Chapter 10

*Algorithms for Factoring and Computing Discrete Logarithms

In the last chapter, we introduced several number-theoretic problems—most prominently, *factoring* the product of two large primes and *computing discrete logarithms* in certain groups—that are widely believed to be hard. As defined there, this means there are presumed to be no polynomial-time algorithms for these problems. This *asymptotic* notion of hardness, however, tells us little about how to set the security parameter—sometimes called the *key length*, although the terms are not interchangeable—to achieve some desired, *concrete* level of security in practice. A proper understanding of this issue is extremely important for the real-world deployment of cryptosystems based on these problems. Setting the security parameter too low means a cryptosystem may be vulnerable to attacks more efficient than anticipated; being overly conservative and setting the security parameter too high will give good security, but at the expense of efficiency for the honest users. The relative difficulty of different number-theoretic problems can also play a role in determining which problems to use as the basis for building cryptosystems in the first place.

The fundamental issue, of course, is that a brute-force search may not be the best algorithm for solving a given problem; thus, using key length n does not, in general, give security against attackers running for 2^n time. This is in contrast to the private-key setting where the best attacks on existing block ciphers have roughly the complexity of brute-force search. As a consequence, the key lengths used in the public-key setting tend to be significantly larger than those used in the private-key setting.

To gain a better appreciation of this point, we explore in this chapter several algorithms for factoring and computing discrete logarithms that do not run in polynomial time, but nevertheless perform far better than brute-force search. The goal is merely to give a taste of existing algorithms for these problems, as well as to provide some basic guidance for setting parameters in practice. Our focus is on the high-level ideas, and we consciously do not address many important implementation-level details that would be critical to deal with if these algorithms were to be used in practice. We also concentrate exclusively on *classical* algorithms here, deferring a discussion about the effect of *quantum* algorithms to Chapter 14.

The reader may also notice that we only describe algorithms for factoring and computing discrete logarithms, and not algorithms for, say, solving the

RSA or decisional Diffie–Hellman problems. Our choice is justified by the facts that the best known algorithms for solving RSA require factoring the modulus, and (in the groups discussed in Sections 9.3.3 and 9.3.4) the best known approaches for solving the decisional Diffie–Hellman problem require computing discrete logarithms.

10.1 Algorithms for Factoring

Throughout this chapter, we assume that $N = pq$ is a product of two distinct primes with $p < q$. We will be most interested in the case when p and q each has the same (known) length n, and so $n = \Theta(\log N)$.

We will frequently use the Chinese remainder theorem along with the notation developed in Section 9.1.5. The Chinese remainder theorem states that

$$\mathbb{Z}_N \simeq \mathbb{Z}_p \times \mathbb{Z}_q \quad \text{and} \quad \mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*,$$

with isomorphism given by $f(x) \stackrel{\text{def}}{=} ([x \bmod p], [x \bmod q])$. The fact that f is an isomorphism means, in particular, that it gives a bijection between elements $x \in \mathbb{Z}_N$ and pairs $(x_p, x_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$. We write $x \leftrightarrow (x_p, x_q)$ to denote this bijection, with $x_p = [x \bmod p]$ and $x_q = [x \bmod q]$.

Recall from Section 9.2 that *trial division*—a trivial, brute-force factoring method—finds a factor of a given number N in time $\mathcal{O}(N^{1/2} \cdot \mathsf{polylog}(N))$. (This is an exponential-time algorithm, since the size of the input is $\|N\|$, the length of the binary representation of N, and $\|N\| = \mathcal{O}(\log N)$.[1]) We show here three factoring algorithms with better performance:

- *Pollard's $p-1$ method* is effective if $p-1$ has only "small" prime factors.
- *Pollard's rho method* applies to arbitrary N. (As such, it is called a *general-purpose* factoring algorithm.) Its running time for N of the form discussed at the beginning of this section is $\mathcal{O}(N^{1/4} \cdot \mathsf{polylog}(N))$. Note this is still *exponential* in n, the length of N.
- The *quadratic sieve algorithm* is a general-purpose factoring algorithm that runs in time *sub-exponential* in the length of N. We give a high-level overview of how this algorithm works, but the details are somewhat complex and beyond the scope of this book.

The fastest known general-purpose factoring algorithm is the *general number field sieve*. Heuristically, this algorithm factors its input N in expected time $2^{\mathcal{O}((\log N)^{1/3} \cdot (\log \log N)^{2/3})}$, which is sub-exponential in the length of N.

[1] Thus, a running time of $N^{\mathcal{O}(1)} = 2^{\mathcal{O}(\|N\|)}$ is exponential, a running time of $2^{o(\log N)} = 2^{o(\|N\|)}$ is sub-exponential, and a running time of $(\log N)^{\mathcal{O}(1)} = \|N\|^{\mathcal{O}(1)}$ is polynomial.

