

Chapter 19 Zero-Knowledge Techniques

19.1 The Basic Setup

A few years ago, it was reported that some thieves set up a fake automatic teller machine at a shopping mall.

When a person inserted a bank card and typed in an identification number, the machine recorded the information but responded with the message that it could not accept the card. The thieves then made counterfeit bank cards and went to legitimate teller machines and withdrew cash, using the identification numbers they had obtained.

How can this be avoided? There are several situations where someone reveals a secret identification number or password in order to complete a transaction. Anyone who obtains this secret number, plus some (almost public) identification information (for example, the information on a bank card), can masquerade as this person. What is needed is a way to use the secret number without giving any information that can be reused by an eavesdropper. This is where zero-knowledge techniques come in.

The basic challenge-response protocol is best illustrated by an example due to Quisquater, Guillou, and Berson [Quisquater et al.]. Suppose there is a tunnel with a door, as in [Figure 19.1](#). Peggy (the prover) wants to prove to Victor (the verifier) that she can go through the door without giving any information to Victor about how she does it. She doesn't even want to let Victor know which direction she can pass through the door (otherwise, she

could simply walk down one side and emerge from the other). They proceed as follows. Peggy enters the tunnel and goes down either the left side or the right side of the tunnel. Victor waits outside for a minute, then comes in and stands at point B. He calls out “Left” or “Right” to Peggy. Peggy then comes to point B by the left or right tunnel, as requested. This entire protocol is repeated several times, until Victor is satisfied. In each round, Peggy chooses which side she will go down, and Victor randomly chooses which side he will request.

Figure 19.1 The Tunnel Used in the Zero-Knowledge Protocol

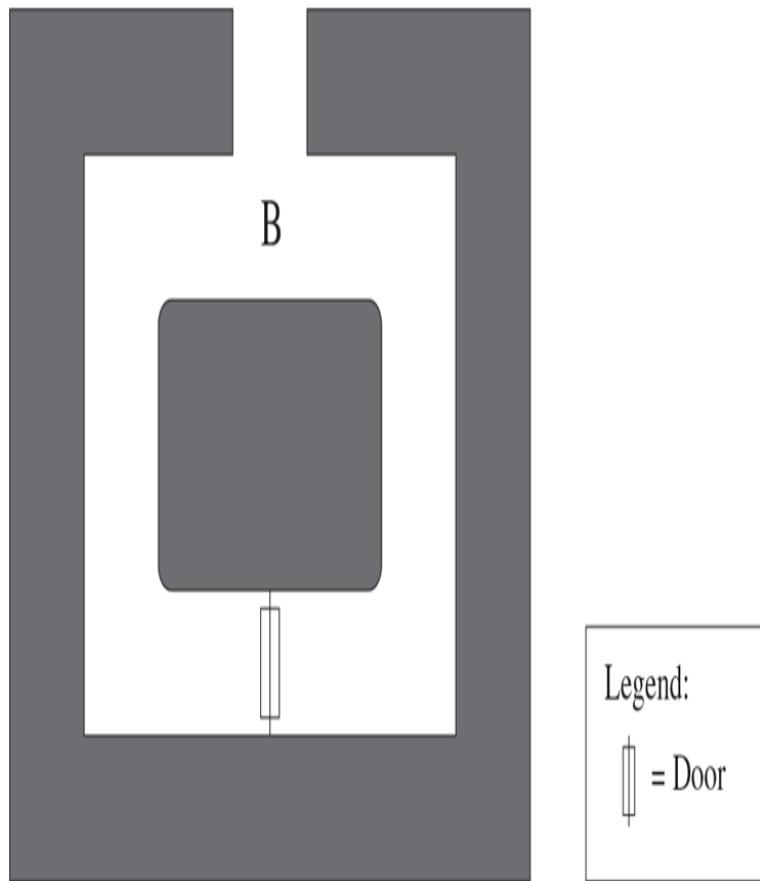


Figure 19.1 Full Alternative Text

Since Peggy must choose to go down the left or right side before she knows what Victor will say, she has only a 50% chance of fooling Victor if she doesn't know how to go through the door. Therefore, if Peggy comes out the correct side for each of 10 repetitions, there is only one chance in $2^{10} = 1024$ that Peggy doesn't know how to go through the door. At this point, Victor is probably convinced, though he could try a few more times just to be sure.

Suppose Eve is watching the proceedings on a video monitor carried by Victor. She will not be able to use anything she sees to convince Victor or anyone else that she, too, can go through the door. Moreover, she might not even be convinced that Peggy can go through the door. After all, Peggy and Victor could have planned the sequence of rights and lefts ahead of time. By this reasoning, there is no useful information that Victor obtains that can be transmitted to anyone.

Note that there is never a proof, in a strict mathematical sense, that Peggy can go through the door. But there is overwhelming evidence, obtained through a series of challenges and responses. This is a feature of zero-knowledge “proofs.”

There are several mathematical versions of this procedure, but we'll concentrate on one of them. Let $n = pq$ be the product of two large primes. Let y be a square mod n with $\gcd(y, n) = 1$. Recall that finding square roots mod n is hard; in fact, finding square roots mod n is equivalent to factoring n (see [Section 3.9](#)). However, Peggy claims to know a square root s of y . Victor wants to verify this, but Peggy does not want to reveal s . Here is the method:

1. Peggy chooses a random number r_1 and lets $r_2 \equiv sr_1^{-1} \pmod{n}$, so

$$r_1 r_2 \equiv s \pmod{n}.$$

(We may assume that $\gcd(r_1, n) = 1$, so $r_1^{-1} \pmod{n}$ exists; otherwise, Peggy has factored n .) She computes

$$x_1 \equiv r_1^2, \quad x_2 \equiv r_2^2 \pmod{n}$$

and sends x_1 and x_2 to Victor.

2. Victor checks that $x_1 x_2 \equiv y \pmod{n}$, then chooses either x_1 or x_2 and asks Peggy to supply a square root of it. He checks that it is an actual square root.
3. The first two steps are repeated several times, until Victor is convinced.

Of course, if Peggy knows s , the procedure proceeds without problems. But what if Peggy doesn't know a square root of y ? She can still send Victor two numbers x_1 and x_2 with $x_1 x_2 \equiv y$. If she knows a square root of x_1 and a square root of x_2 , then she knows a square root of $y \equiv x_1 x_2$. Therefore, for at least one of them, she does not know a square root. At least half the time, Victor is going to ask her for a square root she doesn't know. Since computing square roots is hard, she is not able to produce the desired answer, and therefore Victor finds out that she doesn't know s .

Suppose, however, that Peggy predicts correctly that Victor will ask for a square root of x_2 . Then she chooses a random r_2 , computes $x_2 \equiv r_2^2 \pmod{n}$, and lets $x_1 \equiv yx_2^{-1} \pmod{n}$. She sends x_1 and x_2 to Victor, and everything works. This method gives Peggy a 50% chance of fooling Victor on any given round, but it requires her to guess which number Victor will request each time. As soon as she fails, Victor will find out that she doesn't know s .

If Victor verifies that Peggy knows a square root, does he obtain any information that can be used by someone else? No, since in any step he is only learning the square root of a random square, not a square root of y . Of course, if Peggy uses the same random numbers more than once, he could find out the square roots of both x_1

and x_2 and hence a square root of y . So Peggy should be careful in her choice of random numbers.

Suppose Eve is listening. She also will only learn square roots of random numbers. If she tries to use the same sequence of random numbers to masquerade as Peggy, she needs to be asked for the square roots of exactly the same sequence of x_1 's and x_2 's. If Victor asks for a square root of an x_1 in place of an x_2 at one step, for example, Eve will not be able to supply it.

19.2 The Feige-Fiat-Shamir Identification Scheme

The preceding protocol requires several communications between Peggy and Victor. The Feige-Fiat-Shamir method reduces this number and uses a type of parallel verification. This then is used as the basis of an identification scheme.

Again, let $n = pq$ be the product of two large primes.

Peggy has secret numbers s_1, \dots, s_k . Let $v_i \equiv s_i^{-2} \pmod{n}$ (we assume $\gcd(s_i, n) = 1$). The numbers v_i are sent to Victor. Victor will try to verify that Peggy knows the numbers s_1, \dots, s_k . Peggy and Victor proceed as follows:

1. Peggy chooses a random integer r , computes $x \equiv r^2 \pmod{n}$ and sends x to Victor.
2. Victor chooses numbers b_1, \dots, b_k with each $b_i \in \{0, 1\}$. He sends these to Peggy.
3. Peggy computes $y \equiv rs_1^{b_1}s_2^{b_2} \cdots s_k^{b_k} \pmod{n}$ and sends y to Victor.
4. Victor checks that $x \equiv y^2 v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k} \pmod{n}$.
5. Steps 1 through 4 are repeated several times (each time with a different r).

Consider the case $k = 1$. Then Peggy is asked for either r or rs_1 . These are two random numbers whose quotient is a square root of v_1 . Therefore, this is essentially the same idea as the simplified scheme discussed previously, with quotients instead of products.

Now let's analyze the case of larger k . Suppose, for example, that Victor sends $b_1 = 1, b_2 = 1, b_4 = 1$, and all other $b_i = 0$. Then Peggy must produce $y \equiv rs_1s_2s_4$

, which is a square root of $xv_1^{-1}v_2^{-1}v_4^{-1}$. In fact, in each round, Victor is asking for a square root of a number of the form $xv_{i_1}^{-1}v_{i_2}^{-1}\cdots v_{i_j}^{-1}$. Peggy can supply a square root if she knows $r, s_{i_1}, \dots, s_{i_j}$. If she doesn't, she will have a hard time computing a square root.

If Peggy doesn't know any of the numbers s_1, \dots, s_k (the likely scenario also if someone other than Peggy is pretending to be Peggy), she could guess the string of bits that Victor will send. Suppose she guesses correctly, before she sends x . She lets y be a random number and declares $x \equiv y^2v_1^{b_1}v_2^{b_2}\cdots v_k^{b_k} \pmod{n}$. When Victor sends the string of bits, Peggy sends back the value of y . Of course, the verification congruence is satisfied. But if Peggy guesses incorrectly, she will need to modify her choice of y , which means she will need some square roots of v_i 's.

