having a worst-case running time of $O(\lg^3 n)$ bit operations, and a second having an average-case running time of $O(\lg^2 n)$ bit operations. A more recent algorithm of Bernstein [121] runs in essentially linear time $O((\lg n)^{1+o(1)})$. Fact 3.7 is from Knuth [692]. Pages 367–369 of this reference contain explicit formulas regarding the expected sizes of the largest and second largest prime factors, and the expected total number of prime factors, of a randomly chosen positive integer. For further results, see Knuth and Trabb Pardo [694], who prove that the average number of bits in the k^{th} largest prime factor of a random m-bit number is asymptotically equivalent to the average length of the k^{th} longest cycle in a permutation on m objects.

Floyd's cycle-finding algorithm (Note 3.8) is described by Knuth [692, p.7]. Sedgewick, Szymanski, and Yao [1106] showed that by saving a small number of values from the x_i sequence, a collision can be found by doing roughly one-third the work as in Floyd's cycle-finding algorithm. Pollard's rho algorithm for factoring (Algorithm 3.9) is due to Pollard [985]. Regarding Note 3.12, Cohen [263, p.422] provides an explanation for the restriction $c \neq 0, -2$. Brent [196] presented a cycle-finding algorithm which is better on average than Floyd's cycle-finding algorithm, and applied it to yield a factorization algorithm which is similar to Pollard's but about 24 percent faster. Brent and Pollard [197] later modified this algorithm to factor the eighth Fermat number $F_8 = 2^{2^8} + 1$. Using techniques from algebraic geometry, Bach [67] obtained the first rigorously proven result concerning the expected running time of Pollard's rho algorithm: for fixed k, the probability that a prime factor p is discovered before step k is at least $\binom{k}{2}/p + O(p^{-3/2})$ as $p \to \infty$.

The $p-1$ algorithm (Algorithm 3.14) is due to Pollard [984]. Several practical improvements have been proposed for the $p-1$ algorithm, including those by Montgomery [894] and Montgomery and Silverman [895], the latter using fast Fourier transform techniques. Williams [1247] presented an algorithm for factoring n which is efficient if n has a prime factor p such that $p+1$ is smooth. These methods were generalized by Bach and Shallit [69] to techniques that factor n efficiently provided n has a prime factor p such that the k^{th} cyclotomic polynomial $\Phi_k(p)$ is smooth. The first few cyclotomic polynomials are $\Phi_1(p) = p-1$, $\Phi_2(p) = p+1$, $\Phi_3(p) = p^2+p+1$, $\Phi_4(p) = p^2+1$, $\Phi_5(p) = p^4+p^3+p^2+p+1$, and $\Phi_6(p) = p^2 - p + 1$.

The elliptic curve factoring algorithm (ECA) of §3.2.4 was invented by Lenstra [756]. Montgomery [894] gave several practical improvements to the ECA. Silverman and

Wagstaff [1136] gave a practical analysis of the complexity of the ECA, and suggested optimal parameter selection and running-time guidelines. Lenstra and Manasse [753] implemented the ECA on a network of MicroVAX computers, and were successful in finding 35-decimal digit prime factors of large (at least 85 digit) composite integers. Later, Dixon and Lenstra [350] implemented the ECA on a 16K MasPar (massively parallel) SIMD (single instruction, multiple data) machine. The largest factor they found was a 40-decimal digit prime factor of an 89-digit composite integer. On November 26 1995, Peter Montgomery reported finding a 47-decimal digit prime factor of the 99-digit composite integer $5^{256}+1$ with the ECA.

Hafner and McCurley [536] estimated the number of integers $n \leq x$ that can be factored with probability at least $\frac{1}{2}$ using at most t arithmetic operations, by trial division and the elliptic curve algorithm. Pomerance and Sorenson [997] provided the analogous estimates for Pollard's $p-1$ algorithm and Williams' $p+1$ algorithm. They conclude that for a given running time bound, both Pollard's $p-1$ and Williams' $p+1$ algorithms factor more integers than trial division, but fewer than the elliptic curve algorithm.