10.1.1 Pollard's $p-1$ Algorithm

If $N = pq$ and $p-1$ has *only* "small" prime factors, Pollard's $p-1$ algorithm can be used to efficiently factor N. The basic idea is simple. Let B be an integer for which $(p-1) \mid B$ and $(q-1) \nmid B$; we defer to below the details of how such a B is computed. Say $B = \gamma \cdot (p-1)$ for some integer γ. Choose a uniform $x \in \mathbb{Z}_N^*$ and compute $y := [x^B - 1 \bmod N]$. (Note that y can be computed using the efficient exponentiation algorithm from Appendix B.2.3.) Since $1 \leftrightarrow (1,1)$, we have

$$\begin{aligned} y = [x^B - 1 \bmod N] &\leftrightarrow (x_p, x_q)^B - (1,1) \\ &= (x_p^B - 1 \bmod p,\ x_q^B - 1 \bmod q) \\ &= ((x_p^{p-1})^\gamma - 1 \bmod p,\ x_q^B - 1 \bmod q) \\ &= (0, [x_q^B - 1 \bmod q]), \end{aligned}$$

using Theorem 9.14 and the fact that the order of $\mathbb{Z}_p^*$ is $p-1$. We show below that, with high probability, $x_q^B \neq 1 \bmod q$. Assuming this is the case, we have obtained an integer $y \in \mathbb{Z}_N^*$ for which

$$y = 0 \bmod p \quad \text{but} \quad y \neq 0 \bmod q;$$

that is, $p \mid y$ but $q \nmid y$. This, in turn, implies that $\gcd(y, N) = p$. Thus, a simple gcd computation (which can be done efficiently as described in Appendix B.1.2) yields a prime factor of N.

ALGORITHM 10.1
Pollard's $p-1$ algorithm for factoring

Input: Integer N
Output: A nontrivial factor of N

$x \leftarrow \mathbb{Z}_N^*$
$y := [x^B - 1 \bmod N]$
  // B is as in the text
$p := \gcd(y, N)$
if $p \notin \{1, N\}$ **return** p

We now argue that the algorithm works with high probability. Because $(q-1) \nmid B$, as long as $x_q \stackrel{\text{def}}{=} [x \bmod q]$ is a generator of $\mathbb{Z}_q^*$ we must have $x_q^B \neq 1 \bmod q$. (This follows from Proposition 9.53.) It remains to analyze the probability that x_q is a generator. Here we rely on some results proved in Appendix B.3.1. Since q is prime, $\mathbb{Z}_q^*$ is a cyclic group of order $q-1$ that has exactly $\phi(q-1)$ generators (cf. Theorem B.16). If x is chosen uniformly from $\mathbb{Z}_N^*$, then x_q is uniformly distributed in $\mathbb{Z}_q^*$. (This is a consequence of the fact that the Chinese remainder theorem gives a bijection between $\mathbb{Z}_N^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.) Thus, the probability that x_q is a generator is $\frac{\phi(q-1)}{q-1} =$

$\Omega(1/\log q) = \Omega(1/n)$ (cf. Theorem B.15). Multiple values of x can be chosen to boost the probability of success.

We are left with the problem of finding B such that $(p-1) \mid B$ but $(q-1) \nmid B$. One possibility is to choose $B = \prod_{i=1}^{k} p_i^{\lfloor n/\log p_i \rfloor}$ for some k, where p_i denotes the ith prime (i.e., $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$) and n is the length of p. (Note that $p_i^{\lfloor n/\log p_i \rfloor}$ is the largest power of p_i that can possibly divide $p-1$.) If $p-1$ can be written as $\prod_{i=1}^{k} p_i^{e_i}$ with $e_i \geq 0$ (that is, if the largest prime factor of $p-1$ is less than p_k), then it will hold that $(p-1) \mid B$. In contrast, if $q-1$ has *any* prime factor larger than p_k, then $(q-1) \nmid B$.

Choosing a larger value for k increases B and so increases the running time of the algorithm (which performs a modular exponentiation to the power B). A larger value of k also makes it more likely that $(p-1) \mid B$, but at the same time makes it less likely that $(q-1) \nmid B$. It is, of course, possible to run the algorithm repeatedly using multiple choices for k.

Pollard's $p-1$ algorithm is thwarted if both $p-1$ and $q-1$ have any large prime factors. (More precisely, the algorithm still works but only for B so large that the algorithm becomes impractical.) For this reason, when generating a modulus $N = pq$ for cryptographic applications, p and q are sometimes chosen to be *strong* primes, namely, with $(p-1)/2$ and $(q-1)/2$ themselves prime. This ensures that both $p-1$ and $q-1$ have a large prime factor, and so the resulting modulus will not be vulnerable to Algorithm 10.1. Selecting p and q in this way is markedly less efficient than choosing p and q as *arbitrary* primes. Moreover, if p and q are uniform n-bit primes, it is unlikely that either $p-1$ or $q-1$ will have only small prime factors and so unlikely that Algorithm 10.1 will apply. Finally, better factoring algorithms are available anyway (as we will see below). For these reasons, the current consensus is that the added computational cost of generating p and q as strong primes does not yield any appreciable security gains.