For example, suppose Peggy is able to supply the correct response when $b_1 = 1, b_2 = 1, b_4 = 1$, and all other $b_i = 0$. This could be accomplished by guessing the bits and using the preceding method of choosing x . However, suppose Victor sends $b_1 = 1, b_3 = 1$, and all other $b_i = 0$. Then Peggy will be ready to supply a square root of $xv_1^{-1}v_2^{-1}v_4^{-1}$ but will be asked to supply a square root of $xv_1^{-1}v_3^{-1}$. This, combined with what she knows, is equivalent to knowing a square root of $v_2^{-1}v_3v_4^{-1}$, which she is not able to compute. In an extreme case, Victor could send all bits equal to 0, which means Peggy must supply a square root of x . With Peggy's guess as before, this means she would know a square root of $v_1v_2v_4$. In summary, if Peggy's guess is not correct, she will need to know the square root of a nonempty product of v_i 's, which she cannot compute. Therefore, there are 2^k possible strings of bits that Victor can send, and only one will allow Peggy to fool Victor. In one iteration of the protocol, the chances are only one in 2^k that Victor will be fooled. If the procedure is repeated t times, the

chances are 1 in 2^{kt} that Victor is fooled. Recommended values are $k = 5$ and $t = 4$. Note that this gives the same probability as 20 iterations of the earlier scheme, so the present procedure is more efficient in terms of communication between Peggy and Victor. Of course, Victor has not obtained as strong a verification that Peggy knows, for example, s_1 , but he is very certain that Eve is not masquerading as Peggy, since Eve should not know any of the s_i 's.

There is an interesting feature of how the numbers are arranged in Steps 1 and 4. It is possible for Peggy to use a cryptographic hash function and send the hash of x after computing it in Step 1. After Victor computes $y^2 v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k} \pmod{n}$ in Step 4, he can compute the hash of this number and compare with the hash sent by Peggy. The hash function is assumed to be collision resistant, so Victor can be confident that the congruence in Step 4 is satisfied. Since the output of the hash function is probably shorter than x (for example, 256 bits for the hash, compared to 2048 bits for x), this saves a few bits of transmission.

The preceding can be used to design an identification scheme. Let I be a string that includes Peggy's name, birth date, and any other information deemed appropriate. Let H be a public hash function. A trusted authority Arthur (the bank, a passport agency, ...) chooses $n = pq$ to be the product of two large primes. Arthur computes $H(I \parallel j)$ for some small values of j , where $I \parallel j$ means j is appended to I . Using his knowledge of p, q , he can determine which of these numbers $H(I \parallel j)$ have square roots mod n and calculate a square root for each such number. This yields numbers $v_1 = H(I \parallel j_1), \dots, v_k = H(I \parallel j_k)$ and square roots s_1, \dots, s_k . The numbers I, n, j_1, \dots, j_k are made public. Arthur gives the numbers s_1, \dots, s_k to Peggy, who keeps them secret. The prime numbers p, q are discarded once the square

roots are calculated. Likewise, Arthur does not need to store s_1, \dots, s_k once they are given to Peggy. These two facts add to the security, since someone who breaks into Arthur's computer cannot compromise Peggy's security. Moreover, a different n can be used for each person, so it is hard to compromise the security of more than one individual at a time.

Note that since approximately half the numbers mod p and half the numbers mod q have square roots, the Chinese remainder theorem implies that approximately 1/4 of the numbers mod n have square roots. Therefore, each $H(I \parallel j)$ has a 1/4 probability of having a square root mod n . This means that Arthur should be able to produce the necessary numbers j_1, \dots, j_k quickly.

Peggy goes to an automatic teller machine, for example. The machine reads I from Peggy's card. It downloads n, j_1, \dots, j_k from a database and calculates $v_i = H(I \parallel j_i)$ for $1 \leq i \leq k$. It then performs the preceding procedure to verify that Peggy knows s_1, \dots, s_k . After a few iterations, the machine is convinced that the person is Peggy and allows her to withdraw cash. A naive implementation would require a lot of typing on Peggy's part, but at least Eve won't get Peggy's secret numbers. A better implementation would use chips embedded in the card and store some information in such a way that it cannot be extracted.

If Eve obtains the communications used in the transaction, she cannot determine Peggy's secret numbers. In fact, because of the zero-knowledge nature of the protocol, Eve obtains no information on the secret numbers s_1, \dots, s_k that can be reused in future transactions.

19.3 Exercises

1. Consider the diagram of tunnels in Figure 19.2. Suppose each of the four doors to the central chamber is locked so that a key is needed to enter, but no key is needed to exit. Peggy claims she has the key to one of the doors. Devise a zero-knowledge protocol in which Peggy proves to Victor that she can enter the central chamber. Victor should obtain no knowledge of which door Peggy can unlock.

Figure 19.2 Diagram for
Exercise 1

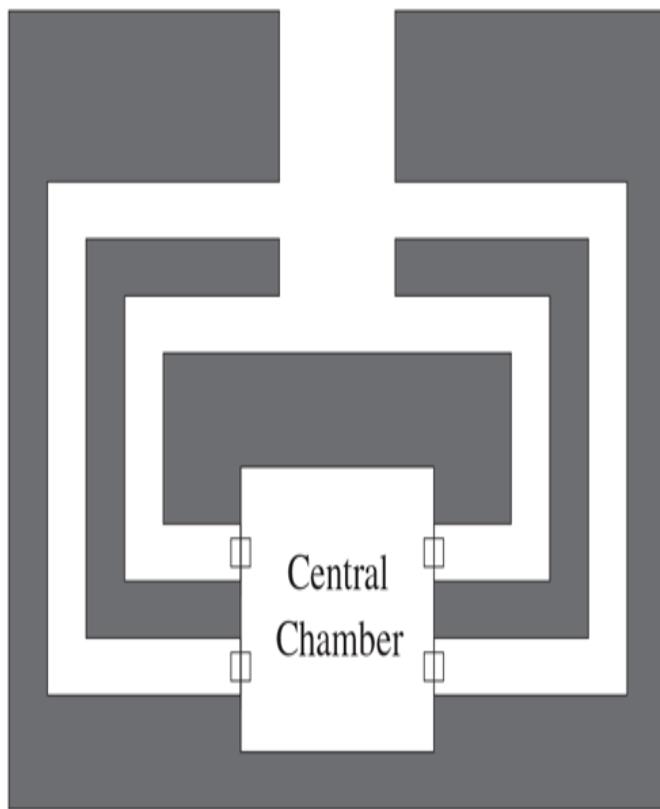


Figure 19.2 Full Alternative Text

2. Suppose p is a large prime, α is a primitive root, and $\beta \equiv \alpha^a \pmod{p}$. The numbers p, α, β are public. Peggy wants

to prove to Victor that she knows a without revealing it. They do the following:

1. Peggy chooses a random number $r \pmod{p-1}$.
2. Peggy computes $h_1 \equiv \alpha^r \pmod{p}$ and $h_2 \equiv \alpha^{a-r} \pmod{p}$ and sends h_1, h_2 to Victor.
3. Victor chooses $i = 1$ or $i = 2$ and asks Peggy to send either $r_1 = r$ or $r_2 = a - r \pmod{p-1}$.
4. Victor checks that $h_1 h_2 \equiv \beta \pmod{p}$ and that $h_i \equiv \alpha^{r_i} \pmod{p}$.

They repeat this procedure t times, for some specified t .

1. Suppose Peggy does not know a . Why will she usually be unable to produce numbers that convince Victor?
2. If Peggy does not know a , what is the probability that Peggy can convince Victor that she knows a ?
3. Suppose naive Nelson tries a variant. He wants to convince Victor that he knows a , so he chooses a random r as before, but does not send h_1, h_2 . Victor asks for r_i and Nelson sends it. They do this several times. Why is Victor not convinced of anything? What is the essential difference between Nelson's scheme and Peggy's scheme that causes this?
3. Naive Nelson thinks he understands zero-knowledge protocols. He wants to prove to Victor that he knows the factorization of n (which equals pq for two large primes p and q) without revealing this factorization to Victor or anyone else. Nelson devises the following procedure: Victor chooses a random integer $x \pmod{n}$, computes $y \equiv x^2 \pmod{n}$, and sends y to Nelson. Nelson computes a square root s of $y \pmod{n}$ and sends s to Victor. Victor checks that $s^2 \equiv y \pmod{n}$. Victor repeats this 20 times.
 1. Describe how Nelson computes s . You may assume that p and q are $\equiv 3 \pmod{4}$ (see [Section 3.9](#)).
 2. Explain how Victor can use this procedure to have a high probability of finding the factorization of n . (Therefore, this is not a zero-knowledge protocol.)
 3. Suppose Eve is eavesdropping and hears the values of each y and s . Is it likely that Eve obtains any useful information? (Assume no value of y repeats.)
4. [Exercise 2](#) gave a zero-knowledge proof that Peggy knows a discrete logarithm. Here is another method. Suppose p is a large prime, α is a primitive root, and $\beta \equiv \alpha^a \pmod{p}$. The numbers

p, α, β are public. Peggy wants to prove to Victor that she knows a without revealing it. They do the following:

1. Peggy chooses a random integer k with $1 \leq k < p - 1$, computes $\gamma \equiv \alpha^k \pmod{p}$, and sends γ to Victor.
 2. Victor chooses a random integer r with $1 \leq r < p - 1$ and sends r to Peggy.
 3. Peggy computes $y \equiv k - ar \pmod{p-1}$ and sends y to Victor.
 4. Victor checks whether $\gamma \equiv \alpha^y \beta^r \pmod{p}$. If so, he believes that Peggy knows a .
1. Show that the verification equation holds if the procedure is followed correctly.
 2. Does Victor obtain any information that will allow him to compute a ?
 3. Suppose Eve finds out the values of γ, r , and y . Will she be able to determine a ?
 4. Suppose Peggy repeats the procedure with the same value of k , but Victor uses different values r_1 and r_2 . How can Eve, who has listened to all communications between Victor and Peggy, determine a ?

