11

Finding generators and discrete logarithms in $\mathbb{Z}_p^*$

As we have seen in Theorem 7.28, for a prime p, $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$. This means that there exists a generator $\gamma \in \mathbb{Z}_p^*$, such that each $\alpha \in \mathbb{Z}_p^*$ can be written uniquely as $\alpha = \gamma^x$, where x is an integer with $0 \le x < p-1$; the integer x is called the **discrete logarithm** of α to the base γ, and is denoted $\log_\gamma \alpha$.

This chapter discusses some computational problems in this setting; namely, how to efficiently find a generator γ, and given γ and α, how to compute $\log_\gamma \alpha$.

More generally, if γ generates a subgroup G of $\mathbb{Z}_p^*$ of order q, where $q \mid (p-1)$, and $\alpha \in G$, then $\log_\gamma \alpha$ is defined to be the unique integer x with $0 \le x < q$ and $\alpha = \gamma^x$. In some situations it is more convenient to view $\log_\gamma \alpha$ as an element of $\mathbb{Z}_q$. Also for $x \in \mathbb{Z}_q$, with $x = [a]_q$, one may write γ^x to denote γ^a. There can be no confusion, since if $x = [a']_q$, then $\gamma^{a'} = \gamma^a$. However, in this chapter, we shall view $\log_\gamma \alpha$ as an integer.

Although we work in the group $\mathbb{Z}_p^*$, all of the algorithms discussed in this chapter trivially generalize to any finite cyclic group that has a suitably compact representation of group elements and an efficient algorithm for performing the group operation on these representations.

11.1 Finding a generator for $\mathbb{Z}_p^*$

In this section, we consider the problem of how to find a generator for $\mathbb{Z}_p^*$. There is no efficient algorithm known for this problem, unless the prime factorization of $p-1$ is given, and even then, we must resort to the use of a probabilistic algorithm. Of course, factoring in general is believed to be a very difficult problem, so it may not be easy to get the prime factorization of $p-1$. However, if our goal is to construct a large prime p, together with a generator for $\mathbb{Z}_p^*$, then we may use Algorithm RFN in §9.6 to generate a random factored number n in some range, test $n+1$ for primality, and then repeat until we get a factored number n such that

$p = n + 1$ is prime. In this way, we can generate a random prime p in a given range along with the factorization of $p - 1$.

We now present an efficient probabilistic algorithm that takes as input an odd prime p, along with the prime factorization

$$p - 1 = \prod_{i=1}^{r} q_i^{e_i},$$

and outputs a generator for $\mathbb{Z}_p^*$. It runs as follows:

```
for $i \leftarrow 1$ to $r$ do
    repeat
        choose $\alpha \in \mathbb{Z}_p^*$ at random
        compute $\beta \leftarrow \alpha^{(p-1)/q_i}$
    until $\beta \neq 1$
    $\gamma_i \leftarrow \alpha^{(p-1)/q_i^{e_i}}$
$\gamma \leftarrow \prod_{i=1}^{r} \gamma_i$
output $\gamma$
```

First, let us analyze the correctness of this algorithm. When the ith loop iteration terminates, by construction, we have

$$\gamma_i^{q_i^{e_i}} = 1 \text{ but } \gamma_i^{q_i^{e_i - 1}} \neq 1.$$

It follows (see Theorem 6.37) that γ_i has multiplicative order $q_i^{e_i}$. From this, it follows (see Theorem 6.38) that γ has multiplicative order $p - 1$. Thus, we have shown that if the algorithm terminates, its output is always correct.

Let us now analyze the running time of this algorithm. Fix $i = 1, \ldots, r$, and consider the repeat/until loop in the ith iteration of the outer loop. Let L_i be the random variable whose value is the number of iterations of this repeat/until loop. Since α is chosen at random from $\mathbb{Z}_p^*$, the value of β is uniformly distributed over the image of the $(p - 1)/q_i$-power map (see Theorem 8.5), and since the latter is a subgroup of $\mathbb{Z}_p^*$ of order q_i (see Example 7.61), we see that $\beta = 1$ with probability $1/q_i$. Thus, L_i has a geometric distribution with associated success probability $1 - 1/q_i$, and $\mathsf{E}[L_i] = 1/(1 - 1/q_i) \leq 2$ (see Theorem 9.3).

Now set $L := L_1 + \cdots + L_r$. By linearity of expectation (Theorem 8.14), we have $\mathsf{E}[L] = \mathsf{E}[L_1] + \cdots + \mathsf{E}[L_r] \leq 2r$. The running time Z of the entire algorithm is $O(L \cdot \operatorname{len}(p)^3)$, and hence the expected running time is $\mathsf{E}[Z] = O(r \operatorname{len}(p)^3)$, and since $r \leq \log_2 p$, we have $\mathsf{E}[Z] = O(\operatorname{len}(p)^4)$.

Although this algorithm is quite practical, there are asymptotically faster algorithms for this problem (see Exercise 11.2).