10.1.2 Pollard's Rho Algorithm

In contrast to Algorithm 10.1, which is only effective for certain moduli, Pollard's *rho algorithm* can be used to factor an arbitrary integer $N = pq$; in that sense, it is a *general-purpose* factoring algorithm. Heuristically, the algorithm factors N with constant probability in $\mathcal{O}\left(N^{1/4} \cdot \mathsf{polylog}(N)\right)$ time; this is still exponential, but a vast improvement over trial division.

The core idea of the approach is to find distinct values $x, x' \in \mathbb{Z}_N^*$ that are equivalent modulo p (i.e., for which $x = x' \bmod p$); call such a pair *good*. Note that for a good pair x, x' it holds that $\gcd(x - x', N) = p$ (since $x \neq x' \bmod N$), so computing the gcd gives a nontrivial factor of N.

How can we find a good pair? Say we choose values $x^{(1)}, \ldots, x^{(k)}$ uniformly from $\mathbb{Z}_N^*$, where $k = 2^{n/2} = \mathcal{O}(\sqrt{p})$. Viewing these in their Chinese-remaindering representation as $(x_p^{(1)}, x_q^{(1)}), \ldots, (x_p^{(k)}, x_q^{(k)})$, we have that each $x_p^{(i)} \stackrel{\text{def}}{=} [x^{(i)} \bmod p]$ is uniform in $\mathbb{Z}_p^*$. (This follows from bijectivity between

$\mathbb{Z}_N^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.) Thus, using the birthday bound of Lemma A.15, we see that with high probability there exist distinct i, j with $x_p^{(i)} = x_p^{(j)}$ or, equivalently, $x^{(i)} = x^{(j)} \bmod p$. Moreover, Lemma A.15 shows that $x^{(i)} \neq x^{(j)}$ except with negligible probability. Thus, with high probability we obtain a good pair $x^{(i)}, x^{(j)}$ that can be used to find a nontrivial factor of N, as discussed earlier.

ALGORITHM 10.2
Pollard's rho algorithm for factoring

Input: Integer N, a product of two n-bit primes
Output: A nontrivial factor of N

$x \leftarrow \mathbb{Z}_N^*, \quad x' := x$
for $i = 1$ to $2^{n/2}$:
  $x := F(x)$
  $x' := F(F(x'))$
  $p := \gcd(x - x', N)$
  if $p \notin \{1, N\}$ **return** p and **stop**

We can generate $k = \mathcal{O}(\sqrt{p})$ uniform elements of $\mathbb{Z}_N^*$ in $\mathcal{O}(\sqrt{p}) = \mathcal{O}(N^{1/4})$ time. *Testing* all pairs of elements in order to identify a good pair, however, would require $\binom{k}{2} = \mathcal{O}(k^2) = \mathcal{O}(p) = \mathcal{O}\left(N^{1/2}\right)$ time! (Note that since p is unknown we cannot simply compute $x_p^{(1)}, \ldots, x_p^{(k)}$ explicitly and then sort the $x_p^{(i)}$ to find a good pair. Instead, for all distinct pairs i, j we must compute $\gcd(x^{(i)} - x^{(j)}, N)$ to see whether this gives a nontrivial factor of N.) Without further optimizations, this will be no better than trial division.

Pollard's idea was to use a technique we have seen in Section 6.4.2 in the context of small-space birthday attacks. Specifically, we compute the sequence $x^{(1)}, x^{(2)}, \ldots$ by letting each value be a function of the one before it, i.e., we fix some function $F : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$, choose a uniform $x^{(0)} = x \in \mathbb{Z}_N^*$, and then set $x^{(i)} := F(x^{(i-1)})$ for $i = 1, \ldots, k$. We require F to have the property that if $x = x' \bmod p$, then $F(x) = F(x') \bmod p$; this ensures that once equivalence modulo p occurs, it persists. (A standard choice is $F(x) = [x^2 + 1 \bmod N]$, but any polynomial modulo N will have this property.) If we heuristically model F as a random function, then with high probability there is a good pair in the first k elements of this sequence. Proceeding roughly as in Algorithm 6.9 from Section 6.4.2, we can detect a good pair (if there is one) using only $\mathcal{O}(k)$ gcd computations; see Algorithm 10.2.

10.1.3 The Quadratic Sieve Algorithm

Pollard's rho algorithm is better than trial division, but still runs in exponential time. The *quadratic sieve* algorithm runs in sub-exponential time. It was the fastest known factoring algorithm until the early 1990s and remains

the factoring algorithm of choice for numbers up to about 300 bits long. We describe the general principles of the algorithm but caution the reader that several important details are omitted.

An element $z \in \mathbb{Z}_N^*$ is a *quadratic residue modulo* N if there is an $x \in \mathbb{Z}_N^*$ such that $x^2 = z \bmod N$; in this case, we say that x is a *square root of* z. The following observations serve as our starting point:

- If N is a product of two distinct, odd primes, then every quadratic residue modulo N has exactly four square roots. (See Section 15.4.2.)