Pomerance [994] credits the idea of multiplying congruences to produce a solution to $x^2 \equiv y^2 \pmod n$ for the purpose of factoring n (§3.2.5) to some old work of Kraitchik circa 1926-1929. The *continued fraction factoring algorithm*, first introduced by Lehmer and Powers [744] in 1931, and refined more than 40 years later by Morrison and Brillhart [908], was the first realization of a random square method to result in a subexponential-time algorithm. The algorithm was later analyzed by Pomerance [989] and conjectured to have an expected running time of $L_n[\frac{1}{2},\sqrt{2}]$. If the smoothness testing in the algorithm is done with the elliptic curve method, then the expected running time drops to $L_n[\frac{1}{2},1]$. Morrison and Brillhart were also the first to use the idea of a factor base to test for good (a_i,b_i) pairs. The continued fraction algorithm was the champion of factoring algorithms from the mid 1970s until the early 1980s, when it was surpassed by the quadratic sieve algorithm.

The quadratic sieve (QS) (§3.2.6) was discovered by Pomerance [989, 990]. The multiple polynomial variant of the quadratic sieve (Note 3.25) is due to P. Montgomery, and is described by Pomerance [990]; see also Silverman [1135]. A detailed practical analysis of the QS is given by van Oorschot [1203]. Several practical improvements to the original algorithms have subsequently been proposed and successfully implemented. The first serious implementation of the QS was by Gerver [448] who factored a 47-decimal digit number. In 1984, Davis, Holdridge, and Simmons [311] factored a 71-decimal digit number with the QS. In 1988, Lenstra and Manasse [753] used the QS to factor a 106-decimal digit number by distributing the computations to hundreds of computers by electronic mail; see also Lenstra and Manasse [754]. In 1993, the QS was used by Denny et al. [333] to factor a 120-decimal digit number. In 1994, the 129-decimal digit (425 bit) RSA-129 challenge number (see Gardner [440]), was factored by Atkins et al. [59] by enlisting the help of about 1600 computers around the world. The factorization was carried out in 8 months. Table 3.3 shows the estimated time taken, in mips years, for the above factorizations. A *mips year* is equivalent to the computational power of a computer that is rated at 1 mips (million instructions per second) and utilized for one year, or, equivalently, about $3 \cdot 10^{13}$ instructions.

The number field sieve was first proposed by Pollard [987] and refined by others. Lenstra et al. [752] described the special number field sieve (SNFS) for factoring integers of the form $r^e - s$ for small positive r and $|s|$. A readable introduction to the algorithm is provided by Pomerance [995]. A detailed report of an SNFS implementation is given by Lenstra et al. [751]. This implementation was used to factor the ninth Fermat number $F_9 = 2^{512}+1$, which is the product of three prime factors having 7, 49, and 99 decimal digits. The general number field sieve (GNFS) was introduced by Buhler, Lenstra, and Pomerance [219].

Year	Number of digits	mips years
1984	71	0.1
1988	106	140
1993	120	825
1994	129	5000

Table 3.3: *Running time estimates for numbers factored with QS.*

Coppersmith [269] proposed modifications to the GNFS which improve its running time to $L_n[\frac{1}{3}, 1.902]$, however, the method is not practical; another modification (also impractical) allows a precomputation taking $L_n[\frac{1}{3}, 2.007]$ time and $L_n[\frac{1}{3}, 1.639]$ storage, following which all integers in a large range of values can be factored in $L_n[\frac{1}{3}, 1.639]$ time. A detailed report of a GNFS implementation on a massively parallel computer with 16384 processors is given by Bernstein and Lenstra [122]. See also Buchmann, Loho, and Zayer [217], and Golliver, Lenstra, and McCurley [493]. More recently, Dodson and Lenstra [356] reported on their GNFS implementation which was successful in factoring a 119-decimal digit number using about 250 mips years of computing power. They estimated that this factorization completed about 2.5 times faster than it would with the quadratic sieve. Most recently, Lenstra [746] announced the factorization of the 130-decimal digit RSA-130 challenge number using the GNFS. This number is the product of two 65-decimal digit primes. The factorization was estimated to have taken about 500 mips years of computing power (compare with Table 3.3). The book edited by Lenstra and Lenstra [749] contains several other articles related to the number field sieve.

