Chapter 2

Discrete Logarithms and Diffie–Hellman

2.1 The birth of public key cryptography

In 1976, Whitfield Diffie and Martin Hellman published their now famous paper [36] entitled "New Directions in Cryptography." In this paper they formulated the concept of a public key encryption system and made several groundbreaking contributions to this new field. A short time earlier, Ralph Merkle had independently isolated one of the fundamental problems and invented a public key construction for an undergraduate project in a computer science class at Berkeley, but this was little understood at the time. Merkle's work "Secure communication over insecure channels" appeared in 1982 [74].

However, it turns out that the concept of public key encryption was originally discovered by James Ellis while working at the British Government Communications Headquarters (GCHQ). Ellis's discoveries in 1969 were classified as secret material by the British government and were not declassified and released until 1997, after his death. It is now known that two other researchers at GCHQ, Malcolm Williamson and Clifford Cocks, discovered the Diffie–Hellman key exchange algorithm and the RSA public key encryption system, respectively, before their rediscovery and public dissemination by Diffie, Hellman, Rivest, Shamir, and Adleman. To learn more about the fascinating history of public key cryptography, see for example [35, 39, 58, 128].

The Diffie–Hellman publication was an extremely important event—it set forth the basic definitions and goals of a new field of mathematics/computer science, a field whose existence was dependent on the then emerging age of the digital computer. Indeed, their paper begins with a call to arms:

We stand today on the brink of a revolution in cryptography.

J. Hoffstein et al., *An Introduction to Mathematical Cryptography*,
DOI: 10.1007/978-0-387-77994-2_2,

An original or breakthrough scientific idea is often called revolutionary, but in this instance, as the authors were fully aware, the term revolutionary was relevant in another sense. Prior to the publication of "New Directions...," encryption research in the United States was the domain of the National Security Agency, and all information in this area was classified. Indeed, until the mid-1990s, the United States government treated cryptographic algorithms as munitions, which meant that their export was prosecutable as a treasonable offense. Eventually, the government realized the futility of trying to prevent free and open discussion about abstract cryptographic algorithms and the dubious legality of restricting domestic use of strong cryptographic methods. However, in order to maintain some control, the government continued to restrict export of high security cryptographic algorithms if they were "machine readable." Their object, to prevent widespread global dissemination of sophisticated cryptography programs to potential enemies of the United States, was laudable,[1] but there were two difficulties that rendered the government's policy unworkable.

First, the existence of optical scanners creates a very blurry line between "machine readable" and "human text." To protest the government's policy, people wrote a three line version of the RSA algorithm in a programming language called perl and printed it on tee shirts and soda cans, thereby making these products into munitions. In principle, wearing an "RSA enabled" tee shirt on a flight from New York to Europe subjected the wearer to a large fine and a ten year jail term. Even more amusing (or frightening, depending on your viewpoint), tattoos of the RSA perl code made people's bodies into non-exportable munitions!

Second, although these and other more serious protests and legal challenges had some effect, the government's policy was ultimately rendered moot by a simple reality. Public key algorithms are quite simple, and although it requires a certain expertise to implement them in a secure fashion, the world is full of excellent mathematicians and computer scientists and engineers. Thus government restrictions on the export of "strong crypto" simply encouraged the creation of cryptographic industries in other parts of the world. The government was able to slow the adoption of strong crypto for a few years, but it is now possible for anyone to purchase for a nominal sum cryptographic software that allows completely secure communications.[2]

The first important contribution of Diffie and Hellman in [36] was the definition of a *Public Key Cryptosystem* (PKC) and its associated components—

[1]It is surely laudable to keep potential weapons out of the hands of one's enemies, but many have argued, with considerable justification, that the government also had the less benign objective of preventing other governments from using communication methods secure from United States prying.

[2]Of course, one never knows what cryptanalytic breakthroughs have been made by the scientists at the National Security Agency, since virtually all of their research is classified. The NSA is reputed to be the world's largest single employer of Ph.D.s in mathematics. However, in contrast to the situation before the 1970s, there are now far more cryptographers employed in academia and in the business world than there are in government agencies.

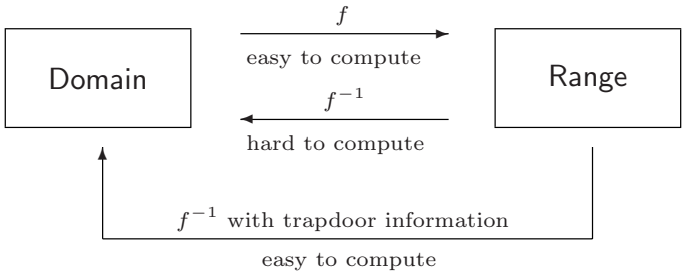

Figure 2.1: Illustration of a one-way trapdoor function

one-way functions and trapdoor information. A *one-way function* is an invertible function that is easy to compute, but whose inverse is difficult to compute. What does it mean to be "difficult to compute"? Intuitively, a function is difficult to compute if any algorithm that attempts to compute the inverse in a "reasonable" amount of time, e.g., less than the age of the universe, will almost certainly fail, where the phrase "almost certainly" must be defined probabilistically. (For a more rigorous definition of "hardness," see Section 2.6.)

Secure PKCs are built using one-way functions that have a *trapdoor*. The trapdoor is a piece of auxiliary information that allows the inverse to be easily computed. This idea is illustrated in Figure 2.1, although it must be stressed that there is a vast chasm separating the abstract idea of a one-way trapdoor function and the actual construction of such a function.

As described in Section 1.7.6, the key for a public key (or asymmetric) cryptosystem consists of two pieces, a private key k_{priv} and a public key k_{pub}, where in practice k_{pub} is computed by applying some key-creation algorithm to k_{priv}. For each public/private key pair $(k_{\mathsf{priv}}, k_{\mathsf{pub}})$ there is an encryption algorithm $e_{k_{\mathsf{pub}}}$ and a corresponding decryption algorithm $d_{k_{\mathsf{priv}}}$. The encryption algorithm $e_{k_{\mathsf{pub}}}$ corresponding to k_{pub} is public knowledge and easy to compute. Similarly, the decryption algorithm $d_{k_{\mathsf{priv}}}$ must be easily computable by someone who knows the private key k_{priv}, but it should be very difficult to compute for someone who knows only the public key k_{pub}.

One says that the private key k_{priv} is *trapdoor information* for the function $e_{k_{\mathsf{pub}}}$, because without the trapdoor information it is very hard to compute the inverse function to $e_{k_{\mathsf{pub}}}$, but with the trapdoor information it is easy to compute the inverse. Notice that in particular, the function that is used to create k_{pub} from k_{priv} must be difficult to invert, since k_{pub} is public knowledge and k_{priv} allows efficient decryption.

It may come as a surprise to learn that despite years of research, it is still not known whether one-way functions exist. In fact, a proof of the existence of one-way functions would simultaneously solve the famous $\mathcal{P} = \mathcal{N}P$

probem in complexity theory.[3] Various candidates for one-way functions have been proposed, and some of them are used by modern public key encryption algorithms. But it must be stressed that the security of these cryptosystems rests on the *assumption* that inverting the underlying function (or finding the private key from the public one) is a hard problem.