- Given x, y with $x^2 = y^2 \bmod N$ and $x \neq \pm y \bmod N$, it is possible to compute a nontrivial factor of N in polynomial time. This is by virtue of the fact that $x^2 = y^2 \bmod N$ implies

$$0 = x^2 - y^2 = (x - y)(x + y) \bmod N,$$

  and so $N \mid (x - y)(x + y)$. However, $N \nmid (x - y)$ and $N \nmid (x + y)$ because $x \neq \pm y \bmod N$. So it must be the case that $\gcd(x - y, N)$ is equal to one of the prime factors of N. (See also Lemma 15.35.)

The quadratic sieve algorithm tries to generate x, y with $x^2 = y^2 \bmod N$ and $x \neq \pm y \bmod N$. A naive way of doing this—which forms the basis of an older factoring algorithm due to Fermat—is to choose an $x \in \mathbb{Z}_N^*$, compute $q := [x^2 \bmod N]$, and then check whether q is a square *over the integers* (i.e., without reduction modulo N). If so, then $q = y^2$ for some integer y and so $x^2 = y^2 \bmod N$. Unfortunately, the probability that $[x^2 \bmod N]$ is a square is so low that this process must be repeated exponentially many times.

A significant improvement is obtained by generating a sequence of values $q_1 := [x_1^2 \bmod N], \ldots$ and identifying a subset of those values whose *product* is a square over the integers. In the quadratic sieve algorithm this is accomplished using the following two steps:

Step 1. Fix some bound B. Say an integer is B-*smooth* if all its prime factors are less than or equal to B. In the first phase of the algorithm, we search for integers of the form $q_i = [x_i^2 \bmod N]$ that are B-smooth and factor them. (Although factoring is hard, finding and factoring B-smooth numbers is feasible when B is small enough.) These $\{x_i\}$ are chosen by successively trying $x = \sqrt{N}+1, \sqrt{N}+2, \ldots$; this ensures a nontrivial reduction modulo N (since $x > \sqrt{N}$) and has the advantage that $q \stackrel{\text{def}}{=} [x^2 \bmod N] = x^2 - N$ is "small" so that q is more likely to be B-smooth.

Let $\{p_1, \ldots, p_k\}$ be the set of prime numbers less than or equal to B. Once we have found and factored the B-smooth $\{q_i\}$ as described above, we have a

set of equations of the form:

$$\begin{aligned} q_1 &= [x_1^2 \bmod N] = \prod_{i=1}^{k} p_i^{e_{1,i}} \\ &\vdots \\ q_\ell &= [x_\ell^2 \bmod N] = \prod_{i=1}^{k} p_i^{e_{\ell,i}}. \end{aligned} \tag{10.1}$$

(Note that the above equations are over the integers.)

Step 2. We next want to find some subset of the $\{q_i\}$ whose product is a square. If we multiply some subset S of the $\{q_i\}$, we see that the result

$$z = \prod_{j \in S} q_j = \prod_{i=1}^{k} p_i^{\sum_{j \in S} e_{j,i}}$$

is a square if and only if the exponent of each prime p_i is even. This suggests that we care about the exponents $\{e_{j,i}\}$ in Equation (10.1) only modulo 2; moreover, we can use linear algebra to find a subset of the $\{q_i\}$ whose "exponent vectors" sum to the 0-vector modulo 2.

In more detail: if we reduce the exponents in Equation (10.1) modulo 2, we obtain the 0/1-matrix Γ given by

$$\begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{\ell,1} & \gamma_{\ell,2} & \cdots & \gamma_{\ell,k} \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} [e_{1,1} \bmod 2] & [e_{1,2} \bmod 2] & \cdots & [e_{1,k} \bmod 2] \\ \vdots & \vdots & \ddots & \vdots \\ [e_{\ell,1} \bmod 2] & [e_{\ell,2} \bmod 2] & \cdots & [e_{\ell,k} \bmod 2] \end{pmatrix}.$$

If $\ell = k + 1$, then Γ has more rows than columns and there must be some nonempty subset S of the rows that sum to the 0-vector modulo 2. Such a subset can be found efficiently using linear algebra. Then:

$$z \stackrel{\text{def}}{=} \prod_{j \in S} q_j = \prod_{i=1}^{k} p_i^{\sum_{j \in S} e_{j,i}} = \left(\prod_{i=1}^{k} p_i^{(\sum_{j \in S} e_{j,i})/2} \right)^2,$$

using the fact that all the $\left\{ \sum_{j \in S} e_{j,i} \right\}$ are even. Since

$$z = \prod_{j \in S} q_j = \prod_{j \in S} x_j^2 = \left(\prod_{j \in S} x_j \right)^2 \bmod N,$$

we have obtained two square roots (modulo N) of z. Although there is no guarantee that these square roots will enable factorization of N (for reasons

discussed at the beginning of this section), heuristically they do with constant probability. By taking $\ell > k + 1$ we can obtain multiple subsets S with the desired property and try to factor N using each possibility.