EXERCISE 11.1. Suppose we are not given the prime factorization of $p - 1$, but rather, just a prime q dividing $p-1$, and we want to find an element of multiplicative order q in $\mathbb{Z}_p^*$. Design and analyze an efficient algorithm to do this.

EXERCISE 11.2. Suppose we are given a prime p, along with the prime factorization $p - 1 = \prod_{i=1}^{r} q_i^{e_i}$.

(a) If, in addition, we are given $\alpha \in \mathbb{Z}_p^*$, show how to compute the multiplicative order of α in time $O(r \operatorname{len}(p)^3)$. Hint: use Exercise 6.40.

(b) Improve the running time bound to $O(\operatorname{len}(r) \operatorname{len}(p)^3)$. Hint: use Exercise 3.39.

(c) Modifying the algorithm you developed for part (b), show how to construct a generator for $\mathbb{Z}_p^*$ in expected time $O(\operatorname{len}(r) \operatorname{len}(p)^3)$.

EXERCISE 11.3. Suppose we are given a positive integer n, along with its prime factorization $n = p_1^{e_1} \cdots p_r^{e_r}$, and that for each $i = 1, \ldots, r$, we are also given the prime factorization of $p_i - 1$. Show how to efficiently compute the multiplicative order of any element $\alpha \in \mathbb{Z}_n^*$.

EXERCISE 11.4. Suppose there is an efficient algorithm that takes as input a positive integer n and an element $\alpha \in \mathbb{Z}_n^*$, and computes the multiplicative order of α. Show how to use this algorithm to build an efficient integer factoring algorithm.

11.2 Computing discrete logarithms in $\mathbb{Z}_p^*$

In this section, we consider algorithms for computing the discrete logarithm of $\alpha \in \mathbb{Z}_p^*$ to a given base γ. The algorithms we present here are, in the worst case, exponential-time algorithms, and are by no means the best possible; however, in some special cases, these algorithms are not so bad.

11.2.1 Brute-force search

Suppose that $\gamma \in \mathbb{Z}_p^*$ generates a subgroup G of $\mathbb{Z}_p^*$ of order $q > 1$ (not necessarily prime), and we are given p, q, γ, and $\alpha \in G$, and wish to compute $\log_\gamma \alpha$.

The simplest algorithm to solve this problem is **brute-force search**:

```
$\beta \leftarrow 1$
$i \leftarrow 0$
while $\beta \neq \alpha$ do
    $\beta \leftarrow \beta \cdot \gamma$
    $i \leftarrow i + 1$
output $i$
```

This algorithm is clearly correct, and the main loop will always halt after at most q iterations (assuming, as we are, that $\alpha \in G$). So the total running time is $O(q \operatorname{len}(p)^2)$.

11.2.2 Baby step/giant step method

As above, suppose that $\gamma \in \mathbb{Z}_p^*$ generates a subgroup G of $\mathbb{Z}_p^*$ of order $q > 1$ (not necessarily prime), and we are given p, q, γ, and $\alpha \in G$, and wish to compute $\log_\gamma \alpha$.

A faster algorithm than brute-force search is the **baby step/giant step method**. It works as follows.

Let us choose an approximation m to $q^{1/2}$. It does not have to be a very good approximation — we just need $m = \Theta(q^{1/2})$. Also, let $m' = \lfloor q/m \rfloor$, so that $m' = \Theta(q^{1/2})$ as well.

The idea is to compute all the values γ^i for $i = 0, \ldots, m-1$ (the "baby steps") and to build an "associative array" (or "lookup table") T that maps the key γ^i to the value i. For $\beta \in \mathbb{Z}_p^*$, we shall write $T[\beta]$ to denote the value associated with the key β, writing $T[\beta] = \perp$ if there is no such value. We shall assume that T is implemented so that accessing $T[\beta]$ is fast. Using an appropriate data structure, T can be implemented so that accessing individual elements takes time $O(\operatorname{len}(p))$. One such data structure is a *radix tree* (also called a *search trie*). Other data structures may be used (for example, a *hash table* or a *binary search tree*), but these may have somewhat different access times.

We can build the associative array T using the following algorithm:

```
initialize $T$   // $T[\beta] = \perp$ for all $\beta \in \mathbb{Z}_p^*$
$\beta \leftarrow 1$
for $i \leftarrow 0$ to $m - 1$ do
    $T[\beta] \leftarrow i$
    $\beta \leftarrow \beta \cdot \gamma$
```

Clearly, this algorithm takes time $O(q^{1/2} \operatorname{len}(p)^2)$.