The situation is somewhat analogous to theories in physics that gain credibility over time, as they fail to be disproved and continue to explain or generate interesting phenomena. Diffie and Hellman made several suggestions in [36] for one-way functions, including knapsack problems and exponentiation mod q, but they did not produce an example of a PKC, mainly for lack of finding the right trapdoor information. They did, however, describe a public key method by which certain material could be securely shared over an insecure channel. Their method, which is now called Diffie–Hellman key exchange, is based on the assumption that the discrete logarithm problem (DLP) is difficult to solve. We discuss the DLP in Section 2.2, and then describe Diffie–Hellman key exchange in Section 2.3. In their paper, Diffie and Hellman also defined a variety of cryptanalytic attacks and introduced the important concepts of digital signatures and one-way authentication, which we discuss in Chapter 7 and Section 8.5.

With the publication of [36] in 1976, the race was on to invent a practical public key cryptosystem. Within two years, two major papers describing public key cryptosystems were published: the RSA scheme of Rivest, Shamir, and Adleman [100] and the knapsack scheme of Merkle and Hellman [75]. Of these two, only RSA has withstood the test of time, in the sense that its underlying hard problem of integer factorization is still sufficiently computationally difficult to allow RSA to operate efficiently. By way of contrast, the knapsack system of Merkle and Hellman was shown to be insecure at practical computational levels [114]. However, the cryptanalysis of knapsack systems introduces important links to hard computational problems in the theory of integer lattices that we explore in Chapter 6.

2.2 The discrete logarithm problem

The discrete logarithm problem is a mathematical problem that arises in many settings, including the mod p version described in this section and the elliptic curve version that will be studied later, in Chapter 5. The first published public key construction, due to Diffie and Hellman [36], is based on the discrete logarithm problem in a finite field $\mathbb{F}_p$, where recall that $\mathbb{F}_p$ is a field with a prime number of elements. (See Section 1.4.) For convenience, we interchangeably use the notations $\mathbb{F}_p$ and $\mathbb{Z}/p\mathbb{Z}$ for this field, and we use equality notation for elements of $\mathbb{F}_p$ and congruence notation for elements of $\mathbb{Z}/p\mathbb{Z}$ (cf. Remark 1.24).

[3] The $\mathcal{P} = \mathcal{N}P$ problem is one of the so-called Millennium Prizes, each of which has a $1,000,000 prize attached. See Section 4.7 for more on $\mathcal{P}$ versus $\mathcal{N}P$.

Let p be a (large) prime. Theorem 1.31 tells us that there exists a primitive element g. This means that every nonzero element of $\mathbb{F}_p$ is equal to some power of g. In particular, $g^{p-1} = 1$ by Fermat's little theorem (Theorem 1.25), and no smaller power of g is equal to 1. Equivalently, the list of elements

$$1, g, g^2, g^3, \ldots, g^{p-2} \in \mathbb{F}_p^*$$

is a complete list of the elements in $\mathbb{F}_p^*$ in some order.

Definition. Let g be a primitive root for $\mathbb{F}_p$ and let h be a nonzero element of $\mathbb{F}_p$. The *Discrete Logarithm Problem* (DLP) is the problem of finding an exponent x such that

$$g^x \equiv h \pmod{p}.$$

The number x is called the *discrete logarithm of h to the base g* and is denoted by $\log_g(h)$.

Remark 2.1. An older term for the discrete logarithm is the *index*, denoted by $\operatorname{ind}_g(h)$. The index terminology is still commonly used in number theory. It is also convenient if there is a danger of confusion between ordinary logarithms and discrete logarithms, since, for example, the quantity $\log_2$ frequently occurs in both contexts.

Remark 2.2. The discrete logarithm problem is a well-posed problem, namely to find an integer exponent x such that $g^x = h$. However, if there is one solution, then there are infinitely many, because Fermat's little theorem (Theorem 1.25) tells us that $g^{p-1} \equiv 1 \pmod{p}$. Hence if x is a solution to $g^x = h$, then $x + k(p-1)$ is also a solution for every value of k, because

$$g^{x+k(p-1)} = g^x \cdot (g^{p-1})^k \equiv h \cdot 1^k \equiv h \pmod{p}.$$

Thus $\log_g(h)$ is defined only up to adding or subtracting multiples of $p-1$. In other words, $\log_g(h)$ is really defined modulo $p-1$. It is not hard to verify (Exercise 2.3(a)) that $\log_g$ gives a well-defined function[4]

$$\log_g : \mathbb{F}_p^* \longrightarrow \frac{\mathbb{Z}}{(p-1)\mathbb{Z}}. \tag{2.1}$$

Sometimes, for concreteness, we refer to "the" discrete logarithm as the integer x lying between 0 and $p-2$ satisfying the congruence $g^x \equiv h \pmod{p}$.

Remark 2.3. It is not hard to prove (see Exercise 2.3(b)) that

$$\log_g(ab) = \log_g(a) + \log_g(b) \qquad \text{for all } a, b \in \mathbb{F}_p^*.$$

[4]If you have studied complex analysis, you may have noticed an analogy with the complex logarithm, which is not actually well defined on $\mathbb{C}^*$. This is due to the fact that $e^{2\pi i} = 1$, so $\log(z)$ is well defined only up to adding or subtracting multiples of $2\pi i$. The complex logarithm thus defines an isomorphism from $\mathbb{C}^*$ to the quotient group $\mathbb{C}/2\pi i\mathbb{Z}$, analogous to (2.1).

n	$g^n \bmod p$
1	627
2	732
3	697
4	395
5	182
6	253
7	543
8	760
9	374
10	189

n	$g^n \bmod p$
11	878
12	21
13	934
14	316
15	522
16	767
17	58
18	608
19	111
20	904

h	$\log_g(h)$
1	0
2	183
3	469
4	366
5	356
6	652
7	483
8	549
9	938
10	539

h	$\log_g(h)$
11	429
12	835
13	279
14	666
15	825
16	732
17	337
18	181
19	43
20	722

Table 2.1: Powers and discrete logarithms for $g = 627$ modulo $p = 941$

Thus calling $\log_g$ a "logarithm" is reasonable, since it converts multiplication into addition in the same way as the usual logarithm function. In mathematical terminology, the discrete logarithm $\log_g$ is a group isomorphism from $\mathbb{F}_p^*$ to $\mathbb{Z}/(p-1)\mathbb{Z}$.

Example 2.4. The number $p = 56509$ is prime, and one can check that $g = 2$ is a primitive root modulo p. How would we go about calculating the discrete logarithm of $h = 38679$? The only method that is immediately obvious is to compute

$$2^2,\ 2^3,\ 2^4,\ 2^5,\ 2^6,\ 2^7, \ldots \pmod{56509}$$

until we find some power that equals 38679. It would be difficult to do this by hand, but using a computer, we find that $\log_p(h) = 11235$. You can verify this by calculating $2^{11235} \bmod 56509$ and checking that it is equal to 38679.

Remark 2.5. It must be emphasized that the discrete logarithm bears little resemblance to the continuous logarithm defined on the real or complex numbers. The terminology is still reasonable, because in both instances the process of exponentiation is inverted—but exponentiation modulo p varies in a very irregular way with the exponent, contrary to the behavior of its continuous counterpart. The random-looking behavior of exponentiation modulo p is apparent from even a cursory glance at a table of values such as those in Table 2.1, where we list the first few powers and the first few discrete logarithms for the prime $p = 941$ and the base $g = 627$. The seeming randomness is also illustrated by the scatter graph of $627^i \bmod 941$ pictured in Figure 2.2.

Remark 2.6. Our statement of the discrete logarithm problem includes the assumption that the base g is a primitive root modulo p, but this is not strictly necessary. In general, for any $g \in \mathbb{F}_p^*$ and any $h \in \mathbb{F}_p^*$, the discrete logarithm problem is the determination of an exponent x satisfying $g^x \equiv h \pmod{p}$, assuming that such an x exists.