Example 10.3
Take $N = 377753$. We have $6647 = [620^2 \bmod N]$, and we can factor 6647 (over the integers, without any modular reduction) as

$$\left[620^2 \bmod N\right] = 6647 = 17^2 \cdot 23.$$

Similarly,

$$\begin{aligned}\left[621^2 \bmod N\right] &= 2^4 \cdot 17 \cdot 29\\ \left[645^2 \bmod N\right] &= 2^7 \cdot 13 \cdot 23\\ \left[655^2 \bmod N\right] &= 2^3 \cdot 13 \cdot 17 \cdot 29.\end{aligned}$$

Letting our subset S include all four of the above equations, we see that

$$\begin{aligned}620^2 \cdot 621^2 \cdot 645^2 \cdot 655^2 &= 2^{14} \cdot 13^2 \cdot 17^4 \cdot 23^2 \cdot 29^2 \bmod N\\ \Rightarrow [620 \cdot 621 \cdot 645 \cdot 655 \bmod N]^2 &= \left[2^7 \cdot 13 \cdot 17^2 \cdot 23 \cdot 29 \bmod N\right]^2 \bmod N\\ \Rightarrow 127194^2 &= 45335^2 \bmod N,\end{aligned}$$

with $127194 \neq \pm 45335 \bmod N$. Computing $\gcd(127194 - 45335, 377753) = 751$ yields a nontrivial factor of N. ♢

Running time. Choosing a larger value of B makes it more likely that a uniform value $q = [x^2 \bmod N]$ is B-smooth; on the other hand, it means we will have to work harder to identify and factor B-smooth numbers, and we will have to find more of them (since we require $\ell > k$, where k is the number of primes less than or equal to B). It also means that the matrix Γ will be larger, and so the linear-algebraic step will be slower. Choosing the optimal value of B gives an algorithm that (heuristically, at least) factors N in time $2^{\mathcal{O}(\sqrt{\log N \log\log N})}$. (In fact, the constant term in the exponent can be determined quite precisely.) The important point for our purposes is that this is sub-exponential in the length of N.

10.2 Algorithms for Computing Discrete Logarithms

Let $\mathbb{G}$ be a cyclic group of known order q. An instance of the discrete-logarithm problem in $\mathbb{G}$ specifies a generator $g \in \mathbb{G}$ and an element $h \in \mathbb{G}$; the goal is to find $x \in \mathbb{Z}_q$ such that $g^x = h$. (See Section 9.3.2.) The solution

x is called the *discrete logarithm of h with respect to g*. A trivial brute-force search for x can be done in time $\mathcal{O}(q)$, and so we are interested in algorithms whose running time is better than this.

Algorithms for solving the discrete-logarithm problem fall into two categories: those that are *generic* and apply to any group $\mathbb{G}$, and those that are tailored to work for some *specific* class of groups. We begin in this section by discussing three generic algorithms:

- When the group order q is not prime and a (partial or full) factorization of q is known, the *Pohlig–Hellman algorithm* reduces the problem of finding discrete logarithms in $\mathbb{G}$ to that of finding discrete logarithms in *subgroups* of $\mathbb{G}$. When the complete factorization of q is known, the effect is to reduce the complexity of computing discrete logarithms in a group of order q to the complexity of computing discrete logarithms in a group of order q', where q' is the largest prime dividing q. This explains the preference for using prime-order groups (cf. Section 9.3.2).

- The *baby-step/giant-step* method, due to Shanks, computes the discrete logarithm in a group of order q using $\mathcal{O}(\sqrt{q})$ group operations. It also requires $\mathcal{O}(\sqrt{q})$ memory.

- *Pollard's rho algorithm* also computes discrete logarithms with $\mathcal{O}(\sqrt{q})$ group operations, but using *constant* memory. It can be viewed as exploiting the connection between the discrete-logarithm problem and collision-resistant hashing that we have seen in Section 9.4.2.

It can be shown that the time complexity of the latter two algorithms is *optimal* as far as generic algorithms are concerned. Thus, to have any hope of doing better we must look at algorithms for specific groups that exploit the *binary representation* of elements in those groups, i.e., the way group elements are encoded as bit-strings. This point bears some discussion. From a mathematical point of view, any two cyclic groups of the same order are isomorphic, meaning that the groups are identical up to a "renaming" of the group elements. From a computational/algorithmic point of view, however, this "renaming" can have a significant impact. For example, consider the cyclic group $\mathbb{Z}_q$ of integers $\{0, \ldots, q-1\}$ under *addition* modulo q. Computing discrete logarithms in this group is trivial: Say we are given $g, h \in \mathbb{Z}_q$ with g a generator, and we want to find x such that $x \cdot g = h \bmod q$. We must have $\gcd(g, q) = 1$ (cf. Theorem B.16) and so g has a multiplicative inverse g^{-1} modulo q. Moreover, g^{-1} can be computed efficiently, as described in Appendix B.2.2. But then $x = h \cdot g^{-1} \bmod q$ is the desired solution. Note that, formally, x here denotes an integer and not a group element—after all, the group operation is addition, not multiplication. Nevertheless, in solving the discrete-logarithm problem in $\mathbb{Z}_q$ we can make use of the fact that another operation (namely, multiplication) can be defined on the elements of that group. The main takeaway point is that *the group representation matters*.