After building the lookup table, we execute the following procedure (the "giant steps"):

```
$\gamma' \leftarrow \gamma^{-m}$
$\beta \leftarrow \alpha,\ \ j \leftarrow 0,\ \ i \leftarrow T[\beta]$
while $i = \perp$ do
    $\beta \leftarrow \beta \cdot \gamma',\ \ j \leftarrow j + 1,\ \ i \leftarrow T[\beta]$
$x \leftarrow jm + i$
output $x$
```

To analyze this procedure, suppose that $\alpha = \gamma^x$ with $0 \le x < q$. Now, x can be written in a unique way as $x = vm + u$, where u and v are integers with $0 \le u < m$ and $0 \le v \le m'$. In the jth loop iteration, for $j = 0, 1, \ldots$, we have

$$\beta = \alpha\gamma^{-mj} = \gamma^{(v-j)m+u}.$$

So we will detect $i \neq \perp$ precisely when $j = v$, in which case $i = u$. Thus, the output will be correct, and the total running time of the algorithm (for both the "baby steps" and "giant steps" parts) is easily seen to be $O(q^{1/2}\operatorname{len}(p)^2)$.

While this algorithm is much faster than brute-force search, it has the drawback that it requires space for about $q^{1/2}$ elements of $\mathbb{Z}_p$. Of course, there is a "time/space trade-off" here: by choosing m smaller, we get a table of size $O(m)$, but the running time will be proportional to $O(q/m)$. In §11.2.5 below, we discuss an algorithm that runs (at least heuristically) in time $O(q^{1/2}\operatorname{len}(q)\operatorname{len}(p)^2)$, but which requires space for only a constant number of elements of $\mathbb{Z}_p$.

11.2.3 Groups of order q^e

Suppose that $\gamma \in \mathbb{Z}_p^*$ generates a subgroup G of $\mathbb{Z}_p^*$ of order q^e, where $q > 1$ and $e \ge 1$, and we are given p, q, e, γ, and $\alpha \in G$, and wish to compute $\log_\gamma \alpha$.

There is a simple algorithm that allows one to reduce this problem to the problem of computing discrete logarithms in the subgroup of $\mathbb{Z}_p^*$ of order q.

It is perhaps easiest to describe the algorithm recursively. The base case is when $e = 1$, in which case, we use an algorithm for the subgroup of $\mathbb{Z}_p^*$ of order q. For this, we might employ the algorithm in §11.2.2, or if q is *very* small, the algorithm in §11.2.1.

Suppose now that $e > 1$. We choose an integer f with $0 < f < e$. Different strategies for choosing f yield different algorithms—we discuss this below. Suppose $\alpha = \gamma^x$, where $0 \le x < q^e$. Then we can write $x = q^f v + u$, where u and v are integers with $0 \le u < q^f$ and $0 \le v < q^{e-f}$. Therefore,

$$\alpha^{q^{e-f}} = \gamma^{q^{e-f}u}.$$

Note that $\gamma^{q^{e-f}}$ has multiplicative order q^f, and so if we recursively compute the discrete logarithm of $\alpha^{q^{e-f}}$ to the base $\gamma^{q^{e-f}}$, we obtain u.

Having obtained u, observe that

$$\alpha/\gamma^u = \gamma^{q^f v}.$$

Note also that γ^{q^f} has multiplicative order q^{e-f}, and so if we recursively compute the discrete logarithm of α/γ^u to the base γ^{q^f}, we obtain v, from which we then compute $x = q^f v + u$.

Let us put together the above ideas succinctly in a recursive procedure:

Algorithm RDL. On input p, q, e, γ, α as above, do the following:

```
if $e = 1$ then
    return $\log_\gamma \alpha$   // base case: use a different algorithm
else
    select $f \in \{1, \ldots, e-1\}$
    $u \leftarrow \mathrm{RDL}(p, q, f, \gamma^{q^{e-f}}, \alpha^{q^{e-f}})$   // $0 \le u < q^f$
    $v \leftarrow \mathrm{RDL}(p, q, e-f, \gamma^{q^f}, \alpha/\gamma^u)$   // $0 \le v < q^{e-f}$
    return $q^f v + u$
```

To analyze the running time of this recursive algorithm, note that the running time of the body of one recursive invocation (not counting the running time of the recursive calls it makes) is $O(e \operatorname{len}(q) \operatorname{len}(p)^2)$. To calculate the total running time, we have to sum up the running times of all the recursive calls plus the running times of all the base cases.

Regardless of the strategy for choosing f, the total number of base case invocations is e. Note that all the base cases compute discrete logarithms to the base $\gamma^{q^{e-1}}$. Assuming we implement the base case using the baby step/giant step algorithm in §11.2.2, the total running time for all the base cases is therefore $O(eq^{1/2} \operatorname{len}(p)^2)$.

The total running time for the recursion (not including the base case computations) depends on the strategy used to choose the split f. It is helpful to represent the behavior of the algorithm using a **recursion tree**. This is a binary tree, where every node represents one recursive invocation of the algorithm; the root of the tree represents the initial invocation of the algorithm; for every node N in the tree, if N represents the recursive invocation I, then N's children (if any) represent the recursive invocations made by I. We can naturally organize the nodes of the recursion tree by **levels**: the root of the recursion tree is at level 0, its children are at level 1, its grandchildren at level 2, and so on. The **depth** of the recursion tree is defined to be the maximum level of any node.