The ECA, continued fraction algorithm, quadratic sieve, special number field sieve, and general number field sieve have *heuristic* (or *conjectured*) rather than *proven* running times because the analyses make (reasonable) assumptions about the proportion of integers generated that are smooth. See Canfield, Erdös, and Pomerance [231] for bounds on the proportion of y-smooth integers in the interval $[2, x]$. Dixon's algorithm [351] was the first rigorously analyzed subexponential-time algorithm for factoring integers. The fastest rigorously analyzed algorithm currently known is due to Lenstra and Pomerance [759] with an expected running time of $L_n[\frac{1}{2}, 1]$. These algorithms are of theoretical interest only, as they do not appear to be practical.

§3.3

The RSA problem was introduced in the landmark 1977 paper by Rivest, Shamir, and Adleman [1060].

§3.4

The quadratic residuosity problem is of much historical interest, and was one of the main algorithmic problems discussed by Gauss [444].

§3.5

An extensive treatment of the problem of finding square roots modulo a prime p, or more generally, the problem of finding d^{th} roots in a finite field, can be found in Bach and Shallit [70]. The presentation of Algorithm 3.34 for finding square roots modulo a prime is derived from Koblitz [697, pp.48-49]; a proof of correctness can be found there. Bach and Shallit attribute the essential ideas of Algorithm 3.34 to an 1891 paper by A. Tonelli. Algorithm 3.39 is from Bach and Shallit [70], who attribute it to a 1903 paper of M. Cipolla.

The computational equivalence of computing square roots modulo a composite n and factoring n (Fact 3.46 and Note 3.47) was first discovered by Rabin [1023].

§3.6

A survey of the discrete logarithm problem is given by McCurley [827]. See also Odlyzko [942] for a survey of recent advances.

Knuth [693] attributes the baby-step giant-step algorithm (Algorithm 3.56) to D. Shanks. The baby-step giant-step algorithms for searching restricted exponent spaces (cf. Note 3.59) are described by Heiman [546]. Suppose that p is a k-bit prime, and that only exponents of Hamming weight t are used. Coppersmith (personal communication, July 1995) observed that this exponent space can be searched in $k \cdot \binom{k/2}{t/2}$ steps by dividing the exponent into two equal pieces so that the Hamming weight of each piece is $t/2$; if k is much smaller than $2^{t/2}$, this is an improvement over Note 3.59.

Pollard's rho algorithm for logarithms (Algorithm 3.60) is due to Pollard [986]. Pollard also presented a *lambda method* for computing discrete logarithms which is applicable when x, the logarithm sought, is known to lie in a certain interval. More specifically, if the interval is of width w, the method is expected to take $O(\sqrt{w})$ group operations and requires storage for only $O(\lg w)$ group elements. Van Oorschot and Wiener [1207] showed how Pollard's rho algorithm can be parallelized so that using m processors results in a speedup by a factor of m. This has particular significance to cyclic groups such as elliptic curve groups, for which no subexponential-time discrete logarithm algorithm is known.

The Pohlig-Hellman algorithm (Algorithm 3.63) was discovered by Pohlig and Hellman [982]. A variation which represents the logarithm in a mixed-radix notation and does not use the Chinese remainder theorem was given by Thiong Ly [1190].