Turning to groups with cryptographic significance, in Section 10.3 we focus our attention on (subgroups of) $\mathbb{Z}_p^*$ for p prime. (See Section 9.3.3.) As a nontrivial example of an algorithm that is not generic, we give a high-level overview of the *index calculus algorithm* for solving the discrete-logarithm problem in such groups in sub-exponential time. Currently, the best known algorithm for this class of groups is the *general number field sieve*,[2] which heuristically runs in time $2^{\mathcal{O}((\log p)^{1/3} \cdot (\log \log p)^{2/3})}$. Sub-exponential algorithms for computing discrete logarithms in multiplicative subgroups of arbitrary finite fields are also known, but these are beyond our scope.

Importantly, no sub-exponential algorithms are known for computing discrete logarithms in general elliptic-curve groups. This explains why smaller parameters can be used (at the same level of security) when working in elliptic-curve groups than when working in $\mathbb{Z}_p^*$, resulting in more-efficient cryptosystems in the former case.

10.2.1 The Pohlig–Hellman Algorithm

The Pohlig–Hellman algorithm can be used to speed up the computation of discrete logarithms in a group $\mathbb{G}$ when any nontrivial factors of the group order q are known. Recall that the order of an element g, which we denote here by $\mathsf{ord}(g)$, is the smallest positive integer i for which $g^i = 1$. We will need the following lemma:

LEMMA 10.4 *Let* $\mathsf{ord}(g) = q$, *and say* $p \mid q$. *Then* $\mathsf{ord}(g^p) = q/p$.

PROOF Since $(g^p)^{q/p} = g^q = 1$, the order of g^p is at most q/p. Let $i > 0$ be such that $(g^p)^i = 1$. Then $g^{pi} = 1$ and, since q is the order of g, we must have $pi \geq q$ or equivalently $i \geq q/p$. The order of g^p is thus exactly q/p. ■

We will also use a generalization of the Chinese remainder theorem: if $q = \prod_{i=1}^{k} q_i$ and $\gcd(q_i, q_j) = 1$ for all $i \neq j$ then

$$\mathbb{Z}_q \simeq \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k} \quad \text{and} \quad \mathbb{Z}_q^* \simeq \mathbb{Z}_{q_1}^* \times \cdots \times \mathbb{Z}_{q_k}^*.$$

(This can be proved by induction on k, using the basic Chinese remainder theorem for $k = 2$.) Moreover, by an extension of the algorithm in Section 9.1.5 it is possible to convert efficiently between the representation of an element as an element of $\mathbb{Z}_q$ and its representation as an element of $\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k}$ when the factorization $q = \prod_{i=1}^{k} q_i$ is known.

We now describe the Pohlig–Hellman algorithm. We are given a generator g and an element h and wish to find x such that $g^x = h$. Say a factorization

[2] The algorithm is related to the general number field sieve for factoring.

$q = \prod_{i=1}^{k} q_i$ is known with the $\{q_i\}$ pairwise relatively prime. (This need not be the complete prime factorization of q.) We know that

$$\left(g^{q/q_i}\right)^x = (g^x)^{q/q_i} = h^{q/q_i} \quad \text{for } i = 1, \ldots, k. \tag{10.2}$$

Letting $g_i \stackrel{\text{def}}{=} g^{q/q_i}$ and $h_i \stackrel{\text{def}}{=} h^{q/q_i}$, we thus have k instances of a discrete-logarithm problem in k *smaller* groups. Specifically, each problem $g_i^x = h_i$ is in a subgroup of size $\mathsf{ord}(g_i) = q_i$ (by Lemma 10.4). We can solve each of the k resulting instances using any algorithm for solving the discrete-logarithm problem. Solving these instances gives a set of answers $\{x_i\}_{i=1}^{k}$, with $x_i \in \mathbb{Z}_{q_i}$, for which $g_i^{x_i} = h_i = g_i^x$. Proposition 9.54 implies that $x = x_i \bmod q_i$ for all i. By the generalized Chinese remainder theorem discussed earlier, the constraints

$$\begin{aligned} x &= x_1 \bmod q_1 \\ &\vdots \\ x &= x_k \bmod q_k \end{aligned}$$

uniquely determine x modulo q, and so the desired solution x can be efficiently reconstructed from the $\{x_i\}$.