The preceding procedure is the basis for the **Schnorr identification scheme**. Victor could be a bank and a could be Peggy's personal identification number. The bank stores β , and Peggy must prove she knows a to access her account. Alternatively, Victor could be a central computer and Peggy could be logging on to the computer through nonsecure telephone lines. Peggy's password is a , and the central computer stores β .

In the Schnorr scheme, p is usually chosen so that $p - 1$ has a large prime factor q , and α , instead of being a primitive root, is taken to satisfy $\alpha^q \equiv 1 \pmod{p}$. The congruence defining y is then taken mod q . Moreover, r is taken to satisfy $1 \leq r \leq 2^t$ for some t , for example, $t = 40$.

5. Peggy claims that she knows an RSA plaintext. That is, n, e, c are public and Peggy claims that she knows m such that $m^e \equiv c \pmod{n}$. She wants to prove this to Victor using a zero-knowledge protocol. Peggy and Victor perform the following steps:

1. Peggy chooses a random integer r_1 and computes $r_2 \equiv m \cdot r_1^{-1} \pmod{n}$ (assume that $\gcd(r_1, n) = 1$.)

2. Peggy computes $x_1 \equiv r_1^e \pmod{n}$ and
 $x_2 \equiv r_2^e \pmod{n}$ and sends x_1, x_2 to Victor.

3. Victor checks that $x_1x_2 \equiv c \pmod{n}$.

Give the remaining steps of the protocol. Victor should be at least 99% convinced that Peggy is not lying.

6. 1. Suppose that p is a large prime, and
 $g, h \not\equiv 0 \pmod{p}$. Peggy wants to prove to Victor, using a zero-knowledge protocol, that she knows a value of x with $g^x \equiv h \pmod{p}$. Peggy and Victor do the following:

1. Peggy chooses three random integers
 r_1, r_2, r_3 with
 $r_1 + r_2 + r_3 \equiv x \pmod{p-1}$.

2. Peggy computes $h_i \equiv g^{r_i}$, for $i = 1, 2, 3$ and sends h_1, h_2, h_3 to Victor.

3. Victor checks that $h_1h_2h_3 \equiv h \pmod{n}$.

Design the remaining steps of this protocol so that Victor is at least 99% convinced that Peggy is not lying. (Note: There are two ways for Victor to proceed in Step 4. One has a higher probability of catching Peggy, if she is cheating, than the other.)

2. Give a reasonable method for Peggy to choose the three random numbers such that

$r_1 + r_2 + r_3 \equiv x \pmod{p-1}$. (A method that doesn't work is "Choose three random numbers and see if their sum is x . If not, try again.")

7. Suppose that n is the product of two large primes, and that s is given. Peggy wants to prove to Victor, using a zero-knowledge protocol, that she knows a value of x with $x^2 \equiv s \pmod{n}$. Peggy and Victor do the following:

1. Peggy chooses three random integers r_1, r_2, r_3 with
 $r_1r_2r_3 \equiv x \pmod{n}$.

2. Peggy computes $x_i \equiv r_i^2$, for $i = 1, 2, 3$ and sends x_1, x_2, x_3 to Victor.

3. Victor checks that $x_1x_2x_3 \equiv s \pmod{n}$.

1. Design the remaining steps of this protocol so that Victor is at least 99% convinced that Peggy is not lying. (Note: There are two ways for Victor to proceed in Step 4. One

has a higher probability of catching Peggy, if she is cheating, than the other.)

2. Give a reasonable method for Peggy to choose the three random numbers such that $r_1 r_2 r_3 \equiv x \pmod{n}$. (A method that doesn't work is "Choose three random numbers and see if their product is x . If not, try again.")
8. Peggy claims that she knows an RSA plaintext. That is, n, e, c are public and Peggy claims that she knows m such that $m^e \equiv c \pmod{n}$. Devise a zero-knowledge protocol similar to that used in [Exercises 6 and 7](#) for Peggy to convince Victor that she knows m .

Chapter 20 Information Theory

In this chapter we introduce the theoretical concepts behind the security of a cryptosystem. The basic question is the following: If Eve observes a piece of ciphertext, does she gain any new information about the encryption key that she did not already have? To address this issue, we need a mathematical definition of information. This involves probability and the use of a very important measure called entropy.

Many of the ideas in this chapter originated with Claude Shannon in the 1940s.

Before we start, let's consider an example. Roll a standard six-sided die. Let A be the event that the number of dots is odd, and let B be the event that the number of dots is at least 3. If someone tells you that the roll belongs to the event $A \cap B$, then you know that there are only two possibilities for what the roll is. In this sense, $A \cap B$ tells you more about the value of the roll than just the event A , or just the event B . In this sense, the information contained in the event $A \cap B$ is larger than the information just in A or just in B .

The idea of information is closely linked with the idea of uncertainty. Going back to the example of the die, if you are told that the event $A \cap B$ happened, you become less uncertain about what the value of the roll was than if you are simply told that event A occurred. Thus the information increased while the uncertainty decreased. Entropy provides a measure of the increase in information or the decrease in uncertainty provided by the outcome of an experiment.

20.1 Probability Review

In this section we briefly introduce the concepts from probability needed for what follows. An understanding of probability and the various identities that arise is essential for the development of entropy.

Consider an experiment X with possible outcomes in a finite set X . For example, X could be flipping a coin and $X = \{\text{heads, tails}\}$. We assume each outcome is assigned a probability. In the present example, $p(X = \text{heads}) = 1/2$ and $p(X = \text{tails}) = 1/2$. Often, the outcome X of an experiment is called a **random variable**.

In general, for each $x \in X$, denote the probability that $X = x$ by

$$p_X(x) = p_x = p(X = x).$$

Note that $\sum_{x \in X} p_x = 1$. If $A \subseteq X$, let

$$p(A) = \sum_{x \in A} p_x,$$

which is the probability that X takes a value in A .

Often one performs an experiment where one is measuring several different events. These events may or may not be related, but they may be lumped together to form a new random event. For example, if we have two random events X and Y with possible outcomes X and Y , respectively, then we may create a new random event $Z = (X, Y)$ that groups the two events together. In this case, the new event Z has a set of possible outcomes $Z = X \times Y$, and Z is sometimes called a joint random variable.

Example

Draw a card from a standard deck. Let X be the suit of the card, so

$X = \{\text{clubs, diamonds, hearts, spades}\}$. Let Y be the value of the card, so $Y = \{\text{two, three, \dots, ace}\}$. Then Z gives the 52 possibilities for the card. Note that if $x \in X$ and $y \in Y$, then

$p((X, Y) = (x, y)) = p(X = x, Y = y)$ is simply the probability that the card drawn has suit x and value y . Since all cards are equally probable, this probability is $1/52$, which is the probability that $X = x$ (namely $1/4$) times the probability that $Y = y$ (namely $1/13$). As we discuss later, this means X and Y are independent.

Example

Roll a die. Suppose we are interested in two things: whether the number of dots is odd and whether the number is at least 2. Let $X = 0$ if the number of dots is even and $X = 1$ if the number of dots is odd. Let $Y = 0$ if the number of dots is less than 2 and $Y = 1$ if the number of dots is at least 2. Then $Z = (X, Y)$ gives us the results of both experiments together. Note that the probability that the number of dots is odd and less than 2 is $p(Z = (1, 0)) = 1/6$. This is not equal to $p(X = 0) \cdot p(Y = 0)$, which is $(1/2)(1/6) = 1/12$. This means that X and Y are not independent. As we'll see, this is closely related to the fact that knowing X gives us information about Y .

We denote

$$p_{X, Y}(x, y) = p(X = x, Y = y).$$

Note that we can recover the probability that $X = x$ as

$$p_X(x) = \sum_{y \in Y} p_{X, Y}(x, y).$$

We say that two random events X and Y are **independent** if

$$p_{X,Y}(x,y) = p_X(x)p_Y(y)$$

for all $x \in X$ and all $y \in Y$. In the preceding example, the suit of a card and the value of the card were independent.

We are also interested in the probabilities for Y given that $X = x$ has occurred. If $p_X(x) > 0$, define the **conditional probability** of $Y = y$ given that $X = x$ to be

$$p_Y(y|x) = \frac{p_{X,Y}(x,y)}{p_X(x)}.$$

One way to think of this is that we have restricted to the set where $X = x$. This has total probability $p_X(x) = \sum_y p_{X,Y}(x,y)$. The fraction of this sum that comes from $Y = y$ is $p_Y(y|x)$.

Note that X and Y are independent if and only if

$$p_Y(y|x) = p_Y(y)$$

for all x, y . In other words, the probability of y is unaffected by what happens with X .

There is a nice way to go from the conditional probability of Y given X to the conditional probability of X given Y

.

Bayes's Theorem

If $p_X(x) > 0$ and $p_Y(y) > 0$, then

$$p_X(x|y) = \frac{p_X(x)p_Y(y|x)}{p_Y(y)}.$$

The proof consists of simply writing the conditional probabilities in terms of their definitions.

20.2 Entropy

Roll a six-sided die and a ten-sided die. Which experiment has more uncertainty? If you make a guess at the outcome of each roll, you are more likely to be wrong with the ten-sided die than with the six-sided die.

Therefore, the ten-sided die has more uncertainty.

Similarly, compare a fair coin toss in which heads and tails are equally likely with a coin toss in which heads occur 90% of the time. Which has more uncertainty? The fair coin toss does, again because there is more randomness in its possibilities.

In our definition of uncertainty, we want to make sure that two random variables X and Y that have same probability distribution have the same uncertainty. In order to do this, the measure of uncertainty must be a function only of the probability distributions and not of the names chosen for the outcomes.