We consider two different strategies for choosing the split f:

- If we always choose $f = 1$ or $f = e - 1$, then the depth of the recursion tree is $O(e)$. The running time contributed by each level of the recursion tree is $O(e \operatorname{len}(q) \operatorname{len}(p)^2)$, and so the total running time for the recursion is $O(e^2 \operatorname{len}(q) \operatorname{len}(p)^2)$. Note that if $f = 1$, then the algorithm is essentially tail recursive, and so may be easily converted to an iterative algorithm without the need for a stack.
- If we use a "balanced" divide-and-conquer strategy, choosing $f \approx e/2$, then the depth of the recursion tree is $O(\operatorname{len}(e))$, while the running time

contributed by each level of the recursion tree is still $O(e\,\mathrm{len}(q)\,\mathrm{len}(p)^2)$. Thus, the total running time for the recursion is $O(e\,\mathrm{len}(e)\,\mathrm{len}(q)\,\mathrm{len}(p)^2)$.

Assuming we use the faster, balanced recursion strategy, and that we use the baby step/giant step algorithm for the base case, the total running time of Algorithm RDL is:

$$O((eq^{1/2} + e\,\mathrm{len}(e)\,\mathrm{len}(q)) \cdot \mathrm{len}(p)^2).$$

11.2.4 Discrete logarithms in $\mathbb{Z}_p^*$

Suppose that we are given a prime p, along with the prime factorization

$$p - 1 = \prod_{i=1}^{r} q_i^{e_i},$$

a generator γ for $\mathbb{Z}_p^*$, and $\alpha \in \mathbb{Z}_p^*$. We wish to compute $\log_\gamma \alpha$.

Suppose that $\alpha = \gamma^x$, where $0 \le x < p - 1$. Then for $i = 1, \ldots, r$, we have

$$\alpha^{(p-1)/q_i^{e_i}} = \left(\gamma^{(p-1)/q_i^{e_i}}\right)^x.$$

Note that $\gamma^{(p-1)/q_i^{e_i}}$ has multiplicative order $q_i^{e_i}$, and if x_i is the discrete logarithm of $\alpha^{(p-1)/q_i^{e_i}}$ to the base $\gamma^{(p-1)/q_i^{e_i}}$, then we have $0 \le x_i < q_i^{e_i}$ and $x \equiv x_i \pmod{q_i^{e_i}}$.

Thus, if we compute the values $x_1, \ldots, x_r$, using Algorithm RDL in §11.2.3, we can obtain x using the algorithm of the Chinese remainder theorem (see Theorem 4.6). If we define $q := \max\{q_1, \ldots, q_r\}$, then the running time of this algorithm will be bounded by $q^{1/2}\,\mathrm{len}(p)^{O(1)}$.

We conclude that

> *the difficulty of computing discrete logarithms in $\mathbb{Z}_p^*$ is determined by the size of the largest prime dividing $p - 1$.*

11.2.5 A space-efficient square-root time algorithm

We present a more space-efficient alternative to the algorithm in §11.2.2, the analysis of which we leave as a series of exercises for the reader.

The algorithm makes a somewhat heuristic assumption that we have a function that "behaves" for all practical purposes like a random function. Such functions can indeed be constructed using cryptographic techniques under reasonable intractability assumptions; however, for the particular application here, one can get by in practice with much simpler constructions.

Let p be a prime, q a prime dividing $p - 1$, γ an element of $\mathbb{Z}_p^*$ that generates a subgroup G of $\mathbb{Z}_p^*$ of order q, and $\alpha \in G$. Let F be a function mapping elements of G to $\{0, \ldots, q - 1\}$. Define $H : G \to G$ to be the function that sends β to $\beta\alpha\gamma^{F(\beta)}$.

The algorithm runs as follows:

```
$i \leftarrow 1$
$x \leftarrow 0, \beta \leftarrow \alpha,$
$x' \leftarrow F(\beta), \beta' \leftarrow H(\beta)$
while $\beta \neq \beta'$ do
    $x \leftarrow (x + F(\beta)) \bmod q, \beta \leftarrow H(\beta)$
    repeat 2 times
        $x' \leftarrow (x' + F(\beta')) \bmod q, \beta' \leftarrow H(\beta')$
    $i \leftarrow i + 1$
if $i < q$ then
    output $(x - x')i^{-1} \bmod q$
else
    output “fail”
```

To analyze this algorithm, let us define $\beta_1, \beta_2, \ldots,$ as follows: $\beta_1 := \alpha$ and for $i > 1$, $\beta_i := H(\beta_{i-1})$.

EXERCISE 11.5. Show that each time the main loop of the algorithm is entered, we have $\beta = \beta_i = \gamma^x \alpha^i$, and $\beta' = \beta_{2i} = \gamma^{x'} \alpha^{2i}$.

EXERCISE 11.6. Show that if the loop terminates with $i < q$, the value output is equal to $\log_\gamma \alpha$.