More generally, rather than taking nonzero elements of a finite field $\mathbb{F}_p$ and multiplying them together or raising them to powers, we can take elements of

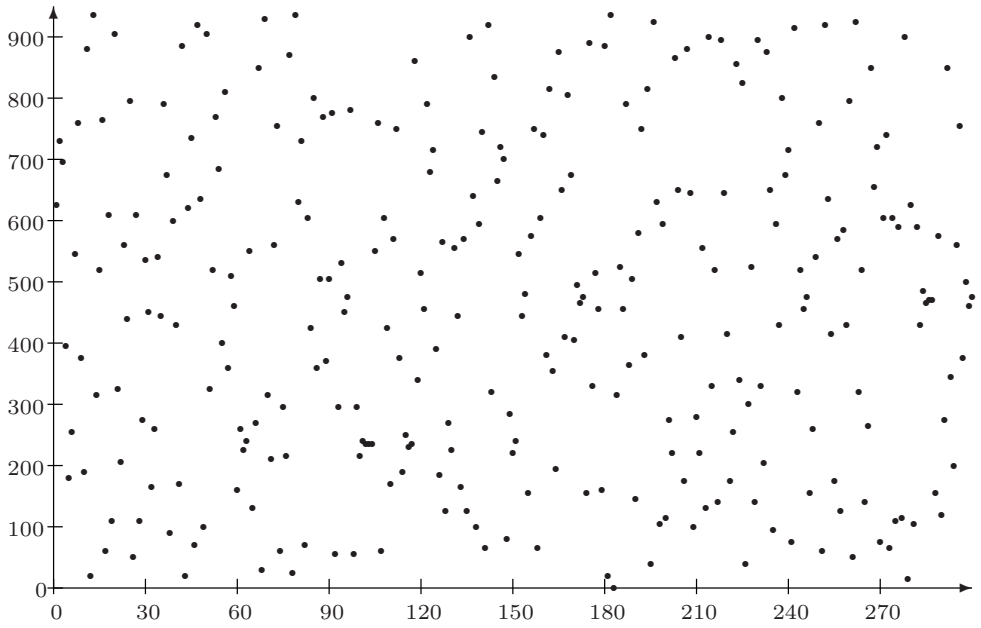

Figure 2.2: Powers $627^i \bmod 941$ for $i = 1, 2, 3, \ldots$

any group and use the group law instead of multiplication. This leads to the most general form of the discrete logarithm problem. (If you are unfamiliar with the theory of groups, we give a brief overview in Section 2.5.)

Definition. Let G be a group whose group law we denote by the symbol $\star$. The *Discrete Logarithm Problem* for G is to determine, for any two given elements g and h in G, an integer x satisfying

$$\underbrace{g \star g \star g \star \cdots \star g}_{x \text{ times}} = h.$$

2.3 Diffie–Hellman key exchange

The Diffie–Hellman key exchange algorithm solves the following dilemma. Alice and Bob want to share a secret key for use in a symmetric cipher, but their only means of communication is insecure. Every piece of information that they exchange is observed by their adversary Eve. How is it possible for Alice and Bob to share a key without making it available to Eve? At first glance it appears that Alice and Bob face an impossible task. It was a brilliant insight of Diffie and Hellman that the difficulty of the discrete logarithm problem for $\mathbb{F}_p^*$ provides a possible solution.

The first step is for Alice and Bob to agree on a large prime p and a nonzero integer g modulo p. Alice and Bob make the values of p and g public knowledge; for example, they might post the values on their web sites, so Eve

knows them, too. For various reasons to be discussed later, it is best if they choose g such that its order in $\mathbb{F}_p^*$ is a large prime. (See Exercise 1.31 for a way of finding such a g.)

The next step is for Alice to pick a secret integer a that she does not reveal to anyone, while at the same time Bob picks an integer b that he keeps secret. Bob and Alice use their secret integers to compute

$$\underbrace{A \equiv g^a \pmod{p}}_{\text{Alice computes this}} \qquad \text{and} \qquad \underbrace{B \equiv g^b \pmod{p}}_{\text{Bob computes this}}.$$

They next exchange these computed values, Alice sends A to Bob and Bob sends B to Alice. Note that Eve gets to see the values of A and B, since they are sent over the insecure communication channel.

Finally, Bob and Alice again use their secret integers to compute

$$\underbrace{A' \equiv B^a \pmod{p}}_{\text{Alice computes this}} \qquad \text{and} \qquad \underbrace{B' \equiv A^b \pmod{p}}_{\text{Bob computes this}}.$$

The values that they compute, A' and B' respectively, are actually the same, since

$$A' \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv B' \pmod{p}.$$

This common value is their exchanged key. The Diffie–Hellman key exchange algorithm is summarized in Table 2.2.

Public Parameter Creation	
A trusted party chooses and publishes a (large) prime p and an integer g having large prime order in $\mathbb{F}_p^*$.	
Private Computations	
Alice	**Bob**
Choose a secret integer a.	Choose a secret integer b.
Compute $A \equiv g^a \pmod{p}$.	Compute $B \equiv g^b \pmod{p}$.
Public Exchange of Values	
Alice sends A to Bob $\longrightarrow$	A
B $\longleftarrow$	Bob sends B to Alice
Further Private Computations	
Alice	**Bob**
Compute the number $B^a \pmod{p}$.	Compute the number $A^b \pmod{p}$.
The shared secret value is $B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \pmod{p}$.	

Table 2.2: Diffie–Hellman key exchange

Example 2.7. Alice and Bob agree to use the prime $p = 941$ and the primitive root $g = 627$. Alice chooses the secret key $a = 347$ and computes

$A = 390 \equiv 627^{347} \pmod{941}$. Similarly, Bob chooses the secret key $b = 781$ and computes $B = 691 \equiv 627^{781} \pmod{941}$. Alice sends Bob the number 390 and Bob sends Alice the number 691. Both of these transmissions are done over an insecure channel, so both $A = 390$ and $B = 691$ should be considered public knowledge. The numbers $a = 347$ and $b = 781$ are not transmitted and remain secret. Then Alice and Bob are both able to compute the number

$$470 \equiv 627^{347 \cdot 781} \equiv A^b \equiv B^a \pmod{941},$$

so 470 is their shared secret.

Suppose that Eve sees this entire exchange. She can reconstitute Alice's and Bob's shared secret if she can solve either of the congruences

$$627^a \equiv 390 \pmod{941} \qquad \text{or} \qquad 627^b \equiv 691 \pmod{941},$$

since then she will know one of their secret exponents. As far as is known, this is the only way for Eve to find the secret shared value without Alice's or Bob's assistance.

Of course, our example uses numbers that are much too small to afford Alice and Bob any real security, since it takes very little time for Eve's computer to check all possible powers of 627 modulo 941. Current guidelines suggest that Alice and Bob choose a prime p having approximately 1000 bits (i.e., $p \approx 2^{1000}$) and an element g whose order is prime and approximately $p/2$. Then Eve will face a truly difficult task.

In general, Eve's dilemma is this. She knows the values of A and B, so she knows the values of g^a and g^b. She also knows the values of g and p, so if she can solve the DLP, then she can find a and b, after which it is easy for her to compute Alice and Bob's shared secret value g^{ab}. It appears that Alice and Bob are safe provided that Eve is unable to solve the DLP, but this is not quite correct. It is true that one method of finding Alice and Bob's shared value is to solve the DLP, but that is not the precise problem that Eve needs to solve. The security of Alice's and Bob's shared key rests on the difficulty of the following, potentially easier, problem.