We require the measure of uncertainty to satisfy the following properties:

1. To each set of nonnegative numbers p_1, \dots, p_n with $p_1 + \dots + p_n = 1$, the uncertainty is given by a number $H(p_1, \dots, p_n)$.
2. H should be a continuous function of the probability distribution, so a small change in the probability distribution should not drastically change the uncertainty.
3. $H\left(\frac{1}{n}, \dots, \frac{1}{n}\right) \leq H\left(\frac{1}{n+1}, \dots, \frac{1}{n+1}\right)$ for all $n > 0$. In other words, in situations where all outcomes are equally likely, the uncertainty increases when there are more possible outcomes.
4. If $0 < q < 1$, then

$$H(p_1, \dots, qp_j, (1-q)p_j, \dots, p_n) = H(p_1, \dots, p_j, \dots, p_n) + p_j H(q, 1-q).$$

What this means is that if the j th outcome is broken into two suboutcomes, with probabilities qp_j and $(1 - q)p_j$, then the total uncertainty is increased by the uncertainty caused by the choice between the two suboutcomes, multiplied by the probability p_j that we are in this case to begin with. For example, if we roll a six-sided die, we can record two outcomes: *even* and *odd*. This has uncertainty $H\left(\frac{1}{2}, \frac{1}{2}\right)$. Now suppose we break the outcome *even* into the suboutcomes 2 and $\{4, 6\}$. Then we have three possible outcomes: 2, $\{4, 6\}$, and *odd*. We have

$$H\left(\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2}H\left(\frac{2}{3}, \frac{1}{3}\right).$$

The first term is the uncertainty caused by *even* versus *odd*. The second term is the uncertainty added by splitting *even* into two suboutcomes.

Starting from these basic assumptions, Shannon [Shannon2] showed the following:

Theorem

Let $H(X)$ be a function satisfying properties (1)–(4). In other words, for each random variable X with outcomes $X = \{x_1, \dots, x_n\}$ having probabilities p_1, \dots, p_n , the function H assigns a number $H(X)$ subject to the conditions (1)–(4). Then H must be of the form

$$H(p_1, \dots, p_n) = -\lambda \sum_k p_k \log_2 p_k$$

where λ is a nonnegative constant and where the sum is taken over those k such that $p_k > 0$.

Because of the theorem, we define the **entropy** of the variable X to be

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x).$$

The entropy $H(X)$ is a measure of the uncertainty in the outcome of X . Note that since $\log_2 p(x) \leq 0$, we have $H(X) \geq 0$, so there is no such thing as negative uncertainty.

The observant reader might notice that there are problems when we have elements $x \in X$ that have probability 0. In this case we define $0 \log_2 0 = 0$, which is justified by looking at the limit of $x \log_2 x$ as $x \rightarrow 0$. It is typical convention that the logarithm is taken base 2, in which case entropy is measured in bits. The entropy of X may also be interpreted as the expected value of $-\log_2 p(X)$ (recall that $E[g(X)] = \sum_x g(x)p(x)$).

We now look at some examples.

Example

Consider a fair coin toss. There are two outcomes, each with probability $1/2$. The entropy of this random event is

$$-\left(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{2} \log_2 \frac{1}{2}\right) = 1.$$

This means that the result of the coin flip gives us 1 bit of information, or that the uncertainty in the outcome of the coin flip is 1 bit.

Example

Consider a nonfair coin toss X with probability p of getting heads and probability $1 - p$ of getting tails (where $0 < p < 1$). The entropy of this event is

$$H(X) = -p \log_2 p - (1 - p) \log_2 (1 - p).$$

If one considers $H(X)$ as a function of p , one sees that the entropy is a maximum when $p = \frac{1}{2}$. (For a more general statement, see [Exercise 14](#).)

Example

Consider an n -sided fair die. There are n outcomes, each with probability $1/n$. The entropy is

$$-\frac{1}{n} \log_2 (1/n) - \dots - \frac{1}{n} \log_2 (1/n) = \log_2 (n).$$

There is a relationship between entropy and the number of yes-no questions needed to determine accurately the outcome of a random event. If one considers a totally nonfair coin toss where $p(1) = 1$, then $H(X) = 0$.

This result can be interpreted as not requiring any questions to determine what the value of the event was. If someone rolls a four-sided die, then it takes two yes-no questions to find out the outcome. For example, is the number less than 3? Is the number odd?

A slightly more subtle example is obtained by flipping two coins. Let X be the number of heads, so the possible outcomes are $\{0, 1, 2\}$. The probabilities are $1/4, 1/2, 1/4$ and the entropy is

$$-\frac{1}{4} \log_2 (1/4) - \frac{1}{2} \log_2 (1/2) - \frac{1}{4} \log_2 (1/4) = \frac{3}{2}.$$

Note that we can average $3/2$ questions to determine the outcome. For example, the first question could be “Is there exactly one head?” Half of the time, this will suffice to determine the outcome. The other half of the time a second question is needed, for example, “Are there two heads?” So the average number of questions equals the entropy.

Another way of looking at $H(X)$ is that it measures the number of bits of information that we obtain when we are given the outcome of X . For example, suppose the outcome of X is a random 4-bit number, where each possibility has probability $1/16$. As computed previously, the entropy is $H(X) = 4$, which says we have received four bits of information when we are told the value of X .

In a similar vein, entropy relates to the minimal amount of bits necessary to represent an event on a computer

(which is a binary device). See [Section 20.3](#). There is no sense recording events whose outcomes can be predicted with 100% certainty; it would be a waste of space. In storing information, one wants to code just the uncertain parts because that is where the real information is.

If we have two random variables X and Y , the joint entropy $H(X, Y)$ is defined as

$$H(X, Y) = - \sum_{x \in X} \sum_{y \in Y} p_{X, Y}(x, y) \log_2 p_{X, Y}(x, y).$$

This is just the entropy of the joint random variable $Z = (X, Y)$ discussed in [Section 20.1](#).

In a cryptosystem, we might want to know the uncertainty in a key, given knowledge of the ciphertext. This leads us to the concept of **conditional entropy**, which is the amount of uncertainty in Y , given X . It is defined to be

$$\begin{aligned} H(Y|X) &= \sum_x p_X(x) H(Y|X=x) \\ &= - \sum_x p_X(x) \left(\sum_y p_Y(y|x) \log_2 p_Y(y|x) \right) \\ &= - \sum_x \sum_y p_{X, Y}(x, y) \log_2 p_Y(y|x). \end{aligned}$$

The last equality follows from the relationship $p_{X, Y}(x, y) = p_Y(y|x)p_X(x)$. The quantity $H(Y|X=x)$ is the uncertainty in Y given the information that $X = x$. It is defined in terms of conditional probabilities by the expression in parentheses on the second line. We calculate $H(Y|X)$ by forming a weighted sum of these uncertainties to get the total uncertainty in Y given that we know the value of X .

Remark

The preceding definition of conditional entropy uses the weighted average, over the various $x \in X$, of the entropy of Y given $X = x$. Note that

$H(Y|X) \neq -\sum_{x,y} p_Y(y|x) \log_2 (p_Y(y|x))$. This sum does not have properties that information or uncertainty should have. For example, if X and Y are independent, then this definition would imply that the uncertainty of Y given X is greater than the uncertainty of Y (see [Exercise 15](#)). This clearly should not be the case.

We now derive an important tool, the chain rule for entropies. It will be useful in [Section 20.4](#).

Theorem

(Chain Rule). $H(X, Y) = H(X) + H(Y|X)$.

Proof

$$\begin{aligned}
 H(X, Y) &= -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_{X,Y}(x, y) \\
 &= -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_X(x) p_Y(y|x) \\
 &= -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_X(x) - \sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_Y(y|x) \\
 &= \left(\sum_x \log_2 p_X(x) \sum_Y p_{X,Y}(x, y) \right) + H(Y|X) \\
 &= \sum_x p_X(x) \log_2 p_X(x) + H(Y|X) \quad (\text{since } \sum_y p_{X,Y}(x, y) = p_X(x)) \\
 &= H(X) + H(Y|X).
 \end{aligned}$$

What does the chain rule tell us? It says that the uncertainty of the joint event (X, Y) is equal to the uncertainty of event X + uncertainty of event Y given that event X has happened.

We now state three more results about entropy.

Theorem

1. $H(X) \leq \log_2 |X|$, where $|X|$ denotes the number of elements in X . We have equality if and only if all elements of X are equally likely.
2. $H(X, Y) \leq H(X) + H(Y)$.
3. (Conditioning reduces entropy) $H(Y|X) \leq H(Y)$, with equality if and only if X and Y are independent.

The first result states that you are most uncertain when the probability distribution is uniform. Referring back to the example of the nonfair coin flip, the entropy was maximum for $p = \frac{1}{2}$. This extends to events with more possible outcomes. For a proof of (1), see [Welsh, p. 5].

The second result says that the information contained in the pair (X, Y) is at most the information contained in X plus the information contained in Y . The reason for the inequality is that possibly the information supplied by X and Y overlap (which is when X and Y are not independent). For a proof of (2), see [Stinson].

The third result is one of the most important results in information theory. Its interpretation is very simple. It says that the uncertainty one has in a random event Y given that event X occurred is less than the uncertainty in event Y alone. That is, X can only tell you information about event Y ; it can't make you any more uncertain about Y .

The third result is an easy corollary of the second plus the chain rule:

$$H(X) + H(Y|X) = H(X, Y) \leq H(X) + H(Y).$$

20.3 Huffman Codes

Information theory originated in the late 1940s from the seminal papers by Claude Shannon. One of the primary motivations behind Shannon's mathematical theory of information was the problem of finding a more compact way of representing data. In short, he was concerned with the problem of compression. In this section we briefly touch on the relationship between entropy and compression and introduce Huffman codes as a method for more succinctly representing data.

For more on how to compress data, see [Cover-Thomas] or [Nelson-Gailly].