Example 10.5

Consider the problem of computing discrete logarithms in $\mathbb{Z}_{31}^*$, a group of order $q = 30 = 5 \cdot 3 \cdot 2$. Say $g = 3$ and $h = 26 = g^x$ with x unknown. We have:

$$\begin{aligned} (g^{30/5})^x = h^{30/5} &\Rightarrow (3^6)^x = 26^6 \Rightarrow 16^x = 1 \\ (g^{30/3})^x = h^{30/3} &\Rightarrow (3^{10})^x = 26^{10} \Rightarrow 25^x = 5 \\ (g^{30/2})^x = h^{30/2} &\Rightarrow (3^{15})^x = 26^{15} \Rightarrow 30^x = 30. \end{aligned}$$

(All the above equations are modulo 31.) We have $\mathsf{ord}(16) = 5$, $\mathsf{ord}(25) = 3$, and $\mathsf{ord}(30) = 2$. Solving each equation, we obtain

$$x = 0 \bmod 5, \quad x = 2 \bmod 3, \quad \text{and} \quad x = 1 \bmod 2,$$

and so $x = 5 \bmod 30$. Indeed, $3^5 = 26 \bmod 31$. ♢

If q has (known) prime factorization $q = \prod_{i=1}^{k} p_i^{e_i}$ then, by using the Pohlig–Hellman algorithm, the time to compute discrete logarithms in a group of order q is dominated by the computation of a discrete logarithm in a subgroup of size $\max_i\{p_i^{e_i}\}$. This can be further reduced to computation of a discrete logarithm in a subgroup of size $\max_i\{p_i\}$; see Exercise 10.5.

10.2.2 The Baby-Step/Giant-Step Algorithm

The baby-step/giant-step algorithm computes discrete logarithms in a group of order q using $\mathcal{O}(\sqrt{q})$ group operations. The idea is simple. Given a generator $g \in \mathbb{G}$, we can imagine the powers of g as forming a cycle

$$1 = g^0,\ g^1,\ g^2,\ \ldots,\ g^{q-2},\ g^{q-1},\ g^q = 1.$$

We know that h must lie somewhere in this cycle. Computing all the points in this cycle to find h would take $\Omega(q)$ time. Instead, we "mark off" the cycle at intervals of size $t \stackrel{\text{def}}{=} \lfloor \sqrt{q} \rfloor$; more precisely, we compute and store the $\lfloor q/t \rfloor + 1 = \mathcal{O}(\sqrt{q})$ elements

$$g^0, g^t, g^{2t}, \ldots, g^{\lfloor q/t \rfloor \cdot t}.$$

(These are the "giant steps.") Note that the gap between any consecutive "marks" (wrapping around at the end) is at most t. Furthermore, we know that $h = g^x$ lies in one of these gaps. Thus, if we take "baby steps" and compute the t elements

$$h \cdot g^1,\ \ldots,\ h \cdot g^t,$$

each of which corresponds to a "shift" of h, we know that one of these values will be equal to one of the marked points. Say we find $h \cdot g^i = g^{k \cdot t}$. We can then easily compute $\log_g h := [(kt - i) \bmod q]$. Pseudocode for this algorithm follows.

ALGORITHM 10.6
The baby-step/giant-step algorithm

Input: Elements $g, h \in \mathbb{G}$; the order q of $\mathbb{G}$
Output: $\log_g h$

$t := \lfloor \sqrt{q} \rfloor$
for $i = 0$ to $\lfloor q/t \rfloor$:
  compute $g_i := g^{i \cdot t}$
sort the pairs (i, g_i) by their second component
for $i = 1$ to t:
  compute $h_i := h \cdot g^i$
  if $h_i = g_k$ for some k, **return** $[(kt - i) \bmod q]$

The algorithm requires $\mathcal{O}(\sqrt{q})$ exponentiations/multiplications in $\mathbb{G}$. (In fact, after computing $g_1 = g^t$, each subsequent value g_i can be computed using a single multiplication as $g_i := g_{i-1} \cdot g_1$. Similarly, each h_i can be computed as $h_i := h_{i-1} \cdot g$.) Sorting the $\mathcal{O}(\sqrt{q})$ pairs $\{(i, g_i)\}$ takes time $\mathcal{O}(\sqrt{q} \cdot \log q)$, and we can then use binary search to check if each h_i is equal to some g_k in time $\mathcal{O}(\log q)$. The overall algorithm thus runs in time $\mathcal{O}(\sqrt{q} \cdot \mathsf{polylog}(q))$.