Definition. Let p be a prime number and g an integer. The *Diffie–Hellman Problem* (DHP) is the problem of computing the value of $g^{ab} \pmod{p}$ from the known values of $g^a \pmod{p}$ and $g^b \pmod{p}$.

It is clear that the DHP is no harder than the DLP. If Eve can solve the DLP, then she can compute Alice and Bob's secret exponents a and b from the intercepted values $A = g^a$ and $B = g^b$, and then it is easy for her to compute their shared key g^{ab}. (In fact, Eve needs to compute only one of a and b.) But the converse is less clear. Suppose that Eve has an algorithm that efficiently solves the DHP. Can she use it to also efficiently solve the DLP? The answer is not known.

2.4 The ElGamal public key cryptosystem

Although the Diffie–Hellman key exchange algorithm provides a method of publicly sharing a random secret key, it does not achieve the full goal of being a public key cryptosystem, since a cryptosystem permits exchange of specific information, not just a random string of bits. The first public key cryptosystem was the RSA system of Rivest, Shamir, and Adleman [100], which they published in 1978. RSA was, and still is, a fundamentally important discovery, and we discuss it in detail in Chapter 3. However, although RSA was historically first, the most natural development of a public key cryptosystem following the Diffie–Hellman paper [36] is a system described by Taher ElGamal in 1985 [38]. The ElGamal public key encryption algorithm is based on the discrete log problem and is closely related to Diffie–Hellman key exchange from Section 2.3. In this section we describe the version of the ElGamal PKC that is based on the discrete logarithm problem for $\mathbb{F}_p^*$, but the construction works quite generally using the DLP in any group. In particular, in Section 5.4.2 we discuss a version of the ElGamal PKC based on elliptic curve groups.

The ElGamal PKC is our first example of a public key cryptosystem, so we proceed slowly and provide all of the details. Alice begins by publishing information consisting of a public key and an algorithm. The public key is simply a number, and the algorithm is the method by which Bob encrypts his messages using Alice's public key. Alice does not disclose her private key, which is another number. The private key allows Alice, and only Alice, to decrypt messages that have been encrypted using her public key.

This is all somewhat vague and applies to any public key cryptosystem. For the ElGamal PKC, Alice needs a large prime number p for which the discrete logarithm problem in $\mathbb{F}_p^*$ is difficult, and she needs an element g modulo p of large (prime) order. She may choose p and g herself, or they may have been preselected by some trusted party such as an industry panel or government agency.

Alice chooses a secret number a to act as her private key, and she computes the quantity

$$A \equiv g^a \pmod{p}.$$

Notice the resemblance to Diffie–Hellman key exchange. Alice publishes her public key A and she keeps her private key a secret.

Now suppose that Bob wants to encrypt a message using Alice's public key A. We will assume that Bob's message m is an integer between 2 and p. (Recall that we discussed how to convert messages into numbers in Section 1.7.2.) In order to encrypt m, Bob first randomly chooses another number k modulo p.[5] Bob uses k to encrypt one, and only one, message, and

[5]Most public key cryptosystems require the use of random numbers in order to operate securely. The generation of random or random-looking integers is actually a delicate process. We discuss the problem of generating pseudorandom numbers in Section 8.2, but for now we ignore this issue and assume that Bob has no trouble generating random numbers modulo p.

then he discards it. The number k is called an *ephemeral key*, since it exists only for the purposes of encrypting a single message.

Bob takes his plaintext message m, his chosen random ephemeral key k, and Alice's public key A and uses them to compute the two quantities

$$c_1 \equiv g^k \pmod{p} \qquad \text{and} \qquad c_2 \equiv mA^k \pmod{p}.$$

(Remember that g and p are public parameters, so Bob also knows their values.) Bob's ciphertext, i.e., his encryption of m, is the pair of numbers (c_1, c_2), which he sends to Alice.

How does Alice decrypt Bob's ciphertext (c_1, c_2)? Since Alice knows a, she can compute the quantity

$$x \equiv c_1^a \pmod{p},$$

and hence also $x^{-1} \pmod{p}$. Alice next multiplies c_2 by x^{-1}, and lo and behold, the resulting value is the plaintext m. To see why, we expand the value of $x^{-1} \cdot c_2$ and find that

$$\begin{aligned}
x^{-1} \cdot c_2 &\equiv (c_1^a)^{-1} \cdot c_2 && \pmod{p}, && \text{since } x \equiv c_1^a \pmod{p},\\
&\equiv (g^{ak})^{-1} \cdot (mA^k) && \pmod{p}, && \text{since } c_1 \equiv g^k,\ c_2 \equiv mA^k \pmod{p},\\
&\equiv (g^{ak})^{-1} \cdot (m(g^a)^k) && \pmod{p}, && \text{since } A \equiv g^a \pmod{p},\\
&\equiv m && \pmod{p}, && \text{since the } g^{ak} \text{ terms cancel out.}
\end{aligned}$$

The ElGamal public key cryptosystem is summarized in Table 2.3.

What is Eve's task in trying to decrypt the message? Eve knows the public parameters p and g, and she also knows the value of $A \equiv g^a \pmod{p}$, since Alice's public key A is public knowledge. If Eve can solve the discrete logarithm problem, she can find a and decrypt the message. Otherwise it appears difficult for Eve to find the plaintext, although there are subtleties, some of which we'll discuss after doing an example with small numbers.

Example 2.8. Alice uses the prime $p = 467$ and the primitive root $g = 2$. She chooses $a = 153$ to be her private key and computes her public key

$$A \equiv g^a \equiv 2^{153} \equiv 224 \pmod{467}.$$

Bob decides to send Alice the message $m = 331$. He chooses an ephemeral key at random, say he chooses $k = 197$, and he computes the two quantities

$$c_1 \equiv 2^{197} \equiv 87 \pmod{467} \qquad \text{and} \qquad c_2 \equiv 331 \cdot 224^{197} \equiv 57 \pmod{467}.$$

The pair $(c_1, c_2) = (87, 57)$ is the ciphertext that Bob sends to Alice.

Alice, knowing $a = 153$, first computes

$$x \equiv c_1^a \equiv 87^{153} \equiv 367 \pmod{467}, \quad \text{and then} \quad x^{-1} \equiv 14 \pmod{467}.$$

Public Parameter Creation	
A trusted party chooses and publishes a large prime p and an element g modulo p of large (prime) order.	
Alice	**Bob**
Key Creation	
Chooses private key $1 \leq a \leq p-1$. Computes $A = g^a \pmod{p}$. Publishes the public key A.	
Encryption	
	Chooses plaintext m. Chooses random ephemeral key k. Uses Alice's public key A to compute $c_1 = g^k \pmod{p}$ and $c_2 = mA^k \pmod{p}$. Sends ciphertext (c_1, c_2) to Alice.
Decryption	
Compute $(c_1^a)^{-1} \cdot c_2 \pmod{p}$. This quantity is equal to m.	

Table 2.3: ElGamal key creation, encryption, and decryption

Finally, she computes

$$c_2 x^{-1} \equiv 57 \cdot 14 \equiv 331 \pmod{467}$$

and recovers the plaintext message m.

Remark 2.9. In the ElGamal cryptosystem, the plaintext is an integer m between 2 and $p-1$, while the ciphertext consists of two integers c_1 and c_2 in the same range. Thus in general it takes twice as many bits to write down the ciphertext as it does to write down the plaintext. We say that ElGamal has a 2-*to*-1 *message expansion*.