Example

Suppose we have an alphabet with four letters a, b, c, d , and suppose these letters appear in a text with frequencies as follows.

a	b	c	d
.5	.3	.1	.1

We could represent a as the binary string 00 , b as 01 , c as 10 , and d as 11 . This means that the message would average two bits per letter. However, suppose we represent a as 1 , b as 01 , c as 001 , and d as 000 . Then the average number of bits per letter is

$$(1)(.5) + (2)(.3) + (3)(.1) + (3)(.1) = 1.7$$

(the number of bits for a times the frequency of a , plus the number of bits for b times the frequency of b , etc.). This encoding of the letters is therefore more efficient.

In general, we have a random variable with outputs in a set X . We want to represent the outputs in binary in an efficient way; namely, the average number of bits per output should be as small as possible.

An early example of such a procedure is Morse code, which represents letters as sequences of dots and dashes and was developed to send messages by telegraph. Morse asked printers which letters were used most, and made the more frequent letters have smaller representations. For example, e is represented as \cdot and t as $-$. But x is $- \cdot \cdot -$ and z is $- - \cdot \cdot$.

A more recent method was developed by Huffman. The idea is to list all the outputs and their probabilities. The smallest two are assigned 1 and 0 and then combined to form an output with a larger probability. The same procedure is then applied to the new list, assigning 1 and 0 to the two smallest, then combining them to form a new list. This procedure is continued until there is only one output remaining. The binary strings are then obtained by reading backward through the procedure, recording the bits that have been assigned to a given output and to combinations containing it. This is best explained by an example.

Suppose we have outputs a, b, c, d with probabilities 0.5, 0.3, 0.1, 0.1, as in the preceding example. The diagram in Figure 20.1 gives the procedure.

Figure 20.1 An Example of Huffman Encoding

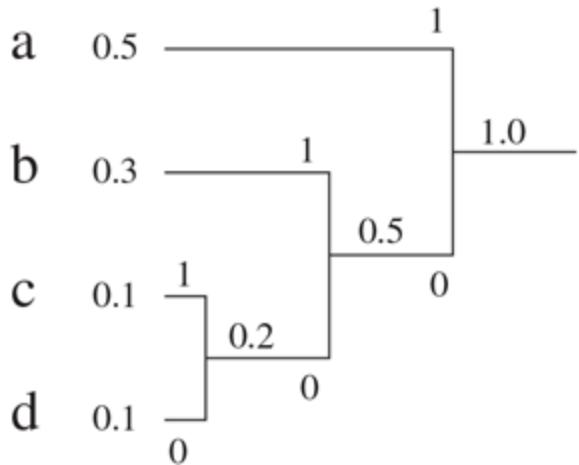


Figure 20.1 Full Alternative Text

Note that when there were two choices for the lowest, we made a random choice for which one received 0 and which one received 1. Tracing backward through the table, we see that *a* only received a 1, *b* received 01, *c* received 001, and *d* received 000. These are exactly the assignments made previously that gave a low number of bits per letter.

A useful feature of Huffman encoding is that it is possible to read a message one letter at a time. For example, the string 011000 can only be read as *bad*; moreover, as soon as we have read the first two bits 01, we know that the first letter is *b*.

Suppose instead that we wrote the bits assigned to letters in reverse order, so *b* is 10 and *c* is 001. Then the message 101000 cannot be determined until all bits have been read, since it potentially could start with *bb* or *ba*.

Even worse, suppose we had assigned 0 to *a* instead of 1. Then the messages *aaa* and *d* would be the same. It is possible to show that Huffman encoding avoids these two problems.

The average number of bits per output is closely related to the entropy.

Theorem

Let L be the average number of bits per output for Huffman encoding for the random variable X . Then

$$H(X) \leq L < H(X) + 1.$$

This result agrees with the interpretation that the entropy measures how many bits of information are contained in the output of X . We omit the proof. In our example, the entropy is

$$H(X) = -(.5 \log_2 (.5) + .3 \log_2 (.3) + .1 \log (.1) + .1 \log (.1)) \approx 1.685.$$

20.4 Perfect Secrecy

Intuitively, the one-time pad provides perfect secrecy. In [Section 4.4](#), we gave a mathematical meaning to this statement. In the present section, we repeat some of the arguments of that section and phrase some of the ideas in terms of entropy.

Suppose we have a cipher system with possible plaintexts P , ciphertexts C , and keys K . Each plaintext in P has a certain probability of occurring; some are more likely than others. The choice of a key in K is always assumed to be independent of the choice of plaintext. The possible ciphertexts in C have various probabilities, depending on the probabilities for P and K .

If Eve intercepts a ciphertext, how much information does she obtain for the key? In other words, what is $H(K|C)$? Initially, the uncertainty in the key was $H(K)$. Has the knowledge of the ciphertext decreased the uncertainty?

Example

Suppose we have three possible plaintexts: a , b , c with probabilities .5, .3, .2 and two keys k_1 , k_2 with probabilities .5 and .5. Suppose the possible ciphertexts are U , V , W . Let e_k be the encryption function for the key k . Suppose

$$\begin{aligned} e_{k_1}(a) &= U, \quad e_{k_1}(b) = V, \quad e_{k_1}(c) = W \\ e_{k_2}(a) &= U, \quad e_{k_2}(b) = W, \quad e_{k_2}(c) = V. \end{aligned}$$

Let $p_P(a)$ denote the probability that the plaintext is a , etc. The probability that the ciphertext is U is

$$\begin{aligned} p_C(U) &= p_K(k_1)p_P(a) + p_K(k_2)p_P(a) \\ &= (.5)(.5) + (.5)(.5) = .50. \end{aligned}$$

Similarly, we calculate $p_C(V) = .25$ and $p_C(W) = .25$.

Suppose someone intercepts a ciphertext. This gives some information on the plaintext. For example, if the ciphertext is U , then it can be deduced immediately that the plaintext was a . If the ciphertext is V , the plaintext was either b or c .

We can even say more: The probability that a ciphertext is V is $.25$, so the conditional probability that the plaintext was b , given that the ciphertext is V is

$$p(b|V) = \frac{p_{(P, C)}(b, V)}{p_C(V)} = \frac{p_{(P, K)}(b, k_1)}{p_C(V)} = \frac{(.3)(.5)}{.25} = .6.$$

Similarly, $p(c|V) = .4$ and $p(a|V) = 0$. We can also calculate

$$p(a|W) = 0, \quad p(b|W) = .6, \quad p(c|W) = .4.$$

Note that the original probabilities of the plaintexts were $.5$, $.3$, and $.2$; knowledge of the ciphertext allows us to revise the probabilities. Therefore, the ciphertext gives us information about the plaintext. We can quantify this via the concept of conditional entropy. First, the entropy of the plaintext is

$$H(P) = -(.5 \log_2 (.5) + .3 \log_2 (.3) + .2 \log_2 (.2)) = 1.485.$$

The conditional entropy of P given C is

$$H(P|C) = - \sum_{x \in \{a, b, c\}} \sum_{Y \in \{U, V, W\}} p(Y)p(x|Y) \log_2 (p(x|Y)) = .485.$$

Therefore, in the present example, the uncertainty for the plaintext decreases when the ciphertext is known.

On the other hand, we suspect that for the one-time pad the ciphertext yields no information about the plaintext that was not known before. In other words, the

uncertainty for the plaintext should equal the uncertainty for the plaintext given the ciphertext. This leads us to the following definition and theorem.

Definition

A cryptosystem has **perfect secrecy** if
 $H(P|C) = H(P)$.

Theorem

The one-time pad has perfect secrecy.

Proof. Recall that the basic setup is the following: There is an alphabet with Z letters (for example, Z could be 2 or 26). The possible plaintexts consist of strings of characters of length L . The ciphertexts are strings of characters of length L . There are Z^L keys, each consisting of a sequence of length L denoting the various shifts to be used. The keys are chosen randomly, so each occurs with probability $1/Z^L$.

Let $c \in C$ be a possible ciphertext. As before, we calculate the probability that c occurs:

$$p_C(c) = \sum_{\substack{x \in P \\ k \in K \\ e_k(x) = c}} p_P(x)p_K(k).$$

Here $e_k(x)$ denotes the ciphertext obtained by encrypting x using the key k . The sum is over those pairs x, k such that k encrypts x to c . Note that we have used the independence of P and K to write joint probability $p_{(P, K)}(x, k)$ as the product of the individual probabilities.

In the one-time pad, every key has equal probability $1/Z^L$, so we can replace $p_K(k)$ in the above sum by $1/Z^L$. We obtain

$$p_C(c) = \frac{1}{Z^L} \sum_{\substack{x \in P \\ k \in K \\ e_k(x) = c}} p_P(x).$$

We now use another important feature of the one-time pad: For each plaintext x and each ciphertext c , there is exactly one key k such that $e_k(x) = c$. Therefore, every $x \in P$ occurs exactly once in the preceding sum, so we have $Z^{-L} \sum_{x \in P} p_P(x)$. But the sum of the probabilities of all possible plaintexts is 1, so we obtain

$$p_C(c) = \frac{1}{Z^L}.$$

This confirms what we already suspected: Every ciphertext occurs with equal probability.

Now let's calculate some entropies. Since K and C each have equal probabilities for all Z^L possibilities, we have

$$H(K) = H(C) = \log_2 (Z^L).$$

We now calculate $H(P, K, C)$ in two different ways. Since knowing (P, K, C) is the same as knowing (P, K) , we have

$$H(P, K, C) = H(P, K) = H(P) + H(K).$$

The last equality is because P and K are independent. Also, knowing (P, K, C) is the same as knowing (P, C) since C and P determine K for the one-time pad. Therefore,

$$H(P, K, C) = H(P, C) = H(P|C) + H(C).$$

The last equality is the chain rule. Equating the two expressions, and using the fact that $H(K) = H(C)$, we obtain $H(P|C) = H(P)$. This proves that the one-time pad has perfect secrecy.

The preceding proof yields the following more general result. Let $\#K$ denote the number of possible keys, etc.

