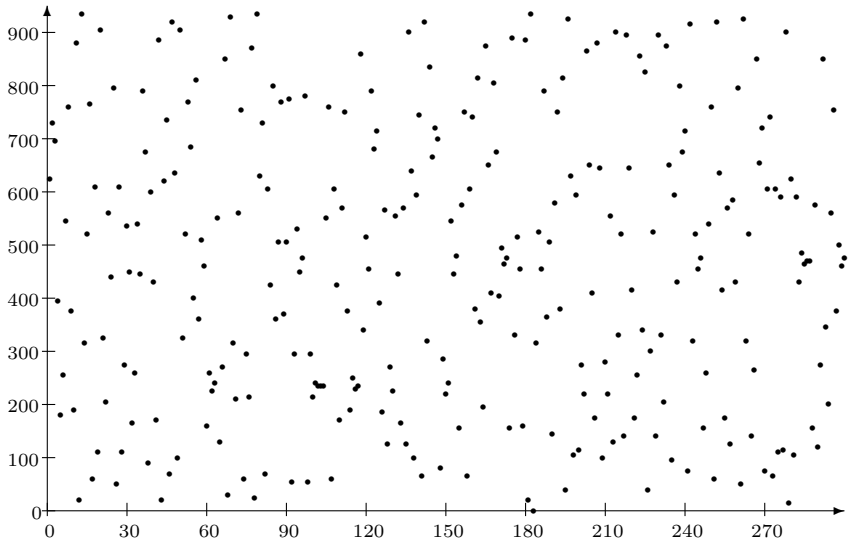

Figure 2.2: Powers $627^i \bmod 941$ for $i = 1, 2, 3, \ldots$

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

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

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

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

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.

4.4 Collision algorithms and meet-in-the-middle attacks

A simple, yet surprisingly powerful, search method is based on the observation that it is usually much easier to find matching objects than it is to find a particular object. Methods of this sort go by many names, including meet-in-the-middle attacks and collision algorithms.

4.4.1 The birthday paradox

The fundamental idea behind collision algorithms is strikingly illustrated by the famous birthday paradox. In a random group of 40 people, consider the following two questions:

(1) What is the probability that someone has the same birthday as you?

(2) What is the probability that at least two people share the same birthday?

It turns out that the answers to (1) and (2) are very different. As a warm-up, we start by answering the easier first question.

A rough answer is that since any one person has a 1-in-365 chance of sharing your birthday, then in a crowd of 40 people, the probability of someone having your birthday is approximately $\frac{40}{365} \approx 11\%$. However, this is an overestimate, since it double counts the occurrences of more than one person in the crowd sharing your birthday.[14] The exact answer is obtained by computing the probability that none of the people share your birthday and then subtracting that value from 1.

$$\begin{aligned}
\Pr\begin{pmatrix}\text{someone has}\\ \text{your birthday}\end{pmatrix} &= 1 - \Pr\begin{pmatrix}\text{none of the 40 people}\\ \text{has your birthday}\end{pmatrix} \\
&= 1 - \prod_{i=1}^{40} \Pr\begin{pmatrix}i^{\text{th}}\text{ person does not}\\ \text{have your birthday}\end{pmatrix} \\
&= 1 - \left(\frac{364}{365}\right)^{40} \\
&\approx 10.4\%.
\end{aligned}$$

Thus among 40 strangers, there is only slightly better than a 10% chance that one of them shares your birthday.

Now consider the second question, in which you win if any two of the people in the group have the same birthday. Again it is easier to compute the probability that all 40 people have different birthdays. However, the computation changes because we now require that the i^{th} person have a birthday

[14] If you think that $\frac{40}{365}$ is the right answer, think about the same situation with 366 people. The probability that someone shares your birthday cannot be $\frac{366}{365}$, since that's larger than 1.

that is different from all of the previous $i-1$ people's birthdays. Hence the calculation is

$$\begin{aligned}
\Pr\begin{pmatrix}\text{two people have}\\ \text{the same birthday}\end{pmatrix} &= 1 - \Pr\begin{pmatrix}\text{all 40 people have}\\ \text{different birthdays}\end{pmatrix} \\
&= 1 - \prod_{i=1}^{40} \Pr\begin{pmatrix}i^{\text{th}}\text{ person does not have}\\ \text{the same birthday as any}\\ \text{of the previous } i-1 \text{ people}\end{pmatrix} \\
&= 1 - \prod_{i=1}^{40} \frac{365-(i-1)}{365} \\
&= 1 - \frac{365}{365}\cdot\frac{364}{365}\cdot\frac{363}{365}\cdots\frac{326}{365} \\
&\approx 89.1\%.
\end{aligned}$$

Thus among 40 strangers, there is almost a 90% chance that two of them share a birthday.