EXERCISE 11.7. Let j be the smallest index such that $\beta_j = \beta_k$ for some index $k < j$. Show that $j \leq q + 1$ and that the loop terminates with $i < j$ (and in particular, $i \leq q$).

EXERCISE 11.8. Assume that F is a random function, meaning that it is chosen at random, uniformly from among all functions from G into $\{0, \ldots, q-1\}$. Show that this implies that H is a random function, meaning that it is uniformly distributed over all functions from G into G.

EXERCISE 11.9. Assuming that F is a random function as in the previous exercise, apply the result of Exercise 8.45 to conclude that the expected running time of the algorithm is $O(q^{1/2} \operatorname{len}(q) \operatorname{len}(p)^2)$, and that the probability that the algorithm fails is exponentially small in q.

11.3 The Diffie–Hellman key establishment protocol

One of the main motivations for studying algorithms for computing discrete logarithms is the relation between this problem and the problem of breaking a protocol called the **Diffie–Hellman key establishment protocol**, named after its inventors.

In this protocol, Alice and Bob need never to have talked to each other before, but nevertheless, can establish a shared secret key that nobody else can easily compute. To use this protocol, a third party must provide a "telephone book," which contains the following information:

- p, q, and γ, where p and q are primes with $q \mid (p-1)$, and γ is an element generating a subgroup G of $\mathbb{Z}_p^*$ of order q;
- an entry for each user, such as Alice or Bob, that contains the user's name, along with a "public key" for that user, which is an element of the group G.

To use this system, Alice posts her public key in the telephone book, which is of the form $\alpha = \gamma^x$, where $x \in \{0, \ldots, q-1\}$ is chosen by Alice at random. The value x is Alice's "secret key," which Alice never divulges to anybody. Likewise, Bob posts his public key, which is of the form $\beta = \gamma^y$, where $y \in \{0, \ldots, q-1\}$ is chosen by Bob at random, and is his secret key.

To establish a shared key known only between them, Alice retrieves Bob's public key β from the telephone book, and computes $\kappa_A := \beta^x$. Likewise, Bob retrieves Alice's public key α, and computes $\kappa_B := \alpha^y$. It is easy to see that

$$\kappa_A = \beta^x = (\gamma^y)^x = \gamma^{xy} = (\gamma^x)^y = \alpha^y = \kappa_B,$$

and hence Alice and Bob share the same secret key $\kappa := \kappa_A = \kappa_B$.

Using this shared secret key, they can then use standard methods for encryption and message authentication to hold a secure conversation. We shall not go any further into how this is done; rather, we briefly (and only superficially) discuss some aspects of the security of the key establishment protocol itself. Clearly, if an attacker obtains α and β from the telephone book, and computes $x = \log_\gamma \alpha$, then he can compute Alice and Bob's shared key as $\kappa = \beta^x$—in fact, given x, an attacker can efficiently compute *any* key shared between Alice and another user.

Thus, if this system is to be secure, it should be very difficult to compute discrete logarithms. However, the assumption that computing discrete logarithms is hard is not enough to guarantee security. Indeed, it is not entirely inconceivable that the discrete logarithm problem is hard, and yet the problem of computing κ from α and β is easy. The latter problem—computing κ from α and β—is called the **Diffie–Hellman problem**.

As in the discussion of the RSA cryptosystem in §4.7, the reader is warned that the above discussion about security is a bit of an oversimplification. A complete discussion of all the security issues related to the above protocol is beyond the scope of this text.

Note that in our presentation of the Diffie–Hellman protocol, we work with a generator of a subgroup G of $\mathbb{Z}_p^*$ of prime order, rather than a generator for $\mathbb{Z}_p^*$. There are several reasons for doing this: one is that there are no known discrete logarithm algorithms that are any more practical in G than in $\mathbb{Z}_p^*$, provided the order q of G is sufficiently large; another is that by working in G, the protocol becomes substantially more efficient. In typical implementations, p is 1024 bits long, so as to protect against subexponential-time algorithms such as those discussed later in §15.2, while q is 160 bits long, which is enough to protect against the square-root-time algorithms discussed in §11.2.2 and §11.2.5. The modular exponentiations in the protocol will run several times faster using "short," 160-bit exponents rather than "long," 1024-bit exponents.

For the following exercise, we need the following notions from complexity theory.

- We say problem A is **deterministic poly-time reducible** to problem B if there exists a deterministic algorithm R for solving problem A that makes calls to a subroutine for problem B, where the running time of R (not including the running time for the subroutine for B) is polynomial in the input length.
- We say that problems A and B are **deterministic poly-time equivalent** if A is deterministic poly-time reducible to B and B is deterministic poly-time reducible to A.

EXERCISE 11.10. Consider the following problems.

(a) Given a prime p, a prime q that divides $p - 1$, an element $\gamma \in \mathbb{Z}_p^*$ generating a subgroup G of $\mathbb{Z}_p^*$ of order q, and two elements $\alpha, \beta \in G$, compute γ^{xy}, where $x := \log_\gamma \alpha$ and $y := \log_\gamma \beta$. (This is just the Diffie–Hellman problem.)