Theorem

Consider a cryptosystem such that

1. Every key has probability $1/\#K$.
2. For each $x \in P$ and $c \in C$ there is exactly one $k \in K$ such that $e_k(x) = c$.

Then this cryptosystem has perfect secrecy.

It is easy to deduce from condition (2) that $\#C = \#K$. Conversely, it can be shown that if $\#P = \#C = \#K$ and the system has perfect secrecy, then (1) and (2) hold (see [Stinson, Theorem 2.4]).

It is natural to ask how the preceding concepts apply to RSA. The possibly surprising answer is that

$H(P|C) = 0$; namely, the ciphertext determines the plaintext. The reason is that entropy does not take into account computation time. The fact that it might take billions of years to factor n is irrelevant. What counts is that all the information needed to recover the plaintext is contained in the knowledge of n , e , and c .

The more relevant concept for RSA is the computational complexity of breaking the system.

20.5 The Entropy of English

In an English text, how much information is obtained per letter? If we had a random sequence of letters, each appearing with probability $1/26$, then the entropy would be $\log_2 (26) = 4.70$; so each letter would contain 4.7 bits of information. If we include spaces, we get $\log_2 (27) = 4.75$. But the letters are not equally likely: a has frequency .082, b has frequency .015, etc. (see [Section 2.3](#)). Therefore, we consider

$$-(.082 \log_2 .082 + .015 \log_2 .015 + \dots) = 4.18.$$

However, this doesn't tell the whole story. Suppose we have the sequence of letters *we are studyin*. There is very little uncertainty as to what the last letter is; it is easy to guess that it is *g*. Similarly, if we see the letter *q*, it is extremely likely that the next letter is *u*. Therefore, the existing letters often give information about the next letter, which means that there is not as much additional information carried by that letter. This says that the entropy calculated previously is still too high. If we use tables of the frequencies of the $26^2 = 676$ digrams (a digram is a two-letter combination), we can calculate the conditional entropy of one letter, given the preceding letter, to be 3.56. Using trigram frequencies, we find that the conditional entropy of a letter, given the preceding two letters, is approximately 3.3. This means that, on the average, if we know two consecutive letters in a text, the following letter carries 3.3 bits of additional information. Therefore, if we have a long text, we should expect to be able to compress it at least by a factor of around $3.3/4.7 = .7$.

Let L represent the letters of English. Let L^N denote the N -gram combinations. Define the entropy of English to be

$$H_{\text{English}} = \lim_{N \rightarrow \infty} \frac{H(L^N)}{N},$$

where $H(L^N)$ denotes the entropy of N -grams. This gives the average amount of information per letter in a long text, and it also represents the average amount of uncertainty in guessing the next letter, if we already know a lot of the text. If the letters were all independent of each other, so the probability of the digram qu equaled the probability of q times the probability of u , then we would have $H(L^N) = N \cdot H(L)$, and the limit would be $H(L)$, which is the entropy for one-letter frequencies. But the interactions of letters, as noticed in the frequencies for digrams and trigrams, lower the value of $H(L^N)$.

How do we compute $H(L^N)$? Calculating 100-gram frequencies is impossible. Even tabulating the most common of them and getting an approximation would be difficult. Shannon proposed the following idea.

Suppose we have a machine that is an optimal predictor, in the sense that, given a long string of text, it can calculate the probabilities for the letter that will occur next. It then guesses the letter with highest probability. If correct, it notes the letter and writes down a 1. If incorrect, it guesses the second most likely letter. If correct, it writes down a 2, etc. In this way, we obtain a sequence of numbers. For example, consider the text *itissunnytoday*. Suppose the predictor says that t is the most likely for the 1st letter, and it is wrong; its second guess is i , which is correct, so we write the i and put 2 below it. The predictor then predicts that t is the next letter, which is correct. We put 1 beneath the t . Continuing, suppose it finds i on its 1st guess, etc. We obtain a situation like the following:

<i>i</i>	<i>t</i>	<i>i</i>	<i>s</i>	<i>s</i>	<i>u</i>	<i>n</i>	<i>n</i>	<i>y</i>	<i>t</i>	<i>o</i>	<i>d</i>	<i>a</i>	<i>y</i>
2	1	1	1	4	3	2	1	4	1	1	1	1	1

Using the prediction machine, we can reconstruct the text. The prediction machine says that its second guess for the first letter will be i , so we know the 1st letter is i . The predictor says that its first guess for the next letter is t , so we know that's next. The first guess for the next is i , etc.

What this means is that if we have a machine for predicting, we can change a text into a string of numbers without losing any information, because we can reconstruct the text. Of course, we could attempt to write a computer program to do the predicting, but Shannon suggested that the best predictor is a person who speaks English. Of course, a person is unlikely to be as deterministic as a machine, and repeating the experiment (assuming the person forgets the text from the first time) might not yield an identical result. So reconstructing the text might present a slight difficulty. But it is still a reasonable assumption that a person approximates an optimal predictor.

Given a sequence of integers corresponding to a text, we can count the frequency of each number. Let

$$q_i = \text{frequency of the number } i.$$

Since the text and the sequence of numbers can be reconstructed from each other, their entropies must be the same. The largest the entropy can be for the sequence of numbers is when these numbers are independent. In this case, the entropy is $-\sum_{i=1}^{26} q_i \log_2 (q_i)$. However, the numbers are probably not independent. For example, if there are a couple consecutive 1s, then perhaps the predictor has guessed the rest of the word, which means that there will be a few more 1s. However, we get an upper bound for the entropy, which is usually better than the one we obtain using frequencies of letters. Moreover, Shannon also found a lower bound for the entropy. His results are

$$\sum_{i=1}^{26} i(q_i - q_{i+1}) \log_2 (i) \leq H_{\text{English}} \leq - \sum_{i=1}^{26} q_i \log_2 (q_i).$$

Actually, these are only approximate upper and lower bounds, since there is experimental error, and we are really considering a limit as $N \rightarrow \infty$.

These results allow an experimental estimation of the entropy of English. Alice chooses a text and Bob guesses the first letter, continuing until the correct guess is made. Alice records the number of guesses. Bob then tries to guess the second letter, and the number of guesses is again recorded. Continuing in this way, Bob tries to guess each letter. When he is correct, Alice tells him and records the number of guesses. Shannon gave [Table 20.1](#) as a typical result of an experiment. Note that he included spaces, but ignored punctuation, so he had 27 possibilities: There are 102 symbols. There are seventy-nine 1s, eight 2s, three 3s, etc. This gives

$$q_1 = 79/102, \quad q_2 = 8/102, \quad q_3 = 3/102, \quad q_4 = q_5 = 2/102, \\ q_6 = 3/102, \quad q_7 = q_8 = q_{11} = q_{15} = q_{17} = 1/102.$$

Table 20.1 Shannon's Experiment on the Entropy of English

t h e r e i s n o r e v e r s e

1 1 1 5 1 1 2 1 1 2 1 1 15 1 17 1 1 1 2 1

o n a m o t o r c y c l e a

3 2 1 2 2 7 1 1 1 1 4 1 1 1 1 1 3 1

f r i e n d o f m i n e f o u n d

8 6 1 3 1 1 1 1 1 1 1 1 1 1 6 2 1 1 1 1

t h i s o u t r a t h e r

1 1 2 1 1 1 1 1 1 4 1 1 1 1 1 1

d r a m a t i c a l l y t h e

11 5 1 1 1 1 1 1 1 1 1 1 1 6 1 1 1

o t h e r d a y

1 1 1 1 1 1 1 1 1 1

Table 20.1 Full Alternative Text

The upper bound for the entropy is therefore

$$-\left(\frac{79}{102} \log_2 \frac{79}{102} + \cdots + \frac{1}{102} \log_2 \frac{1}{102}\right) \approx 1.42.$$

Note that since we are using $0 \log_2 0 = 0$, the terms with $q_i = 0$ can be omitted. The lower bound is

$$1 \cdot \left(\frac{79}{102} - \frac{8}{102} \right) \log_2 (1) + 2 \cdot \left(\frac{8}{102} - \frac{3}{102} \right) \log_2 (2) + \dots \approx .72.$$

A reasonable estimate is therefore that the entropy of English is near 1, maybe slightly more than 1.

If we want to send a long English text, we could write each letter (and the space) as a string of five bits. This would mean that a text of length 102, such as the preceding, would require 510 bits. It would be necessary to use something like this method if the letters were independent and equally likely. However, suppose we do a Huffman encoding of the message
1, 1, 1, 5, 1, 1, 2, ... from Table 20.1. Let

$$\begin{aligned} 1 &\leftrightarrow 1 & 2 &\leftrightarrow 110 & 3 &\leftrightarrow 1010 & 4 &\leftrightarrow 0100 \\ 5 &\leftrightarrow 11100 & 6 &\leftrightarrow 0010 & 7 &\leftrightarrow 01100 & 8 &\leftrightarrow 11000 \\ 11 &\leftrightarrow 01000 & 15 &\leftrightarrow 10000 & 17 &\leftrightarrow 100000. \end{aligned}$$

All other numbers up to 27 can be represented by various combinations of six or more bits. To send the message requires

$$79 \cdot 1 + 8 \cdot 3 + 3 \cdot 4 + 2 \cdot 4 + \dots + 1 \cdot 6 = 171 \text{ bits},$$

which is 1.68 bits per letter.

Note that five bits per letter is only slightly more than the “random” entropy 4.75, and 1.68 bits per letter is slightly more than our estimate of the entropy of English. These agree with the result that entropy differs from the average length of a Huffman encoding by at most 1.

One way to look at the preceding entropy calculations is to say that English is around 75% redundant. Namely, if we send a long message in standard written English, compared to the optimally compressed text, the ratio is approximately 4 to 1 (that is, the random entropy 4.75 divided by the entropy of English, which is around 1). In our example, we were close, obtaining a ratio near 3 to 1 (namely 4.75/1.68).