According to McCurley [827], the basic ideas behind the index-calculus algorithm (Algorithm 3.68) first appeared in the work of Kraitchik (circa 1922-1924) and of Cunningham (see Western and Miller [1236]), and was rediscovered by several authors. Adleman [8] described the method for the group $\mathbb{Z}_p^*$ and analyzed the complexity of the algorithm. Hellman and Reyneri [555] gave the first description of an index-calculus algorithm for extension fields $\mathbb{F}_{p^m}$ with p fixed.

Coppersmith, Odlyzko, and Schroeppel [280] presented three variants of the index-calculus method for computing logarithms in $\mathbb{Z}_p^*$: the *linear sieve*, the *residue list sieve*, and the *Gaussian integer method*. Each has a heuristic expected running time of $L_p[\frac{1}{2}, 1]$ (cf. Note 3.71). The Gaussian integer method, which is related to the method of ElGamal [369], was implemented in 1990 by LaMacchia and Odlyzko [736] and was successful in computing logarithms in $\mathbb{Z}_p^*$ with p a 192-bit prime. The paper concludes that it should be feasible to compute discrete logarithms modulo primes of about 332 bits (100 decimal digits) using the Gaussian integer method. Gordon [510] adapted the number field sieve for factoring integers to the problem of computing logarithms in $\mathbb{Z}_p^*$; his algorithm has a heuristic expected running time of $L_p[\frac{1}{3}, c]$, where $c = 3^{2/3} \approx 2.080$. Schirokauer [1092] subsequently presented a modification of Gordon's algorithm that has a heuristic expected running time of $L_p[\frac{1}{3}, c]$, where $c = (64/9)^{1/3} \approx 1.923$ (Note 3.72). This is the same running time as conjectured for the number field sieve for factoring integers (see §3.2.7). Recently, Weber [1232] implemented the algorithms of Gordon and Schirokauer and was successful in computing logarithms in $\mathbb{Z}_p^*$, where p is a 40-decimal digit prime such that $p-1$ is divisible by a 38-decimal digit (127-bit) prime. More recently, Weber, Denny, and Zayer (personal communication, April 1996) announced the solution of a discrete logarithm problem modulo a 75-decimal digit (248-bit) prime p with $(p-1)/2$ prime.

Blake et al. [145] made improvements to the index-calculus technique for $\mathbb{F}_{2^m}^*$ and computed logarithms in $\mathbb{F}_{2^{127}}^*$. Coppersmith [266] dramatically improved the algorithm and showed that under reasonable assumptions the expected running time of his improved algorithm is $L_{2^m}[\frac{1}{3}, c]$ for some constant $c < 1.587$ (Note 3.72). Later, Odlyzko [940] gave several refinements to Coppersmith's algorithm, and a detailed practical analysis; this paper provides the most extensive account to date of the discrete logarithm problem in $\mathbb{F}_{2^m}^*$. A similar practical analysis was also given by van Oorschot [1203]. Most recently in 1992, Gordon and McCurley [511] reported on their massively parallel implementation of Coppersmith's algorithm, combined with their own improvements. Using primarily a 1024 processor nCUBE-2 machine with 4 megabytes of memory per processor, they completed the precomputation of logarithms of factor base elements (which is the dominant step of the algorithm) required to compute logarithms in $\mathbb{F}_{2^{227}}^*$, $\mathbb{F}_{2^{313}}^*$, and $\mathbb{F}_{2^{401}}^*$. The calculations for $\mathbb{F}_{2^{401}}^*$ were estimated to take 5 days. Gordon and McCurley also completed most of the precomputations required for computing logarithms in $\mathbb{F}_{2^{503}}^*$; the amount of time to complete this task on the 1024 processor nCUBE-2 was estimated to be 44 days. They concluded that computing logarithms in the multiplicative groups of fields as large as $\mathbb{F}_{2^{593}}$ still seems to be out of their reach, but might be possible in the near future with a concerted effort.