It's time to raise an important question. Is the ElGamal system as hard for Eve to attack as the Diffie–Hellman problem? Or, by introducing a clever way of encrypting messages, have we unwittingly opened a back door that makes it easy to decrypt messages without solving the Diffie–Hellman problem? One of the goals of modern cryptography is to identify an underlying hard problem like the Diffie–Hellman problem and to *prove* that a given cryptographic construction like ElGamal is at least as hard to attack as the underlying problem.

In this case we would like to prove that anyone who can decrypt arbitrary ciphertexts created by ElGamal encryption, as summarized in Table 2.3, must also be able to solve the Diffie–Hellman problem. Specifically, we would like to prove the following:

Proposition 2.10. *Fix a prime p and base g to use for ElGamal encryption. Suppose that Eve has access to an oracle that decrypts arbitrary ElGamal ciphertexts encrypted using arbitrary ElGamal public keys. Then she can use the oracle to solve the Diffie–Hellman problem described on page 67.*

Proof. Rather than giving a compact formal proof, we will be more discursive and explain how one might approach the problem of using an ElGamal oracle to solve the Diffie–Hellman problem. Recall that in the Diffie–Hellman problem, Eve is given the two values

$$A \equiv g^a \pmod{p} \qquad \text{and} \qquad B \equiv g^b \pmod{p},$$

and she is required to compute the value of $g^{ab} \pmod{p}$. Keep in mind that she knows both of the values of A and B, but she does not know either of the values a and b.

Now suppose that Eve can consult an ElGamal oracle. This means that Eve can send the oracle a prime p, a base g, a purported public key A, and a purported cipher text (c_1, c_2). Referring to Table 2.3, the oracle returns to Eve the quantity

$$(c_1^a)^{-1} \cdot c_2 \pmod{p}.$$

If Eve wants to solve the Diffie–Hellman problem, what values of c_1 and c_2 should she choose? A little thought shows that $c_1 = B = g^b$ and $c_2 = 1$ are good choices, since with this input, the oracle returns $(g^{ab})^{-1} \pmod{p}$, and then Eve can take the inverse modulo p to obtain $g^{ab} \pmod{p}$, thereby solving the Diffie–Hellman problem.

But maybe the oracle is smart enough to know that it should never decrypt ciphertexts having $c_2 = 1$. Eve can still fool the oracle by sending it random-looking ciphertexts as follows. She chooses an arbitrary value for c_2 and tells the oracle that the public key is A and that the ciphertext is (B, c_2). The oracle returns to her the supposed plaintext m that satisfies

$$m \equiv (c_1^a)^{-1} \cdot c_2 \equiv (B^a)^{-1} \cdot c_2 \equiv (g^{ab})^{-1} \cdot c_2 \pmod{p}.$$

After the oracle tells Eve the value of m, she simply computes

$$m^{-1} \cdot c_2 \equiv g^{ab} \pmod{p}$$

to find the value of $g^{ab} \pmod{p}$. It is worth noting that although, with the oracle's help, Eve has computed $g^{ab} \pmod{p}$, she has done so without knowledge of a or b, so she has solved only the Diffie–Hellman problem, not the discrete logarithm problem. □

Remark 2.11. An attack in which Eve has access to an oracle that decrypts arbitrary ciphertexts is known as a *chosen ciphertext attack*. The preceding proposition shows that the ElGamal system is secure against chosen ciphertext attacks. More precisely, it is secure if one assumes that the Diffie–Hellman problem is hard.

2.5 An overview of the theory of groups

For readers unfamiliar with the theory of groups, we briefly introduce a few basic concepts that should help to place the study of discrete logarithms, both here and in Chapter 5, into a broader context.

We've just spent some time talking about exponentiation of elements in $\mathbb{F}_p^*$. Since exponentiation is simply repeated multiplication, this seems like a good place to start. What we'd like to do is to underline some important properties of multiplication in $\mathbb{F}_p^*$ and to point out that these attributes appear in many other contexts.

The properties are:

- There is an element $1 \in \mathbb{F}_p^*$ satisfying $1 \cdot a = a$ for every $a \in \mathbb{F}_p^*$.
- Every $a \in \mathbb{F}_p^*$ has an inverse $a^{-1} \in \mathbb{F}_p^*$ satisfying $a \cdot a^{-1} = a^{-1} \cdot a = 1$.
- Multiplication is associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in \mathbb{F}_p^*$.
- Multiplication is commutative: $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{F}_p^*$.

Suppose that instead of multiplication in $\mathbb{F}_p^*$, we substitute addition in $\mathbb{F}_p$. We also use 0 in place of 1 and $-a$ in place of a^{-1}. Then all four properties are still true:

- $0 + a = a$ for every $a \in \mathbb{F}_p$.
- Every $a \in \mathbb{F}_p$ has an inverse $-a \in \mathbb{F}_p$ with $a + (-a) = (-a) + a = 0$.
- Addition is associative, $a + (b + c) = (a + b) + c$ for all $a, b, c \in \mathbb{F}_p$.
- Addition is commutative, $a + b = b + a$ for all $a, b \in \mathbb{F}_p$.

Sets and operations that behave similarly to multiplication or addition are so widespread that it is advantageous to abstract the general concept and talk about all such systems at once. The leads to the notion of a group.

Definition. A *group* consists of a set G and a rule, which we denote by $\star$, for combining two elements $a, b \in G$ to obtain an element $a \star b \in G$. The composition operation $\star$ is required to have the following three properties:

[Identity Law] There is an $e \in G$ such that
$$e \star a = a \star e = a \quad \text{for every } a \in G.$$

[Inverse Law] For every $a \in G$ there is a (unique) $a^{-1} \in G$ satisfying
$$a \star a^{-1} = a^{-1} \star a = e.$$

[Associative Law] $a \star (b \star c) = (a \star b) \star c \quad \text{for all } a, b, c \in G.$

If, in addition, composition satisfies the

[Commutative Law] $a \star b = b \star a \quad \text{for all } a, b \in G,$

then the group is called a *commutative group* or an *abelian group*.

If G has finitely many elements, we say that G is a *finite group*. The *order of* G is the number of elements in G; it is denoted by $|G|$ or $\#G$.

Example 2.12. Groups are ubiquitous in mathematics and in the physical sciences. Here are a few examples, the first two repeating those mentioned earlier:

(a) $G = \mathbb{F}_p^*$ and $\star =$ multiplication. The identity element is $e = 1$. Proposition 1.22 tells us that inverses exist. G is a finite group of order $p - 1$.

(b) $G = \mathbb{Z}/N\mathbb{Z}$ and $\star =$ addition. The identity element is $e = 0$ and the inverse of a is $-a$. G is a finite group of order N.

(c) $G = \mathbb{Z}$ and $\star =$ addition. The identity element is $e = 0$ and the inverse of a is $-a$. This group G is an infinite group.

(d) Note that $G = \mathbb{Z}$ and $\star =$ multiplication is not a group, since most elements do not have multiplicative inverses inside $\mathbb{Z}$.

(e) However, $G = \mathbb{R}^*$ and $\star =$ multiplication is a group, since all elements have multiplicative inverses inside $\mathbb{R}^*$.

(f) An example of a noncommutative group is

$$G = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} : a, b, c, d \in \mathbb{R} \text{ and } ad - bc \neq 0 \right\}$$

with operation $\star =$ matrix multiplication. The identity element is $e = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ and the inverse is given by the familiar formula

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \begin{pmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{pmatrix}.$$

Notice that G is noncommutative, since for example, $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ is not equal to $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$.