Define the **redundancy** of English to be

$$R = 1 - \frac{H_{\text{English}}}{\log_2 (26)}.$$

Then R is approximately 0.75, which is the 75% redundancy mentioned previously.

20.5.1 Unicity Distance

Suppose we have a ciphertext. How many keys will decrypt it to something meaningful? If the text is long enough, we suspect that there is a unique key and a unique corresponding plaintext. The unicity distance n_0 for a cryptosystem is the length of ciphertext at which one expects that there is a unique meaningful plaintext. A rough estimate for the unicity distance is

$$n_0 = \frac{\log_2 |K|}{R \log_2 |L|},$$

where $|K|$ is the number of possible keys, $|L|$ is the number of letters or symbols, and R is the redundancy (see [Stinson]). We'll take $R = .75$ (whether we include spaces in our language or not; the difference is small).

For example, consider the substitution cipher, which has $26!$ keys. We have

$$n_0 = \frac{\log_2 26!}{.75 \log_2 26} \approx 25.1.$$

This means that if a ciphertext has length 25 or more, we expect that usually there is only one possible meaningful plaintext. Of course, if we have a ciphertext of length 25, there are probably several letters that have not appeared. Therefore, there could be several possible keys, all of which decrypt the ciphertext to the same plaintext.

As another example, consider the affine cipher. There are 312 keys, so

$$n_0 = \frac{\log_2 312}{.75 \log_2 26} \approx 2.35.$$

This should be regarded as only a very rough approximation. Clearly it should take a few more letters to get a unique decryption. But the estimate of 2.35 indicates that very few letters suffice to yield a unique decryption in most cases for the affine cipher.

Finally, consider the one-time pad for a message of length N . The encryption is a separate shift mod 26 for each letter, so there are 26^N keys. We obtain the estimate

$$n_0 \approx \frac{\log_2 26^N}{.75 \log_2 26} = 1.33N.$$

In this case, it says we need more letters than the entire ciphertext to get a unique decryption. This reflects the fact that all plaintexts are possible for any ciphertext.

20.6 Exercises

1. Let X_1 and X_2 be two independent tosses of a fair coin. Find the entropy $H(X_1)$ and the joint entropy $H(X_1, X_2)$. Why is $H(X_1, X_2) = H(X_1) + H(X_2)$?

2. Consider an unfair coin where the two outcomes, heads and tails, have probabilities $p(\text{heads}) = p$ and $p(\text{tails}) = 1 - p$.

1. If the coin is flipped two times, what are the possible outcomes along with their respective probabilities?

2. Show that the entropy in part (a) is $-2p \log_2(p) - 2(1-p) \log_2(1-p)$. How could this have been predicted without calculating the probabilities in part (a)?

3. A random variable X takes the values $1, 2, \dots, n, \dots$ with probabilities $\frac{1}{2}, \frac{1}{2^2}, \dots, \frac{1}{2^n}, \dots$. Calculate the entropy $H(X)$.

4. Let X be a random variable taking on integer values. The probability is $1/2$ that X is in the range $[0, 2^8 - 1]$, with all such values being equally likely, and the probability is $1/2$ that the value is in the range $[2^8, 2^{32} - 1]$, with all such values being equally likely. Compute $H(X)$.

5. Let X be a random event taking on the values $-2, -1, 0, 1, 2$, all with positive probability. What is the general inequality/equality between $H(X)$ and $H(Y)$, where Y is the following?

1. $Y = 2^X$

2. $Y = X^2$

6. 1. In this problem we explore the relationship between the entropy of a random variable X and the entropy of a function $f(X)$ of the random variable. The following is a short proof that shows $H(f(X)) \leq H(X)$. Explain what principles are used in each of the steps.

$$\begin{aligned} H(X, f(X)) &= H(X) + H(f(X)|X) = H(X), \\ H(X, f(X)) &= H(f(X)) + H(X|f(X)) \geq H(f(X)). \end{aligned}$$

2. Letting X take on the values ± 1 and letting $f(x) = x^2$, show that it is possible to have $H(f(X)) < H(X)$.

3. In part (a), show that you have equality if and only if f is a one-to-one function (more precisely, f is one-to-one on the set of outputs of X that have nonzero probability).
4. The preceding results can be used to study the behavior of the run length coding of a sequence. Run length coding is a technique that is commonly used in data compression. Suppose that X_1, X_2, \dots, X_n are random variables that take the values 0 or 1. This sequence of random variables can be thought of as representing the output of a binary source. The run length coding of X_1, X_2, \dots, X_n is a sequence $\mathbf{L} = (L_1, L_2, \dots, L_k)$ that represents the lengths of consecutive symbols with the same value. For example, the sequence 110000100111 has a run length sequence of $\mathbf{L} = (2, 4, 1, 2, 3)$. Observe that \mathbf{L} is a function of X_1, X_2, \dots, X_n . Show that \mathbf{L} and X_1 uniquely determine X_1, X_2, \dots, X_n . Do \mathbf{L} and X_n determine X_1, X_2, \dots, X_n ? Using these observations and the preceding results, compare $H(X_1, X_2, \dots, X_n)$, $H(\mathbf{L})$, and $H(\mathbf{L}, X_1)$.
7. A bag contains five red balls, three white balls, and two black balls that are identical to each other in every manner except color.
1. Choose two balls from the bag with replacement. What is the entropy of this experiment?
 2. What is the entropy of choosing two balls without replacement? (Note: In both parts, the order matters; i.e., red then white is not the same as white then red.)
8. We often run into situations where we have a sequence of n random events. For example, a piece of text is a long sequence of letters. We are concerned with the rate of growth of the joint entropy as n increases. Define the entropy rate of a sequence $\mathbf{X} = \{X_k\}$ of random events as
- $$H_\infty(\mathbf{X}) = \lim_{n \rightarrow \infty} \frac{1}{n} H(X_1, X_2, \dots, X_n).$$
1. A very crude model for a language is to assume that subsequent letters in a piece of text are independent and come from identical probability distributions. Using this, show that the entropy rate equals $H(X_1)$.
 2. In general, there is dependence among the random variables. Assume that X_1, X_2, \dots, X_n have the same probability distribution but are somehow dependent on each other (for example, if I give you the letters TH you can guess that the next letter is E). Show that

$$H(X_1, X_2, \dots, X_n) \leq \sum_k H(X_k)$$

and thus that

$$H_\infty(X) \leq H(X_1)$$

(if the limit defining H_∞ exists).

9. Suppose we have a cryptosystem with only two possible plaintexts.

The plaintext a occurs with probability $1/3$ and b occurs with probability $2/3$. There are two keys, k_1 and k_2 , and each is used with probability $1/2$. Key k_1 encrypts a to A and b to B . Key k_2 encrypts a to B and b to A .

1. Calculate $H(P)$, the entropy for the plaintext.
2. Calculate $H(P|C)$, the conditional entropy for the plaintext given the ciphertext. (*Optional hint:* This can be done with no additional calculation by matching up this system with another well-known system.)

10. Consider a cryptosystem $\{P, K, C\}$.

1. Explain why $H(P, K) = H(C, P, K) = H(P) + H(K)$.

2. Suppose the system has perfect secrecy. Show that

$$H(C, P) = H(C) + H(P)$$

and

$$H(C) = H(K) - H(K|C, P).$$

3. Suppose the system has perfect secrecy and, for each pair of plaintext and ciphertext, there is at most one corresponding key that does the encryption. Show that $H(C) = H(K)$.

11. Prove that for a cryptosystem $\{P, K, C\}$ we have

$$H(C|P) = H(P, K, C) - H(P) - H(K|C, P) = H(K) - H(K|C, P).$$

12. Consider a Shamir secret sharing scheme where any five people of a set of 20 can determine the secret K , but no fewer can do so. Let $H(K)$ be the entropy of the choice of K , and let $H(K|S_1)$ be the conditional entropy of K , given the information supplied to the first person. What are the relative sizes of $H(K)$ and $H(K|S_1)$? (Larger, smaller, equal?)

13. Let X be a random event taking on the values $1, 2, 3, \dots, 36$, all with equal probability.

1. What is the entropy $H(X)$?
 2. Let $Y = X^{36} \pmod{37}$. What is $H(Y)$?
- 14.
1. Show that the maximum of
 $-p \log_2 p - (1-p) \log_2 (1-p)$ for $0 \leq p \leq 1$
occurs when $p = 1/2$.
 2. Let $p_i \geq 0$ for $1 \leq i \leq n$. Show that the maximum of

$$-\sum_i p_i \log_2 p_i,$$
subject to the constraint $\sum_i p_i = 1$, occurs when
 $p_1 = \dots = p_n$. (Hint: Lagrange multipliers could be useful in this problem.)
- 15.
1. Suppose we define
 $\tilde{H}(Y|X) = -\sum_{x,y} p_Y(y|x) \log_2 p_Y(y|x)$. Show
that if X and Y are independent, and X has $|X|$
possible outputs, then $\tilde{H}(Y|X) = |X|H(Y) \geq H(Y)$
.
 2. Use (a) to show that $\tilde{H}(Y|X)$ is not a good description
of the uncertainty of Y given X .

Chapter 21 Elliptic Curves

In the mid-1980s, Miller and Koblitz introduced elliptic curves into cryptography, and Lenstra showed how to use elliptic curves to factor integers. Since that time, elliptic curves have played an increasingly important role in many cryptographic situations. One of their advantages is that they seem to offer a level of security comparable to classical cryptosystems that use much larger key sizes. For example, it is estimated in [Blake et al.] that certain conventional systems with a 4096-bit key size can be replaced by 313-bit elliptic curve systems. Using much shorter numbers can represent a considerable savings in hardware implementations.

In this chapter, we present some of the highlights. For more details on elliptic curves and their cryptologic uses, see [Blake et al.], [Hankerson et al.], or [Washington]. For a list of elliptic curves recommended by NIST for cryptographic uses, see [FIPS 186-2].