Example 10.7
We show an application of the algorithm in the cyclic group $\mathbb{Z}_{29}^*$ of order $q = 29 - 1 = 28$. Take $g = 2$ and $h = 17$. We set $t = 5$ and compute:

$$2^0 = 1, \quad 2^5 = 3, \quad 2^{10} = 9, \quad 2^{15} = 27, \quad 2^{20} = 23, \quad 2^{25} = 11.$$

(It should be understood that all operations are in $\mathbb{Z}_{29}^*$.) Then compute:

$$17 \cdot 2^1 = 5, \quad 17 \cdot 2^2 = 10, \quad 17 \cdot 2^3 = 20, \quad 17 \cdot 2^4 = 11,$$

and notice that $17 \cdot 2^4 = 11 = 2^{25}$. We thus have $\log_2 17 = 25 - 4 = 21$. ♢

10.2.3 Discrete Logarithms from Collisions

A drawback of the baby-step/giant-step algorithm is that it uses a large amount of memory, as it requires storage of $\mathcal{O}(\sqrt{q})$ points. We can obtain an algorithm that uses constant memory—and has the same asymptotic running time—by exploiting the connection between the discrete-logarithm problem and collision-resistant hashing shown in Section 9.4.2, and recalling the small-space birthday attack for finding collisions from Section 6.4.2.

We describe the high-level idea. Fix a generator $g \in \mathbb{G}$ and an element h. If we define the hash function $H_{g,h} : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{G}$ by $H_{g,h}(x_1, x_2) = g^{x_1} h^{x_2}$, then finding a collision in $H_{g,h}$ implies the ability to compute $\log_g h$ (cf. Lemma 9.65 and Theorem 9.79). We have thus reduced the problem of computing $\log_g h$ to that of finding a collision in a hash function, something we know how to do in time $\mathcal{O}(\sqrt{|\mathbb{G}|}) = \mathcal{O}(\sqrt{q})$ using a birthday attack! Moreover, a small-space birthday attack will give a collision in the same time and constant space.

It only remains to address a few technical details. One is that the small-space birthday attack described in Section 6.4.2 assumes that the range of the hash function is a subset of its domain; that is not the case here, and in fact (depending on the representation being used for elements of $\mathbb{G}$) it could even be that $H_{g,h}$ is not compressing. A second issue is that the analysis in Section 6.4.2 treated the hash function as a random function, whereas $H_{g,h}$ has a significant amount of algebraic structure.

Pollard's rho algorithm provides one way to deal with these issues. We describe a different algorithm that can be viewed as a more direct implementation of the above ideas. (In practice, Pollard's algorithm would be more efficient, although both algorithms use only $\mathcal{O}(\sqrt{q})$ group operations.) Let $F : \mathbb{G} \to \mathbb{Z}_q \times \mathbb{Z}_q$ denote a cryptographic hash function obtained by, e.g., a suitable modification of SHA-2. Define $H : \mathbb{G} \to \mathbb{G}$ by $H(k) \stackrel{\text{def}}{=} H_{g,h}(F(k))$. We can use Algorithm 6.9, with natural modifications, to find a collision in H using an expected $\mathcal{O}(\sqrt{|\mathbb{G}|}) = \mathcal{O}(\sqrt{q})$ evaluations of H (and constant memory). With overwhelming probability, this yields a collision in $H_{g,h}$. You are asked to flesh out the details in Exercise 10.7.

It is interesting to observe here a certain duality: the proof that hardness of the discrete-logarithm implies a collision-resistant hash function leads to a better algorithm for solving the discrete-logarithm problem! A little reflection should convince us that this is not surprising: a proof by reduction demonstrates that an attack on some construction (in this case, finding collisions in the hash function) directly yields an attack on the underlying assumption (here, the hardness of the discrete-logarithm problem), which is exactly the property exploited by the above algorithm.

10.3 Index Calculus

We conclude with a brief look at the (non-generic) *index calculus algorithm* for computing discrete logarithms in the cyclic group $\mathbb{Z}_p^*$ (for p prime). In contrast to the preceding (generic) algorithms, this approach has running time *sub-exponential* in the size of the group. The algorithm bears some resemblance to the quadratic sieve algorithm introduced in Section 10.1.3, and we assume readers are familiar with the discussion there. As in that case, we discuss the main ideas of the index calculus method but leave a detailed analysis outside the scope of our treatment. Also, some simplifications are introduced to clarify the presentation.

As in the quadratic sieve algorithm, the index calculus method uses a two-step process. Importantly, the first step requires knowledge only of the modulus p and the base g and so it can be run as a preprocessing step before h—the value whose discrete logarithm we wish to compute—is known. For the same reason, it suffices to run the first step only once in order to solve multiple instances of the discrete-logarithm problem (as long as all those instances share the same p and g).

Step 1. Fix some bound B, and let $\{p_1, \ldots, p_k\}$ be the set of prime numbers less than or equal to B. In this step, we find $\ell \geq k$ distinct values $x_1, \ldots, x_\ell \in \mathbb{Z}_{p-1}$ for which $g_i \stackrel{\text{def}}{=} [g^{x_i} \bmod p]$ is B-smooth. This is done by simply choosing uniform $\{x_i\}$ until suitable values are found.

Factoring the resulting B-smooth numbers, we have the ℓ equations:

$$\begin{aligned} g^{x_1} &= \prod_{i=1}^{k} p_i^{e_{1,i}} \bmod p \\ &\vdots \\ g^{x_\ell} &= \prod_{i=1}^{k} p_i^{e_{\ell,i}} \bmod p\,. \end{aligned}$$