(g) More generally, we can use matrices of any size. This gives the *general linear group*

$$\mathrm{GL}_n(\mathbb{R}) = \big\{ n\text{-by-}n \text{ matrices } A \text{ with real coefficients and } \det(A) \neq 0 \big\}$$

and operation $\star =$ matrix multiplication. We can form other groups by replacing $\mathbb{R}$ with some other field, for example, the finite field $\mathbb{F}_p$. (See exercise 2.15.) The group $\mathrm{GL}_n(\mathbb{F}_p)$ is clearly a finite group, but computing its order is an interesting exercise.

Let g be an element of a group G and let x be a positive integer. Then g^x means that we apply the group operation x times to the element g,

$$g^x = \underbrace{g \star g \star g \star \cdots \star g}_{x \text{ repetitions}}.$$

For example, exponentiation g^x in the group $\mathbb{F}_p^*$ has the usual meaning, multiply g by itself x times. But "exponentiation" g^x in the group $\mathbb{Z}/N\mathbb{Z}$ means to

add g to itself x times. Admittedly, it is more common to write the quantity "g added to itself x times" as $x \cdot g$, but this is just a matter of notation. The key concept underlying exponentiation in a group is repeated application of the group operation to an element of the group.

It is also convenient to give a meaning to g^x when x is not positive. So if x is a negative integer, we define g^x to be $(g^{-1})^{|x|}$. For $x = 0$, we set $g^0 = e$, the identity element of G.

We now introduce a key concept used in the study of groups.

Definition. Let G be a group and let $a \in G$ be an element of the group. Suppose there exists a positive integer d with the property that $a^d = e$. The smallest such d is called the *order* of a. If there is no such d, then a is said to have *infinite order*.

We next prove two propositions describing important properties of the orders of group elements. These are generalizations of Theorem 1.25 (Fermat's little theorem) and Proposition 1.30, which deal with the group $G = \mathbb{F}_p^*$. The proofs are essentially the same.

Proposition 2.13. *Let G be a finite group. Then every element of G has finite order. Further, if $a \in G$ has order d and if $a^k = e$, then $d \mid k$.*

Proof. Since G is finite, the sequence

$$a, a^2, a^3, a^4, \ldots$$

must eventually contain a repetition. That is, there exist positive integers i and j with $i < j$ such that $a^i = a^j$. Multiplying both sides by a^{-i} and applying the group laws leads to $a^{i-j} = e$. Since $i - j > 0$, this proves that some power of a is equal to e. We let d be the smallest positive exponent satisfying $a^d = e$.

Now suppose that $k \geq d$ also satisfies $a^k = e$. We divide k by d to obtain

$$k = dq + r \qquad \text{with } 0 \leq r < d.$$

Using the fact that $a^k = a^d = e$, we find that

$$e = a^k = a^{dq+r} = (a^d)^q \star a^r = e^q \star a^r = a^r.$$

But d is the smallest positive power of a that is equal to e, so we must have $r = 0$. Therefore $k = dq$, so $d \mid q$. □

Proposition 2.14 (Lagrange's Theorem)**.** *Let G be a finite group and let $a \in G$. Then the order of a divides the order G.*

More precisely, let $n = |G|$ be the order of G and let d be the order of a, i.e., a^d is the smallest positive power of a that is equal to e. Then

$$a^n = e \qquad \textit{and} \qquad d \mid n.$$

Proof. We give a simple proof in the case that G is commutative. For a proof in the general case, see any basic algebra textbook, for example [37, §3.2] or [42, §2.3].

Since G is finite, we can list its elements as

$$G = \{g_1, g_2, \ldots, g_n\}.$$

We now multiply each element of G by a to obtain a new set, which we call S_a,

$$S_a = \{a \star g_1, a \star g_2, \ldots, a \star g_n\}.$$

We claim that the elements of S_a are distinct. To see this, suppose that $a \star g_i = a \star g_j$. Multiplying both sides by a^{-1} yields $g_i = g_j$.[6] Thus S_a contains n distinct elements, which is the same as the number of elements of G. Therefore $S_a = G$, so if we multiply together all of the elements of S_a, we get the same answer as multiplying together all of the elements of G. (Note that we are using the assumption that G is commutative.) Thus

$$(a \star g_1) \star (a \star g_2) \star \cdots \star (a \star g_n) = g_1 \star g_2 \star \cdots \star g_n.$$

We can rearrange the order of the product on the left-hand side (again using the commutativity) to obtain

$$a^n \star g_1 \star g_2 \star \cdots \star g_n = g_1 \star g_2 \star \cdots \star g_n.$$

Now multiplying by $(g_1 \star g_2 \star \cdots \star g_n)^{-1}$ yields $a^n = e$, which proves the first statement, and then the divisibility of n by d follows immediately from Proposition 2.13. □

2.6 How hard is the discrete logarithm problem?

Given a group G and two elements $g, h \in G$, the discrete logarithm problem asks for an exponent x such that that $g^x = h$. What does it mean to talk about the difficulty of this problem? How can we quantify "hard"? A natural measure of hardness is the approximate number of operations necessary for a person or a computer to solve the problem using the most efficient method currently known. For example, suppose that we count the process of computing g^x as a single operation. Then a trial-and-error approach to solving the discrete logarithm problem would be to compute g^x for each $x = 1, 2, 3, \ldots$ and compare the values with h. If g has order n, then this algorithm is guaranteed to find the solution in at most n operations, but if n is large, say $n > 2^{80}$, then it is not a practical algorithm with the computing power available today.

[6] We are being somewhat informal here, as is usually done when one is working with groups. Here is a more formal proof. We are given that $a \star g_i = a \star g_j$. We use this assumption and the group law axioms to compute

$$g_i = e \star g_i = (a^{-1} \star a) \star g_i = a^{-1} \star (a \star g_i) = a^{-1} \star (a \star g_j) = (a^{-1} \star a) \star g_j = e \star g_j = g_j.$$

In practice, unless one were to build a special-purpose machine, the process of computing g^x should not be counted as a single basic operation. Using the fast exponentiation method described in Section 1.3.2, it takes a small multiple of $\log_2(x)$ modular multiplications to compute g^x. Suppose that n and x are k-bit numbers, that is, they are each approximately 2^k. Then the trial-and-error approach actually requires about $k \cdot 2^k$ multiplications. And if we are working in the group $\mathbb{F}_p^*$ and if we treat modular addition as the basic operation, then modular multiplication of two k-bit numbers takes (approximately) k^2 basic operations, so now solving the DLP by trial and error takes a small multiple of $k^2 \cdot 2^k$ basic operations.

We are being somewhat imprecise when we talk about "small multiples" of 2^k or $k \cdot 2^k$ or $k^2 \cdot 2^k$. This is because when we want to know whether a computation is feasible, numbers such as $3 \cdot 2^k$ and $10 \cdot 2^k$ and $100 \cdot 2^k$ mean pretty much the same thing if k is large. The important property is that the constant multiple is fixed as k increases. *Order notation* was invented to make these ideas precise.[7] It is prevalent throughout mathematics and computer science and provides a handy way to get a grip on the magnitude of quantities.

Definition (Order Notation)**.** Let $f(x)$ and $g(x)$ be functions of x taking values that are positive. We say that "f is big-$\mathcal{O}$ of g" and write

$$f(x) = \mathcal{O}\big(g(x)\big)$$

if there are positive constants c and C such that

$$f(x) \le cg(x) \qquad \text{for all } x \ge C.$$

In particular, we write $f(x) = \mathcal{O}(1)$ if $f(x)$ is bounded for all $x \ge C$.