(b) Given a prime p, a prime q that divides $p - 1$, an element $\gamma \in \mathbb{Z}_p^*$ generating a subgroup G of $\mathbb{Z}_p^*$ of order q, and an element $\alpha \in G$, compute γ^{x^2}, where $x := \log_\gamma \alpha$.

(c) Given a prime p, a prime q that divides $p - 1$, an element $\gamma \in \mathbb{Z}_p^*$ generating a subgroup G of $\mathbb{Z}_p^*$ of order q, and two elements $\alpha, \beta \in G$, with $\beta \neq 1$, compute $\gamma^{xy'}$, where $x := \log_\gamma \alpha$, $y' := y^{-1} \bmod q$, and $y := \log_\gamma \beta$.

(d) Given a prime p, a prime q that divides $p - 1$, an element $\gamma \in \mathbb{Z}_p^*$ generating a subgroup G of $\mathbb{Z}_p^*$ of order q, and an element $\alpha \in G$, with $\alpha \neq 1$, compute $\gamma^{x'}$, where $x' := x^{-1} \bmod q$ and $x := \log_\gamma \alpha$.

Show that these problems are deterministic poly-time equivalent. Moreover, your reductions should preserve the values of p, q, and γ; that is, if the algorithm that reduces one problem to another takes as input an instance of the former problem of the form $(p, q, \gamma, \ldots)$, it should invoke the subroutine for the latter problem with inputs of the form $(p, q, \gamma, \ldots)$.

EXERCISE 11.11. Suppose there is a probabilistic algorithm A that takes as input a prime p, a prime q that divides $p-1$, and an element $\gamma \in \mathbb{Z}_p^*$ generating a subgroup G of $\mathbb{Z}_p^*$ of order q. The algorithm also takes as input $\alpha \in G$. It outputs either "failure," or $\log_\gamma \alpha$. Furthermore, assume that A runs in expected polynomial time, and that for all p, q, and γ of the above form, and for randomly chosen $\alpha \in G$, A succeeds in computing $\log_\gamma \alpha$ with probability $\varepsilon(p, q, \gamma)$. Here, the probability is taken over the random choice of α, as well as the random choices made during the execution of A. Show how to use A to construct another probabilistic algorithm A' that takes as input p, q, and γ as above, as well as $\alpha \in G$, runs in expected polynomial time, and that satisfies the following property:

> if $\varepsilon(p, q, \gamma) \geq 0.001$, then *for all* $\alpha \in G$, A' computes $\log_\gamma \alpha$ with probability at least 0.999.

The algorithm A' in the previous exercise is an example of a **random self-reduction**, which means an algorithm that reduces the task of solving an *arbitrary* instance of a given problem to that of solving a *random* instance of the *same* problem. Intuitively, the existence of such a reduction means that the problem is no harder in the worst case than on average.

EXERCISE 11.12. Let p be a prime, q a prime that divides $p-1$, $\gamma \in \mathbb{Z}_p^*$ an element that generates a subgroup G of $\mathbb{Z}_p^*$ of order q, and $\alpha \in G$. For $\delta \in G$, a **representation of δ with respect to γ and α** is a pair of integers (r, s), with $0 \leq r < q$ and $0 \leq s < q$, such that $\gamma^r \alpha^s = \delta$.

(a) Show that for every $\delta \in G$, there are precisely q representations (r, s) of δ with respect to γ and α, and among these, there is precisely one with $s = 0$.

(b) Show that given a representation (r, s) of 1 with respect to γ and α such that $s \neq 0$, we can efficiently compute $\log_\gamma \alpha$.

(c) Show that given any $\delta \in G$, along with any two distinct representations of δ with respect to γ and α, we can efficiently compute $\log_\gamma \alpha$.

(d) Suppose we are given access to an "oracle" that, when presented with any $\delta \in G$, tells us some representation of δ with respect to γ and α. Show how to use this oracle to efficiently compute $\log_\gamma \alpha$.

The following two exercises examine the danger of the use of "short" exponents in discrete logarithm based cryptographic schemes that *do not* work with a group of prime order.

EXERCISE 11.13. Let p be a prime and let $p - 1 = q_1^{e_1} \cdots q_r^{e_r}$ be the prime factorization of $p-1$. Let γ be a generator for $\mathbb{Z}_p^*$. Let y be a positive number, and let $Q_p(y)$ be the product of all the prime powers $q_i^{e_i}$ with $q_i \leq y$. Suppose you are given p, y, the primes q_i dividing $p-1$ with $q_i \le y$, along with γ, an element α of $\mathbb{Z}_p^*$, and a bound $\hat{x}$, where $x := \log_\gamma \alpha < \hat{x}$. Show how to compute x in time

$$(y^{1/2} + (\hat{x}/Q_p(y))^{1/2}) \cdot \mathrm{len}(p)^{O(1)}.$$

EXERCISE 11.14. Continuing with the previous, let $Q'_p(y)$ denote the product of all the primes q_i dividing $p-1$ with $q_i \le y$. Note that $Q'_p(y) \mid Q_p(y)$. The goal of this exercise is to estimate the expected value of $\log Q'_p(y)$, assuming p is a large, random prime. To this end, let R be a random variable that is uniformly distributed over all ℓ-bit primes, and assume that $y \le 2^{\ell/3}$. Assuming Conjecture 5.22, show that asymptotically (as $\ell \to \infty$), we have $\mathsf{E}[\log Q'_{\mathsf{R}}(y)] = \log y + O(1)$.