The only part of this calculation that merits some comment is the formula for the probability that the i^{th} person has a birthday different from any of the previous $i-1$ people. Among the 365 possible birthdays, note that the previous $i-1$ people have taken up $i-1$ of them. Hence the probability that the i^{th} person has his or her birthday among the remaining $365-(i-1)$ days is

$$\frac{365-(i-1)}{365}.$$

Most people tend to assume that questions (1) and (2) have essentially the same answer. The fact that they do not is called the *birthday paradox*. In fact, it requires only 23 people to have a better than 50% chance of a matched birthday, while it takes 253 people to have better than a 50% chance of finding someone who has your birthday.

4.4.2 A collision theorem

Cryptographic applications of collision algorithms are generally based on the following setup. Bob has a box that contains N numbers. He chooses n distinct numbers from the box and puts them in a list. He then makes a second list by choosing m (not necessarily distinct) numbers from the box. The remarkable fact is that if n and m are each slightly larger than $\sqrt{N}$, then it is very likely that the two lists contain a common element.

We start with an elementary result that illustrates the sort of calculation that is used to quantify the probability of success of a collision algorithm.

Theorem 4.38 (Collision Theorem). *An urn contains N balls, of which n are red and $N-n$ are blue. Bob randomly selects a ball from the urn, replaces it in the urn, randomly selects a second ball, replaces it, and so on. He does this until he has looked at a total of m balls.*

(a) *The probability that Bob selects at least one red ball is*

$$\Pr(\textit{at least one red}) = 1 - \left(1 - \frac{n}{N}\right)^m. \tag{4.28}$$

(b) *A lower bound for the probability* (4.28) *is*

$$\Pr(\textit{at least one red}) \ge 1 - e^{-mn/N}. \tag{4.29}$$

If N is large and if m and n are not too much larger than $\sqrt{N}$ (e.g., $m, n < 10\sqrt{N}$), then (4.29) *is almost an equality.*

Proof. Each time Bob selects a ball, his probability of choosing a red one is $\frac{n}{N}$, so you might think that since he chooses m balls, his probability of getting a red one is $\frac{mn}{N}$. However, a small amount of thought shows that this must be incorrect. For example, if m is large, this would lead to a probability that is larger than 1. The difficulty, just as in the birthday example in Section 4.4.1, is that we are overcounting the times that Bob happens to select more than one red ball. The correct way to calculate is to compute the probability that Bob chooses only blue balls and then subtract this complementary probability from 1. Thus

$$\begin{aligned}
\Pr\begin{pmatrix}\text{at least one red}\\ \text{ball in } m \text{ attempts}\end{pmatrix} &= 1 - \Pr(\text{all } m \text{ choices are blue}) \\
&= 1 - \prod_{i=1}^{m} \Pr(i^{\text{th}} \text{ choice is blue}) \\
&= 1 - \prod_{i=1}^{m} \left(\frac{N-n}{N}\right) \\
&= 1 - \left(1 - \frac{n}{N}\right)^m.
\end{aligned}$$

This completes the proof of (a).

For (b), we use the inequality

$$e^{-x} \ge 1 - x \qquad \text{for all } x \in \mathbb{R}.$$

(See Exercise 4.36(a) for a proof.) Setting $x = n/N$ and raising both sides of the inequality to the m^{th} power shows that

$$1 - \left(1 - \frac{n}{N}\right)^m \ge 1 - (e^{-n/N})^m = 1 - e^{-mn/N},$$

which proves the important inequality in (b). We leave it to the reader (Exercise 4.36(b)) to prove that the inequality is close to being an equality if m and n is not too large compared to $\sqrt{N}$. □

In order to connect Theorem 4.38 with the problem of finding a match in two lists of numbers, we view the list of numbers as an urn containing N

numbered blue balls. After making our first list of n different numbered balls, we repaint those n balls with red paint and return them to the box. The second list is constructed by drawing m balls out of the urn one at a time, noting their number and color, and then replacing them. The probability of selecting at least one red ball is the same as the probability of a matched number on the two lists.