The next proposition gives a method that can sometimes be used to prove that $f(x) = \mathcal{O}\big(g(x)\big)$.

Proposition 2.15. *If the limit*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)}$$

exists (and is finite), then $f(x) = \mathcal{O}\big(g(x)\big)$.

Proof. Let L be the limit. By definition of limit, for any $\epsilon > 0$ there is a constant C_ϵ such that

$$\left| \frac{f(x)}{g(x)} - L \right| < \epsilon \qquad \text{for all } x > C_\epsilon.$$

[7] Although we use the same word for the *order* of a finite group and the *order* of growth of a function, they are two different concepts. Make sure that you don't confuse them.

In particular, taking $\epsilon = 1$, we find that

$$\frac{f(x)}{g(x)} < L + 1 \qquad \text{for all } x > C_1.$$

Hence by definition, $f(x) = \mathcal{O}\big(g(x)\big)$ with $c = L + 1$ and $C = C_1$. □

Example 2.16. We have $2x^3 - 3x^2 + 7 = \mathcal{O}(x^3)$, since

$$\lim_{x\to\infty} \frac{2x^3 - 3x^2 + 7}{x^3} = 2.$$

Similarly, we have $x^2 = \mathcal{O}(2^x)$, since

$$\lim_{x\to\infty} \frac{x^2}{2^x} = 0.$$

(If you don't know the value of this limit, use L'Hôpital's rule twice.)

However, note that we may have $f(x) = \mathcal{O}\big(g(x)\big)$ even if the limit of $f(x)/g(x)$ does not exist. For example, the limit

$$\lim_{x\to\infty} \frac{(x+2)\cos^2(x)}{x}$$

does not exist, but

$$(x+2)\cos^2(x) = \mathcal{O}(x), \quad \text{since} \quad (x+2)\cos^2(x) \le x + 2 \le 2x \quad \text{for all } x \ge 2.$$

Example 2.17. Here are a few more examples of big-$\mathcal{O}$ notation. We leave the verification as an exercise.

(a) $x^2 + \sqrt{x} = \mathcal{O}\left(x^2\right)$.
(b) $5 + 6x^2 - 37x^5 = \mathcal{O}\left(x^5\right)$.
(c) $k^{300} = \mathcal{O}\left(2^k\right)$.
(d) $(\ln k)^{375} = \mathcal{O}\left(k^{0.001}\right)$.
(e) $k^2 2^k = \mathcal{O}\left(e^{2k}\right)$.
(f) $N^{10} 2^N = \mathcal{O}\left(e^N\right)$.

Order notation allows us to define several fundamental concepts that are used to get a rough handle on the computational complexity of mathematical problems.

Definition. Suppose that we are trying to solve a certain type of mathematical problem, where the input to the problem is a number whose size may vary. As an example, consider the *Integer Factorization Problem*, whose input is a number N and whose output is a prime factor of N. We are interested in knowing how long it takes to solve the problem in terms of the size of the input. Typically, one measures the size of the input by its number of bits, since that is how much storage it takes to record the input.

Suppose that there is a constant $A \ge 0$, independent of the size of the input, such that if the input is $\mathcal{O}(k)$ bits long, then it takes $\mathcal{O}(k^A)$ steps to solve the problem. Then the problem is said to be solvable in *polynomial time*.

If we can take $A = 1$, then the problem is solvable in *linear time*, and if we can take $A = 2$, then the problem is solvable in *quadratic time*. Polynomial-time algorithms are considered to be fast algorithms.

On the other hand, if there is a constant $c > 0$ such that for inputs of size $\mathcal{O}(k)$ bits, there is an algorithm to solve the problem in $\mathcal{O}\left(e^{ck}\right)$ steps, then the problem is solvable in *exponential time*. Exponential-time algorithms are considered to be slow algorithms.

Intermediate between polynomial-time algorithms and exponential-time algorithms are *subexponential-time* algorithms. These have the property that for every $\epsilon > 0$, they solve the problem in $\mathcal{O}_\epsilon\left(e^{\epsilon k}\right)$ steps. This notation means that the constants c and C appearing in the definition of order notation are allowed to depend on ϵ. For example, in Chapter 3 we will study a subexponential-time algorithm for the integer factorization problem whose running time is $\mathcal{O}\big(e^{c\sqrt{k \log k}}\big)$ steps.

As a general rule of thumb in cryptography, problems solvable in polynomial time are considered to be "easy" and problems that require exponential time are viewed as "hard," with subexponential time lying somewhere in between. However, bear in mind that these are asymptotic descriptions that are applicable only as the variables become very large. Depending on the big-$\mathcal{O}$ constants and on the size of the input, an exponential problem may be easier than a polynomial problem. We illustrate these general concepts by considering the discrete logarithm problem in various groups.

Example 2.18. We start with our original discrete logarithm problem $g^x = h$ in $G = \mathbb{F}_p^*$. If the prime p is chosen between 2^k and 2^{k+1}, then g, h, and p all require at most k bits, so the problem can be stated in $\mathcal{O}(k)$-bits. (Notice that $\mathcal{O}(k)$ is the same as $\mathcal{O}(\log_2 p)$.)

If we try to solve the DLP using the trial-and-error method mentioned earlier, then it takes $\mathcal{O}(p)$ steps to solve the problem. Since $\mathcal{O}(p) = \mathcal{O}(2^k)$, this algorithm takes exponential time. (If we consider instead multiplication or addition to be the basic operation, then the algorithm takes $\mathcal{O}(k \cdot 2^k)$ or $\mathcal{O}(k^2 \cdot 2^k)$ steps, but these distinctions are irrelevant; the running time is still exponential, since for example it is $\mathcal{O}(3^k)$.)

However, there are faster ways to solve the DLP in $\mathbb{F}_p^*$, some of which are very fast but work only for some primes, while others are less fast, but work for all primes. For example, the Pohlig–Hellman algorithm described in Section 2.9 shows that if $p - 1$ factors entirely into a product of small primes, then the DLP is quite easy. For arbitrary primes, the algorithm described in Section 2.7 solves the DLP in $\mathcal{O}(\sqrt{p} \log p)$ steps, which is much faster than $\mathcal{O}(p)$, but still exponential. Even better is the index calculus algorithm described in Section 3.8. The index calculus solves the DLP in $\mathcal{O}(e^{c\sqrt{(\log p)(\log\log p)}})$ steps, so it is a subexponential algorithm.

Example 2.19. We next consider the DLP in the group $G = \mathbb{F}_p$, where now the group operation is addition. The DLP in this context asks for a solution x

to the congruence

$$x \cdot g \equiv h \pmod{p},$$

where g and h are given elements of $\mathbb{Z}/p\mathbb{Z}$. As described in Section 1.3, we can solve this congruence using the extended Euclidean algorithm (Theorem 1.11) to compute $g^{-1} \pmod{p}$ and setting $x \equiv g^{-1} \cdot h \pmod{p}$. This takes $\mathcal{O}(\log p)$ steps (see Remark 1.15), so there is a linear-time algorithm to solve the DLP in the additive group $\mathbb{F}_p$. This is a very fast algorithm, so the DLP in $\mathbb{F}_p$ with addition is not a good candidate for use as a one-way function in cryptography.

This is an important lesson to learn. The discrete logarithm problems in different groups may display different levels of difficulty for their solution. Thus the DLP in $\mathbb{F}_p$ with addition has a linear-time solution, while the best known general algorithm to solve the DLP in $\mathbb{F}_p^*$ with multiplication is subexponential. In Chapter 5 we discuss another sort of group called an elliptic curve. The discrete logarithm problem for elliptic curves is believed to be even more difficult than the DLP for $\mathbb{F}_p^*$. In particular, if the elliptic curve group is chosen carefully and has N elements, then the best known algorithm to solve the DLP requires $\mathcal{O}(\sqrt{N})$ steps. Thus it currently takes exponential time to solve the elliptic curve discrete logarithm problem (ECDLP).