The results of the previous two exercises caution against the use of “short” exponents in cryptographic schemes based on the discrete logarithm problem for $\mathbb{Z}_p^*$. For example, suppose that p is a random 1024-bit prime, and that for reasons of efficiency, one chooses $\hat{x} \approx 2^{160}$, thinking that a method such as the baby step/giant step method would require $\approx 2^{80}$ steps to recover x. However, if we choose $y \approx 2^{80}$, then the above analysis implies that $Q_p(y)$ is at least $\approx 2^{80}$ with a reasonable probability, in which case $\hat{x}/Q_p(y)$ is at most $\approx 2^{80}$, and so we can in fact recover x in $\approx 2^{40}$ steps (there are known methods to find the primes up to y that divide $p-1$ quickly enough). While 2^{80} may not be a feasible number of steps, 2^{40} may very well be. Of course, none of these issues arise if one works in a subgroup of $\mathbb{Z}_p^*$ of large prime order, which is the recommended practice.

An interesting fact about the Diffie–Hellman problem is that there is no known efficient algorithm to recognize a solution to the problem. Some cryptographic protocols actually rely on the apparent difficulty of this decision problem, which is called the **decisional Diffie–Hellman problem**. The following three exercises develop a random self-reducibility property for this decision problem.

EXERCISE 11.15. Let p be a prime, q a prime dividing $p-1$, and γ an element of $\mathbb{Z}_p^*$ that generates a subgroup G of order q. Let $\alpha \in G$, and let H be the subgroup of $G \times G$ generated by (γ, α). Let $\tilde{\gamma}, \tilde{\alpha}$ be arbitrary elements of G, and define the map

$$\begin{aligned} \rho : \quad & \mathbb{Z}_q \times \mathbb{Z}_q \to G \times G \\ & ([r]_q, [s]_q) \mapsto (\gamma^r \tilde{\gamma}^s, \alpha^r \tilde{\alpha}^s). \end{aligned}$$

Show that the definition of ρ is unambiguous, that ρ is a group homomorphism, and that

- if $(\tilde{\gamma}, \tilde{\alpha}) \in H$, then $\mathrm{Im}\,\rho = H$, and
- if $(\tilde{\gamma}, \tilde{\alpha}) \notin H$, then $\mathrm{Im}\,\rho = G \times G$.

EXERCISE 11.16. For p, q, γ as in the previous exercise, let $\mathcal{D}_{p,q,\gamma}$ be the set of all triples of the form $(\gamma^x, \gamma^y, \gamma^{xy})$, and let $\mathcal{R}_{p,q,\gamma}$ be the set of all triples of the form $(\gamma^x, \gamma^y, \gamma^z)$. Using the result from the previous exercise, design a probabilistic algorithm that runs in expected polynomial time, and that on input p, q, γ, along with a triple $\Gamma \in \mathcal{R}_{p,q,\gamma}$, outputs a triple $\Gamma^* \in \mathcal{R}_{p,q,\gamma}$ such that

- if $\Gamma \in \mathcal{D}_{p,q,\gamma}$, then Γ^* is uniformly distributed over $\mathcal{D}_{p,q,\gamma}$, and
- if $\Gamma \notin \mathcal{D}_{p,q,\gamma}$, then Γ^* is uniformly distributed over $\mathcal{R}_{p,q,\gamma}$.

EXERCISE 11.17. Suppose that A is a probabilistic algorithm that takes as input p, q, γ as in the previous exercise, along with a triple $\Gamma^* \in \mathcal{R}_{p,q,\gamma}$, and outputs either 0 or 1. Furthermore, assume that A runs in expected polynomial time. Define two random variables, $X_{p,q,\gamma}$ and $Y_{p,q,\gamma}$, as follows:

- $X_{p,q,\gamma}$ is defined to be the output of A on input p, q, γ, and Γ^*, where Γ^* is uniformly distributed over $\mathcal{D}_{p,q,\gamma}$, and
- $Y_{p,q,\gamma}$ is defined to be the output of A on input p, q, γ, and Γ^*, where Γ^* is uniformly distributed over $\mathcal{R}_{p,q,\gamma}$.