It was not until 1992 that a subexponential-time algorithm for computing discrete logarithms over all finite fields $\mathbb{F}_q$ was discovered by Adleman and DeMarrais [11]. The expected running time of the algorithm is conjectured to be $L_q[\frac{1}{2}, c]$ for some constant c. Adleman [9] generalized the number field sieve from algebraic number fields to algebraic function fields which resulted in an algorithm, called the *function field sieve*, for computing discrete logarithms in $\mathbb{F}_{p^m}^*$; the algorithm has a heuristic expected running time of $L_{p^m}[\frac{1}{3}, c]$ for some constant $c > 0$ when $\log p \leq m^{g(m)}$, and where g is any function such that $0 < g(m) < 0.98$ and $\lim_{m\to\infty} g(m) = 0$. The practicality of the function field sieve has not yet been determined. It remains an open problem to find an algorithm with a heuristic expected running time of $L_q[\frac{1}{3}, c]$ for *all* finite fields $\mathbb{F}_q$.

The algorithms mentioned in the previous three paragraphs have *heuristic* (or *conjectured*) rather than *proven* running times because the analyses make some (reasonable) assumptions about the proportion of integers or polynomials generated that are smooth, and also because it is not clear when the system of linear equations generated has full rank, i.e., yields a unique solution. The best rigorously analyzed algorithms known for the discrete logarithm problem in $\mathbb{Z}_p^*$ and $\mathbb{F}_{2^m}^*$ are due to Pomerance [991] with expected running times of $L_p[\frac{1}{2}, \sqrt{2}]$ and $L_{2^m}[\frac{1}{2}, \sqrt{2}]$, respectively. Lovorn [773] obtained rigorously analyzed algorithms for the fields $\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^m}$ with $\log p < m^{0.98}$, having expected running times of $L_{p^2}[\frac{1}{2}, \frac{3}{2}]$ and $L_{p^m}[\frac{1}{2}, \sqrt{2}]$, respectively.

The linear system of equations collected in the quadratic sieve and number field sieve factoring algorithms, and the index-calculus algorithms for computing discrete logarithms in $\mathbb{Z}_p^*$ and $\mathbb{F}_{2^m}^*$, are very large. For the problem sizes currently under consideration, these systems cannot be solved using ordinary linear algebra techniques, due to both time and space constraints. However, the equations generated are extremely *sparse*, typically with at most 50 non-zero coefficients per equation. The technique of *structured* or so-called *intelligent* Gaussian elimination (see Odlyzko [940]) can be used to reduce the original sparse system to a much smaller system that is still fairly sparse. The resulting system can be solved using either ordinary Gaussian elimination, or one of the *conjugate gradient*, *Lanczos* (Coppersmith, Odlyzko, and Schroeppel [280]), or *Wiedemann* algorithms [1239] which were also designed to handle sparse systems. LaMacchia and Odlyzko [737] have implemented some of these algorithms and concluded that the linear algebra stages arising in both integer factorization and the discrete logarithm problem are not running-time bottlenecks in practice. Recently, Coppersmith [272] proposed a modification of the Wiedemann algorithm which allows parallelization of the algorithm; for an analysis of Coppersmith's algorithm, see Kaltofen [657]. Coppersmith [270] (see also Montgomery [896]) presented a modification of the Lanczos algorithm for solving sparse linear equations over $\mathbb{F}_2$; this variant appears to be the most efficient in practice.

As an example of the numbers involved, Gordon and McCurley's [511] implementation for computing logarithms in $\mathbb{F}^*_{2^{401}}$ produced a total of 117164 equations from a factor base consisting of the 58636 irreducible polynomials in $\mathbb{F}_2[x]$ of degree at most 19. The system of equations had 2068707 non-zero entries. Structured Gaussian elimination was then applied to this system, the result being a 16139×16139 system of equations having 1203414 non-zero entries, which was then solved using the conjugate gradient method. Another example is from the recent factorization of the RSA-129 number (see Atkins et al. [59]). The sieving step produced a sparse matrix of 569466 rows and 524339 columns. Structured Gaussian elimination was used to reduce this to a dense 188614×188160 system, which was then solved using ordinary Gaussian elimination.