Example 4.39. A deck of cards is shuffled and eight cards are dealt face up. Bob then takes a second deck of cards and chooses eight cards at random, replacing each chosen card before making the next choice. What is Bob's probability of matching one of the cards from the first deck?

We view the eight dealt cards from the first deck as "marking" those same cards in the second deck. So our "urn" is the second deck, the "red balls" are the eight marked cards in the second deck, and the "blue balls" are the other 48 cards in the second deck. Theorem 4.38(a) tells us that

$$\Pr(\text{a match}) = 1 - \left(1 - \frac{8}{52}\right)^8 \approx 73.7\%.$$

The approximation in Theorem 4.38(b) gives a lower bound of 70.8%.

Suppose instead that Bob deals ten cards from the first deck and chooses only five cards from the second deck. Then

$$\Pr(\text{a match}) = 1 - \left(1 - \frac{10}{52}\right)^5 \approx 65.6\%.$$

Example 4.40. A set contains 10 billion elements. Bob randomly selects two lists of 100,000 elements each from the set. What is the (approximate) probability that there will be a match between the two lists? Formula (4.28) in Theorem 4.38(a)says that

$$\Pr(\text{a match}) = 1 - \left(1 - \frac{100{,}000}{10^{10}}\right)^{100{,}000} \approx 0.632122.$$

The approximate lower bound given by the formula (4.29) in Theorem 4.38(b) is 0.632121. As you can see, the approximation is quite accurate.

It is interesting to observe that if Bob doubles the number of elements in each list to 200,000, then his probability of getting a match increases quite substantially to 98.2%. And if he triples the number of elements in each list to 300,000, then the probability of a match is 99.988%. This rapid increase reflects that fact that the exponential function in (4.29) decreases very rapidly as soon as mn becomes larger than N.

Example 4.41. A set contains N objects. Bob randomly chooses n of them, makes a list of his choices, replaces them, and then chooses another n of them. How large should he choose n to give himself a 50% chance of getting a match? How about if he wants a 99.99% chance of getting a match?

For the first question, Bob uses the reasonably accurate lower bound of formula (4.29) to set

$$\Pr(\text{match}) \approx 1 - e^{-n^2/N} = \frac{1}{2}.$$

It is easy to solve this for n:

$$e^{-n^2/N} = \frac{1}{2} \quad \Longrightarrow \quad -\frac{n^2}{N} = \ln\left(\frac{1}{2}\right) \quad \Longrightarrow \quad n = \sqrt{N \cdot \ln 2} \approx 0.83\sqrt{N}.$$

Thus it is enough to create lists that are a bit shorter than $\sqrt{N}$ in length.

The second question is similar, but now Bob solves

$$\Pr(\text{match}) \approx 1 - e^{-n^2/N} = 0.9999 = 1 - 10^{-4}.$$

The solution is

$$n = \sqrt{N \cdot \ln 10^4} \approx 3.035 \cdot \sqrt{N}.$$

Remark 4.42. Algorithms that rely on finding matching elements from within one or more lists go by a variety of names, including *collision algorithm*, *meet-in-the-middle algorithm*, *birthday paradox algorithm*, and *square root algorithm*. The last refers to the fact that the running time of a collision algorithm is generally a small multiple of the square root of the running time required by an exhaustive search. The connection with birthdays was briefly discussed in Section 4.4.1; see also Exercise 4.34. When one of these algorithms is used to break a cryptosystem, the word "algorithm" is often replaced by the word "attack," so cryptanalysts refer to *meet-in-the-middle attacks*, *square root attacks*, etc.