2.7 A collision algorithm for the DLP

In this section we describe a discrete logarithm algorithm due to Shanks. It is an example of a collision, or meet-in-the-middle, algorithm. Algorithms of this type are discussed in more detail in Sections 4.4 and 4.5. Shanks's algorithm works in any group, not just $\mathbb{F}_p^*$, and the proof that it works is no more difficult for arbitrary groups, so we state and prove it in full generality.

We begin by recalling the running time of the trivial brute-force algorithm to solve the DLP.

Proposition 2.20 (Trivial Bound for DLP)**.** *Let G be a group and let $g \in G$ be an element of order N. (Recall that this means that $g^N = e$ and that no smaller positive power of g is equal to the identity element e.) Then the discrete logarithm problem*

$$g^x = h \tag{2.2}$$

can be solved in $\mathcal{O}(N)$ steps, where each step consists of multiplication by g.

Proof. Simply make a list of the values g^x for $x = 0, 1, 2, \ldots, N-1$. Note that each successive value may be obtained by multiplying the previous value by g. If a solution to $g^x = h$ exists, then h will appear in your list. □

Remark 2.21. If we work in $\mathbb{F}_p^*$, then each computation of $g^x \pmod{p}$ requires $\mathcal{O}((\log p)^k)$ computer operations, where the constant k and the implied big-$\mathcal{O}$ constant depend on the computer and the algorithm used for modular

multiplication. Then the total number of computer steps, or *running time*, is $\mathcal{O}(N(\log p)^k)$. In general, the factor contributed by the $\mathcal{O}((\log p)^k)$ is negligible, so we will suppress it and simply refer to the running time as $\mathcal{O}(N)$.

The idea behind a collision algorithm is to make two lists and look for an element that appears in both lists. For the discrete logarithm problem described in Proposition 2.20, the running time of a collision algorithm is a little more than $\mathcal{O}(\sqrt{N}\,)$ steps, which is a huge savings over $\mathcal{O}(N)$ if N is large.

Proposition 2.22 (Shanks's Babystep–Giantstep Algorithm)**.** *Let G be a group and let $g \in G$ be an element of order $N \ge 2$. The following algorithm solves the discrete logarithm problem $g^x = h$ in $\mathcal{O}(\sqrt{N} \cdot \log N)$ steps.*

(1) *Let $n = 1 + \lfloor \sqrt{N}\,\rfloor$, so in particular, $n > \sqrt{N}$.*

(2) *Create two lists,*

$$\begin{aligned} &\text{List 1:} \quad e,\ g,\ g^2,\ g^3, \ldots,\ g^n, \\ &\text{List 2:} \quad h,\ h \cdot g^{-n},\ h \cdot g^{-2n},\ h \cdot g^{-3n}, \ldots,\ h \cdot g^{-n^2}. \end{aligned}$$

(3) *Find a match between the two lists, say $g^i = hg^{-jn}$.*

(4) *Then $x = i + jn$ is a solution to $g^x = h$.*

Proof. We begin with a couple of observations. First, when creating List 2, we start by computing the quantity $u = g^{-n}$ and then compile List 2 by computing $h, h \cdot u, h \cdot u^2, \ldots, h \cdot u^n$. Thus creating the two lists takes approximately $2n$ multiplications.[8] Second, assuming that a match exists, we can find a match in a small multiple of $\log(n)$ steps using standard sorting and searching algorithms, so Step (3) takes $\mathcal{O}(\log n)$ steps. Hence the total running time for the algorithm is $\mathcal{O}(n \log n) = \mathcal{O}(\sqrt{N}\,\log N)$. For this last step we have used the fact that $n \approx \sqrt{N}$, so

$$n \log n \approx \sqrt{N} \log \sqrt{N} = \frac{1}{2}\sqrt{N} \log N.$$

In order to prove that the algorithm works, we must show that Lists 1 and 2 always have a match. To see this, let x be the unknown solution to $g^x = h$ and write x as

$$x = nq + r \quad \text{with } 0 \le r < n.$$

We know that $1 \le x < N$, so

$$q = \frac{x - r}{n} < \frac{N}{n} < n \quad \text{since } n > \sqrt{N}.$$

[8] Multiplication by g is a "baby step" and multiplication by $u = g^{-n}$ is a "giant step," whence the name of the algorithm.

k	g^k	$h \cdot u^k$
1	9704	347
2	6181	13357
3	5763	12423
4	1128	13153
5	8431	7928
6	16568	1139
7	**14567**	6259
8	2987	12013

k	g^k	$h \cdot u^k$
9	15774	16564
10	12918	11741
11	16360	16367
12	13259	7315
13	4125	2549
14	16911	10221
15	4351	16289
16	1612	4062

k	g^k	$h \cdot u^k$
17	10137	10230
18	17264	3957
19	4230	9195
20	9880	13628
21	9963	10126
22	15501	5416
23	6854	13640
24	15680	5276

k	g^k	$h \cdot u^k$
25	4970	12260
26	9183	6578
27	10596	7705
28	2427	1425
29	6902	6594
30	11969	12831
31	6045	4754
32	7583	**14567**

Table 2.4: Babystep–giantstep to solve $9704^x \equiv 13896 \pmod{17389}$

Hence we can rewrite the equation $g^x = h$ as

$$g^r = h \cdot g^{-qn} \qquad \text{with } 0 \le r < n \text{ and } 0 \le q < n.$$

Thus g^r is in List 1 and $h \cdot g^{-qn}$ is in List 2, which shows that Lists 1 and 2 have a common element. □

Example 2.23. We illustrate Shanks's babystep–giantstep method by using it to solve the discrete logarithm problem

$$g^x = h \quad \text{in } \mathbb{F}_p^* \text{ with } \quad g = 9704, \quad h = 13896, \quad \text{and} \quad p = 17389.$$

The number 9704 has order 1242 in $\mathbb{F}_{17389}^*$. Set $n = \lfloor \sqrt{1242} \rfloor + 1 = 36$ and $u = g^{-n} = 9704^{-36} = 2494$. Table 2.4 lists the values of g^k and $h \cdot u^k$ for $k = 1, 2, \ldots$. From the table we find the collision

$$9704^7 = 14567 = 13896 \cdot 2494^{32} \quad \text{in } \mathbb{F}_{17389}.$$

Using the fact that $2494 = 9704^{-36}$, we compute

$$13896 = 9704^7 \cdot 2494^{-32} = 9704^7 \cdot (9704^{36})^{32} = 9704^{1159} \quad \text{in } \mathbb{F}_{17389}.$$

Hence $x = 1159$ solves the problem $9704^x = 13896$ in $\mathbb{F}_{17389}$.

2.8 The Chinese remainder theorem

The Chinese remainder theorem describes the solutions to a system of simultaneous linear congruences. The simplest situation is a system of two congruences,

$$x \equiv a \pmod{m} \qquad \text{and} \qquad x \equiv b \pmod{n}, \tag{2.3}$$

with $\gcd(m, n) = 1$, in which case the Chinese remainder theorem says that there is a unique solution modulo mn.

The first recorded instance of a problem of this type appears in a Chinese mathematical work from the late third or early fourth century. It actually deals with the harder problem of three simultaneous congruences.