There are many ways of representing a finite field, although any two finite fields of the same order are isomorphic (see also Note 3.55). Lenstra [757] showed how to compute an isomorphism between any two explicitly given representations of a finite field in deterministic polynomial time. Thus, it is sufficient to find an algorithm for computing discrete logarithms in one representation of a given field; this algorithm can then be used, together with the isomorphism obtained by Lenstra's algorithm, to compute logarithms in any other representation of the same field.

Menezes, Okamoto, and Vanstone [843] showed how the discrete logarithm problem for an elliptic curve over a finite field $\mathbb{F}_q$ can be reduced to the discrete logarithm problem in some extension field $\mathbb{F}_{q^k}$. For the special class of *supersingular curves*, k is at most 6, thus providing a subexponential-time algorithm for the former problem. This work was extended by Frey and Rück [422]. No subexponential-time algorithm is known for the discrete logarithm problem in the more general class of *non-supersingular* elliptic curves.

Adleman, DeMarrais, and Huang [12] presented a subexponential-time algorithm for finding logarithms in the jacobian of large genus hyperelliptic curves over finite fields. More precisely, there exists a number c, $0 < c \le 2.181$, such that for all sufficiently large $g \ge 1$ and all odd primes p with $\log p \le (2g+1)^{0.98}$, the expected running time of the algorithm for computing logarithms in the jacobian of a genus g hyperelliptic curve over $\mathbb{Z}_p$ is conjectured to be $L_{p^{2g+1}}[\frac{1}{2}, c]$.

McCurley [826] invented a subexponential-time algorithm for the discrete logarithm problem in the class group of an imaginary quadratic number field. See also Hafner and McCurley [537] for further details, and Buchmann and Düllmann [216] for an implementation report.

In 1994, Shor [1128] conceived randomized polynomial-time algorithms for computing discrete logarithms and factoring integers on a *quantum computer*, a computational device based on quantum mechanical principles; presently it is not known how to build a quantum computer, nor if this is even possible. Also recently, Adleman [10] demonstrated the feasibility of using tools from molecular biology to solve an instance of the directed Hamiltonian path problem, which is **NP**-complete. The problem instance was encoded in molecules of DNA, and the steps of the computation were performed with standard protocols and enzymes. Adleman notes that while the currently available fastest supercomputers can execute approximately 10^{12} operations per second, it is plausible for a DNA computer to execute 10^{20} or more operations per second. Moreover such a DNA computer would be far more energy-efficient than existing supercomputers. It is not clear at present whether it is feasible to build a DNA computer with such performance. However, should either quantum computers or DNA computers ever become practical, they would have a very significant impact on public-key cryptography.

§3.7

Fact 3.77(i) is due to den Boer [323]. Fact 3.77(iii) was proven by Maurer [817], who also proved more generally that the GDHP and GDLP in a group G of order n are computationally equivalent when certain extra information of length $O(\lg n)$ bits is given. The extra information depends only on n and not on the definition of G, and consists of parameters that define cyclic elliptic curves of smooth order over the fields $\mathbb{Z}_{p_i}$ where the p_i are the prime divisors of n.

Waldvogel and Massey [1228] proved that if a and b are chosen uniformly and randomly from the interval $\{0, 1, \ldots, p-1\}$, the values $\alpha^{ab} \bmod p$ are roughly uniformly distributed (see page 537).

§3.8

Facts 3.78 and 3.79 are due to Bach [62]. Fact 3.80 is due to Shmuely [1127]. McCurley [825] refined this result to prove that for specially chosen composite n, the ability to solve the Diffie-Hellman problem in $\mathbb{Z}_n^*$ for the *fixed* base $\alpha = 16$ implies the ability to factor n.