In both cases, the value of the random variable is determined by the random choice of Γ^*, as well as the random choices made by the algorithm. Define

$$\varepsilon(p, q, \gamma) := \Big| \mathsf{P}[X_{p,q,\gamma} = 1] - \mathsf{P}[Y_{p,q,\gamma} = 1] \Big|.$$

Using the result of the previous exercise, show how to use A to design a probabilistic, expected polynomial-time algorithm that takes as input p, q, γ as above, along with $\Gamma \in \mathcal{R}_{p,q,\gamma}$, and outputs either "yes" or "no," so that

> if $\varepsilon(p, q, \gamma) \geq 0.001$, then for *all* $\Gamma \in \mathcal{R}_{p,q,\gamma}$, the probability that A' correctly determines whether $\Gamma \in \mathcal{D}_{p,q,\gamma}$ is at least 0.999.

Hint: use the Chernoff bound.

The following exercise demonstrates that the problem of distinguishing "Diffie–Hellman triples" from "random triples" is hard only if the order of the underlying group is not divisible by any small primes, which is another reason we have chosen to work with groups of large prime order.

EXERCISE 11.18. Assume the notation of the previous exercise, but let us drop the restriction that q is prime. Design and analyze a deterministic algorithm A that takes inputs p, q, γ and $\Gamma^* \in \mathcal{R}_{p,q,\gamma}$, that outputs 0 or 1, and that satisfies the following property: if t is the *smallest* prime dividing q, then A runs in time $(t + \operatorname{len}(p))^{O(1)}$, and the "distinguishing advantage" $\varepsilon(p, q, \gamma)$ for A on inputs p, q, γ is at least $1/t$.

11.4 Notes

The probabilistic algorithm in §11.1 for finding a generator for $\mathbb{Z}_p^*$ can be made deterministic under a generalization of the Riemann hypothesis. Indeed, as discussed in §10.5, under such a hypothesis, Bach's result [10] implies that for each prime $q \mid (p-1)$, the least positive integer a such that $[a]_p \in \mathbb{Z}_p^* \setminus (\mathbb{Z}_p^*)^q$ is at most $2 \log p$.

Related to the problem of constructing a generator for $\mathbb{Z}_p^*$ is the question of how big is the smallest positive integer g such that $[g]_p$ is a generator for $\mathbb{Z}_p^*$; that is, how big is the smallest (positive) primitive root modulo p. The best bounds on the least primitive root are also obtained using the same generalization of the Riemann hypothesis mentioned above. Under this hypothesis, Wang [104] showed that the least primitive root modulo p is $O(r^6 \operatorname{len}(p)^2)$, where r is the number of distinct prime divisors of $p-1$. Shoup [95] improved Wang's bound to $O(r^4 \operatorname{len}(r)^4 \operatorname{len}(p)^2)$ by adapting a result of Iwaniec [50, 51] and applying it to Wang's proof. The best unconditional bound on the smallest primitive root modulo p is $p^{1/4+o(1)}$ (this bound is also in Wang [104]). Of course, even if there exists a small primitive root, there is no known way to efficiently recognize a primitive root modulo p without knowing the prime factorization of $p-1$.

As we already mentioned, all of the algorithms presented in this chapter are completely "generic," in the sense that they work in *any* finite cyclic group—we really did not exploit any properties of $\mathbb{Z}_p^*$ other than the fact that it is a cyclic group. In fact, as far as such "generic" algorithms go, the algorithms presented here for discrete logarithms are optimal [71, 98]. However, there are faster, "non-generic" algorithms (though still not polynomial time) for discrete logarithms in $\mathbb{Z}_p^*$. We shall examine one such algorithm later, in Chapter 15.

The "baby step/giant step" algorithm in §11.2.2 is due to Shanks [91]. See, for example, the book by Cormen, Leiserson, Rivest, and Stein [29] for appropriate data structures to implement the lookup table used in that algorithm. In particular, see Problem 12-2 in [29] for a brief introduction to radix trees, which is the data structure that yields the best running time (at least in principle) for our application.

The algorithms in §11.2.3 and §11.2.4 are variants of an algorithm published by Pohlig and Hellman [75]. See Chapter 4 of [29] for details on how one analyzes recursive algorithms, such as the one presented in §11.2.3; in particular, Section 4.2 in [29] discusses in detail the notion of a recursion tree.

The algorithm in §11.2.5 is a variant of an algorithm of Pollard [76]; in fact, Pollard's algorithm is a bit more efficient than the one presented here, but the analysis of its running time depends on stronger heuristics. Pollard's paper also describes an algorithm for computing discrete logarithms that lie in a restricted interval—if the interval has width w, this algorithm runs (heuristically) in time

$w^{1/2}\operatorname{len}(p)^{O(1)}$, and requires space for $O(\operatorname{len}(w))$ elements of $\mathbb{Z}_p$. This algorithm is useful in reducing the space requirement for the algorithm of Exercise 11.13.

The key establishment protocol in §11.3 is from Diffie and Hellman [34]. That paper initiated the study of **public key cryptography**, which has proved to be a very rich field of research. Exercises 11.13 and 11.14 are based on van Oorschot and Wiener [74]. For more on the decisional Diffie–Hellman assumption, see Boneh [18].