Remark 4.43. Collision algorithms tend to take approximately $\sqrt{N}$ steps in order to find a collision among N objects. A drawback of these algorithms is that they require creation of one or more lists of size approximately $\sqrt{N}$. When N is large, providing storage for $\sqrt{N}$ numbers may be more of an obstacle than doing the computation. In Section 4.5 we describe a collision method due to Pollard that, at the cost of a small amount of extra computation, requires essentially no storage.

4.4.3 A discrete logarithm collision algorithm

There are many applications of collision algorithms to cryptography. These may involve searching a space of keys or plaintexts or ciphertexts, or for public key cryptosystems, they may be aimed at solving the underlying hard mathematical problem. The baby step–giant step algorithm described in Section 2.7 is an example of a collision algorithm that is used to solve the discrete logarithm problem. In this section we further illustrate the general theory by formulating an abstract randomized collision algorithm to solve the discrete

logarithm problem. For the finite field $\mathbb{F}_p$, it solves the discrete logarithm problem (DLP) in approximately $\sqrt{p}$ steps.

Of course, the index calculus described in Section 3.8 solves the DLP in $\mathbb{F}_p$ much more rapidly than this. But there are other groups, such as elliptic curve groups (see Chapter 5), for which collision algorithms are the fastest known way to solve the DLP. This explains why elliptic curve groups are used in cryptography; at present, the DLP in an elliptic curve group is much harder than the DLP in $\mathbb{F}_p^*$ if the groups have about the same size.

It is also worth pointing out that, in a certain sense, there cannot exist a general algorithm to solve the DLP in an arbitrary group with N elements in fewer than $\mathcal{O}\bigl(\sqrt{N}\,\bigr)$ steps. This is the so-called *black box DLP*, in which you are given a box that performs the group operations, but you're not allowed to look inside the box to see how it is doing the computations.

Proposition 4.44. *Let G be a group and let $h \in G$ be an element of order N, i.e., $h^N = e$ and no smaller power of h is equal to e. Then, assuming that the discrete logarithm problem*

$$h^x = b \tag{4.30}$$

has a solution, a solution can be found in $\mathcal{O}\bigl(\sqrt{N}\,\bigr)$ steps, where each step is an exponentiation in the group G. (Note that since $h^N = 1$, the powering algorithm from Section 1.3.2 *lets us raise h to any power using fewer than $2\log_2 N$ group multiplications.)*

Proof. The idea is to write x as $x = y - z$ and look for a solution to

$$h^y = b \cdot h^z.$$

We do this by making a list of h^y values and a list of $b \cdot h^z$ values and looking for a match between the two lists.

We begin by choosing random exponents $y_1, y_2, \ldots, y_n$ between 1 and N and computing the values

$$h^{y_1},\ h^{y_2},\ h^{y_3},\ \ldots,\ h^{y_n} \quad \text{in } G. \tag{4.31}$$

Note that all of the values (4.31) are in the set

$$S = \{1, h, h^2, h^3, \ldots, h^{N-1}\},$$

so (4.31) is a selection of (approximately) n elements of S. In terms of the collision theorem (Theorem 4.38), we view S as an urn containing N balls and the list (4.31) as a way of coloring n of those balls red.

Next we choose additional random exponents $z_1, z_2, \ldots, z_n$ between 1 and k and compute the quantities

$$b \cdot h^{z_1},\ b \cdot h^{z_2},\ b \cdot h^{z_3},\ \ldots,\ b \cdot h^{z_n} \quad \text{in } G. \tag{4.32}$$

Since we are assuming that (4.30) has a solution, i.e., b is equal to some power of h, it follows that each of the values $b \cdot h^{z_i}$ is also in the set S. Thus

the list (4.32) may be viewed as selecting n elements from the urn, and we would like to know the probability of selecting at least one red ball, i.e., the probability that at least one element in the list (4.32) matches an element in the list (4.31). The collision theorem (Theorem 4.38) says that

$$\Pr\begin{pmatrix}\text{at least one match}\\ \text{between (4.31) and (4.32)}\end{pmatrix} \approx \left(1-\frac{n}{N}\right)^n \approx 1-e^{-n^2/N}.$$