§3.9

The notion of a hard Boolean predicate (Definition 3.81) was introduced by Blum and Micali [166], who also proved Fact 3.84. The notion of a hard k-bit predicate (Definition 3.82) was introduced by Long and Wigderson [772], who also proved Fact 3.85; see also Peralta [968]. Fact 3.83 is due to Peralta [968]. The results on hard predicates and k-bit predicates for the RSA functions (Facts 3.86 and 3.87) are due to Alexi et al. [23]. Facts 3.88 and 3.89 are due to Vazirani and Vazirani [1218].

Yao [1258] showed how any one-way length-preserving permutation can be transformed into a more complicated one-way length-preserving permutation which has a hard predicate. Subsequently, Goldreich and Levin [471] showed how any one-way function f can be transformed into a one-way function g which has a hard predicate. Their construction is as follows. Define the function g by $g(p, x) = (p, f(x))$, where p is a binary string of the same length as x, say n. Then g is also a one-way function and $B(p, x) = \sum_{i=1}^{n} p_i x_i \bmod 2$ is a hard predicate for g.

Håstad, Schrift, and Shamir [543] considered the one-way function $f(x) = \alpha^x \bmod n$, where n is a Blum integer and $\alpha \in \mathbb{Z}_n^*$. Under the assumption that factoring Blum integers is intractable, they proved that all the bits of this function are individually hard. Moreover, the lower half as well as the upper half of the bits are simultaneously secure.

§3.10

The subset sum problem (Definition 3.90) is sometimes confused with the *knapsack problem* which is the following: given two sets $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$ of positive integers, and given two positive integers s and t, determine whether or not there is a subset S of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in S} a_i \leq s$ and $\sum_{i \in S} b_i \geq t$. The subset sum problem is actually a special case of the knapsack problem when $a_i = b_i$ for $i = 1, 2, \ldots, n$ and $s = t$. Algorithm 3.94 is described by Odlyzko [941].

The L^3-lattice basis reduction algorithm (Algorithm 3.101) and Fact 3.103 are both due to Lenstra, Lenstra, and Lovász [750]. Improved algorithms have been given for lattice basis reduction, for example, by Schnorr and Euchner [1099]; consult also Section 2.6 of Cohen [263]. Algorithm 3.105 for solving the subset sum problem involving knapsacks sets of low density is from Coster et al. [283]. Unusually good simultaneous diophantine approximations were first introduced and studied by Lagarias [723]; Fact 3.107 and Algorithm 3.108 are from this paper.

§3.11

A readable introduction to polynomial factorization algorithms is given by Lidl and Niederreiter [764, Chapter 4]. Algorithm 3.110 for square-free factorization is from Geddes, Czapor, and Labahn [445]. Yun [1261] presented an algorithm that is more efficient than Algorithm 3.110 for finding the square-free factorization of a polynomial. The running time of the algorithm is only $O(n^2)$ $\mathbb{Z}_p$-operations when $f(x)$ is a polynomial of degree n in $\mathbb{Z}_p[x]$. A lucid presentation of Yun's algorithm is provided by Bach and Shallit [70]. Berlekamp's Q-matrix algorithm (Algorithm 3.111) was first discovered by Prange [999] for the purpose of factoring polynomials of the form $x^n - 1$ over finite fields. The algorithm was later and independently discovered by Berlekamp [117] who improved it for factoring general polynomials over finite fields.

There is no deterministic polynomial-time algorithm known for the problem of factoring polynomials over finite fields. There are, however, many efficient randomized algorithms that work well even when the underlying field is very large, such as the algorithms given by Ben-Or [109], Berlekamp [119], Cantor and Zassenhaus [232], and Rabin [1025]. For recent work along these lines, see von zur Gathen and Shoup [1224], as well as Kaltofen and Shoup [658].