Thus if we choose (say) $n \approx 3\sqrt{N}$, then our probability of getting a match is approximately 99.98%, so we are almost guaranteed a match. Or if that is not good enough, take $n \approx 5\sqrt{N}$ to get a probability of success greater than $1-10^{-10}$. Notice that as soon as we find a match between the two lists, say $h^y = b \cdot h^z$, then we have solved the discrete logarithm problem (4.30) by setting $x = y - z$.[15]

How long does it take us to find this solution? Each of the lists (4.31) and (4.32) has n elements, so it takes approximately $2n$ steps to assemble each list. More precisely, each element in each list requires us to compute h^i for some value of i between 1 and N, and it takes approximately $2\log_2(i)$ group multiplications to compute h^i using the fast exponentiation algorithm described in Section 1.3.2. (Here $\log_2$ is the logarithm to the base 2.) Thus it takes approximately $4n\log_2(N)$ multiplications to assemble the two lists. In addition, it takes about $\log_2(n)$ steps to check whether an element of the second list is in the first list (e.g., sort the first list), so $n\log_2(n)$ comparisons altogether. Hence the total computation time is approximately

$$4n\log_2(N) + n\log_2(n) = n\log_2(N^4 n) \text{ steps.}$$

Taking $n \approx 3\sqrt{N}$, which as we have seen gives us a 99.98% chance of success, we find that

$$\text{Computation Time} \approx 13.5 \cdot \sqrt{N} \cdot \log_2(1.3 \cdot N). \qquad \square$$

Example 4.45. We do an example with small numbers to illustrate the use of collisions. We solve the discrete logarithm problem

$$2^x = 390 \quad \text{in the finite field } \mathbb{F}_{659}.$$

The number 2 has order 658 moduluo 659, so it is a primitive root. In this example $h = 2$ and $b = 390$. We choose random exponents t and compute the values of h^t and $b \cdot h^t$ until we get a match. The results are compiled in Table 4.9. We see that

$$2^{83} = 390 \cdot 2^{564} = 422 \quad \text{in } \mathbb{F}_{659}.$$

[15] If this value of x happens to be negative and we want a positive solution, we can always use the fact that $h^N = 1$ to replace it with $x = y - z + N$.

t	h^t	$b \cdot h^t$
564	410	**422**
469	357	181
276	593	620
601	416	126
9	512	3
350	445	233

t	h^t	$b \cdot h^t$
53	10	605
332	651	175
178	121	401
477	450	206
503	116	428
198	426	72

t	h^t	$b \cdot h^t$
513	164	37
71	597	203
314	554	567
581	47	537
371	334	437
83	**422**	489

Table 4.9: Solving $2^x = 390$ in $\mathbb{F}_{659}$ with random exponent collisions

Hence using two lists of length 18, we have solved a discrete logarithm problem in $\mathbb{F}_{659}$. (We had a 39% chance of getting a match with lists of length 18, so we were a little bit lucky.) The solution is

$$2^{83} \cdot 2^{-564} = 2^{-481} = 2^{177} = 390 \quad \text{in } \mathbb{F}_{659}.$$

4.5 Pollard's ρ method

As we noted in Remark 4.43, collision algorithms tend to require a considerable amount of storage. A beautiful idea of Pollard often allows one to use almost no storage, at the cost of a small amount of extra computation. We explain the basic idea of Pollard's method and then illustrate it by yet again solving a small instance of the discrete logarithm problem in $\mathbb{F}_p$.

4.5.1 Abstract formulation of Pollard's ρ method

We begin in an abstract setting. Let S be a finite set and let

$$f : S \longrightarrow S$$

be a function that does a good job at mixing up the elements of S. Suppose that we start with some element $x \in S$ and we repeatedly apply f to create a sequence of elements

$$x_0 = x, \quad x_1 = f(x_0), \quad x_2 = f(x_1), \quad x_3 = f(x_2), \quad x_4 = f(x_3), \quad \ldots.$$

In other words,

$$x_i = (\underbrace{f \circ f \circ f \circ \cdots \circ f}_{i \text{ iterations of } f})(x).$$

The map f from S to itself is an example of a *discrete dynamical system*. The sequence

$$x_0, x_1, x_2, x_3, x_4, \ldots \tag{4.33}$$

is called the (*forward*) *orbit of* x by the map f and is denoted by $O_f^+(x)$.