21.1 The Addition Law

An elliptic curve E is the graph of an equation

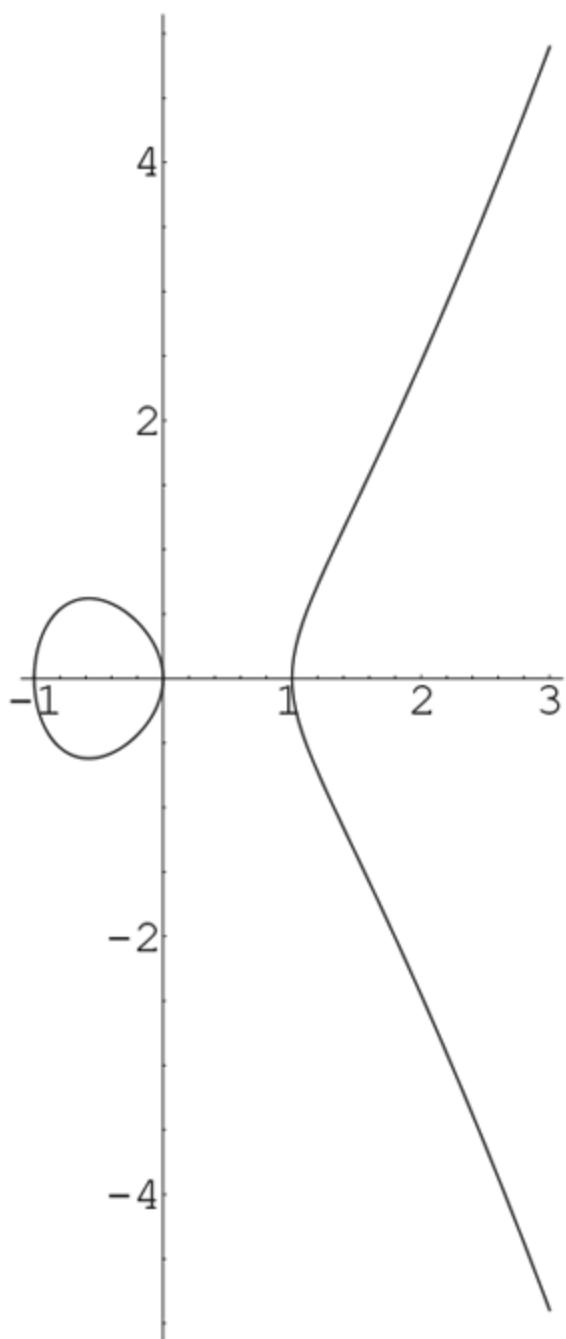
$$E : y^2 = x^3 + ax^2 + bx + c,$$

where a, b, c are in whatever is the appropriate set (rational numbers, real numbers, integers mod p , etc.). In other words, let K be the rational numbers, the real numbers, or the integers mod a prime p (or, for those who know what this means, any field of characteristic not 2; but see [Section 21.4](#)). Then we assume $a, b, c \in K$ and take E to be

$$\{(x, y) \mid x, y \in K, y^2 = x^3 + ax^2 + bx + c\}.$$

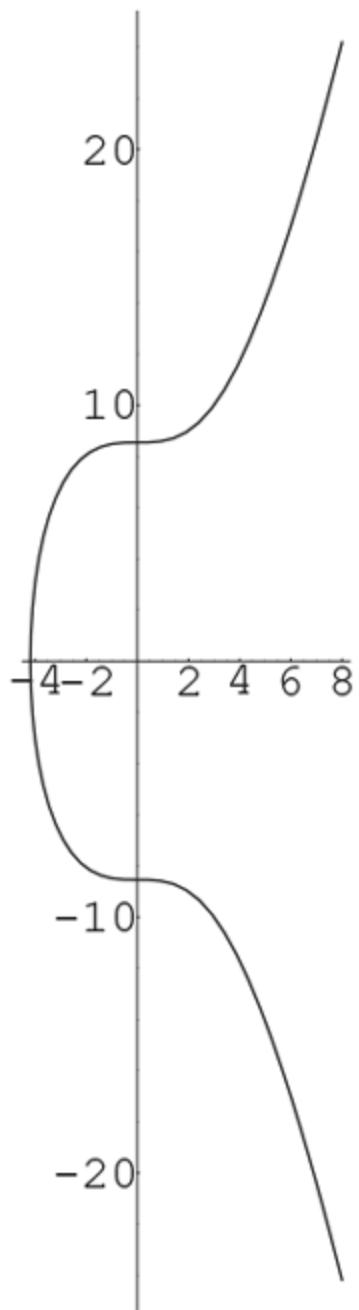
As will be discussed below, it is also convenient to include a point (∞, ∞) , which often will be denoted simply by ∞ .

Let's consider the case of real numbers first, since this case allows us to work with pictures. The graph E has two possible forms, depending on whether the cubic polynomial has one real root or three real roots. For example, the graphs of $y^2 = x(x + 1)(x - 1)$ and $y^2 = x^3 + 73$ are the following:



$$y^2 = x(x + 1)(x - 1)$$

21.1-1 Full Alternative Text



$$y^2 = x^3 + 73$$

21.1-2 Full Alternative Text

The case of two components (for example, $y^2 = x(x + 1)(x - 1)$) occurs when the cubic polynomial has three real roots. The case of one

component (for example, $y^2 = x^3 + 73$) occurs when the cubic polynomial has only one real root.

For technical reasons that will become clear later, we also include a “**point at infinity**,” denoted ∞ , which is most easily regarded as sitting at the top of the y -axis. It can be treated rigorously in the context of projective geometry (see [Washington]), but this intuitive notion suffices for what we need. The bottom of the y -axis is identified with the top, so ∞ also sits at the bottom of the y -axis.

Now let’s look at elliptic curves mod p , where p is a prime. For example, let E be given by

$$y^2 \equiv x^3 + 2x - 1 \pmod{5}.$$

We can list the points on E by letting x run through the values 0, 1, 2, 3, 4 and solving for y :

$$(0, 2), (0, 3), (2, 1), (2, 4), (4, 1), (4, 4), \infty.$$

Note that we again include a point ∞ .

Elliptic curves mod p are finite sets of points. It is these elliptic curves that are useful in cryptography.

Technical point: We assume that the cubic polynomial $x^3 + ax^2 + bx + c$ has no multiple roots. This means we exclude, for example, the graph of $y^2 = (x - 1)^2(x + 2)$. Such curves will be discussed in Subsection 21.3.1.

Technical point: For most situations, equations of the form $y^2 = x^3 + bx + c$ suffice for elliptic curves. In fact, in situations where we can divide by 3, a change of variables changes an equation $y^2 = x^3 + ax^2 + bx + c$ into an equation of the form $y^2 = x^3 + b'x + c'$. See [Exercise 1](#). However, sometimes it is necessary to consider elliptic curves given by equations of the form

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a_1, \dots, a_6 are constants. If we are working mod p , where $p > 3$ is prime, or if we are working with real, rational, or complex numbers, then simple changes of variables transform the present equation into the form $y^2 = x^3 + bx + c$. However, if we are working mod 2 or mod 3, or with a finite field of characteristic 2 or 3 (that is, $1 + 1 = 0$ or $1 + 1 + 1 = 0$), then we need to use the more general form. Elliptic curves over fields of characteristic 2 will be mentioned briefly in [Section 21.4](#).

Historical point: Elliptic curves are not ellipses. They received their name from their relation to *elliptic integrals* such as

$$\int_{z_1}^{z_2} \frac{dx}{\sqrt{x^3 + bx + c}} \quad \text{and} \quad \int_{z_1}^{z_2} \frac{x \, dx}{\sqrt{x^3 + bx + c}}$$

that arise in the computation of the arc length of ellipses.

The main reason elliptic curves are important is that we can use any two points on the curve to produce a third point on the curve. Given points P_1 and P_2 on E , we obtain a third point P_3 on E as follows (see [Figure 21.1](#)): Draw the line L through P_1 and P_2 (if $P_1 = P_2$, take the tangent line to E at P_1). The line L intersects E in a third point Q . Reflect Q through the x -axis (i.e., change y to $-y$) to get P_3 . Define a law of addition on E by

$$P_1 + P_2 = P_3.$$

Figure 21.1 Adding Points on an Elliptic Curve

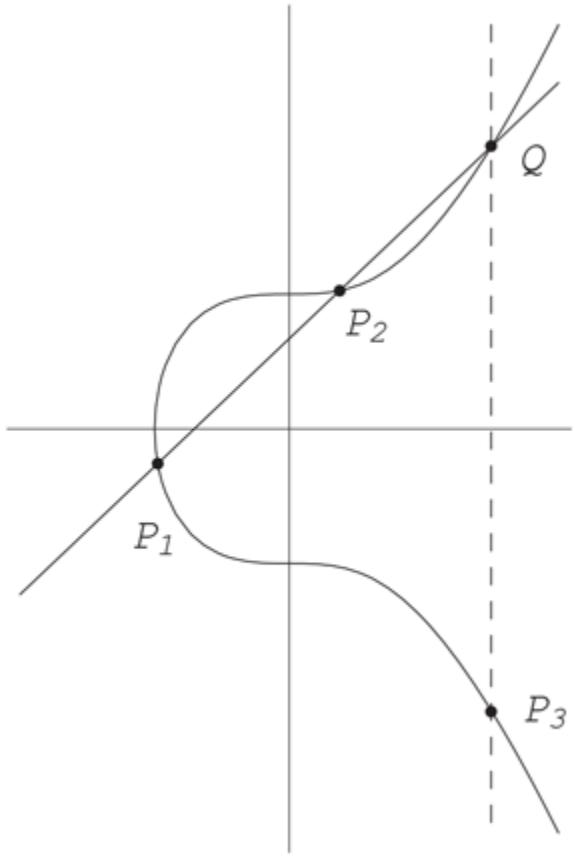


Figure 21.1 Full Alternative Text

Note that this is not the same as adding points in the plane.

Example

Suppose E is defined by $y^2 = x^3 + 73$. Let $P_1 = (2, 9)$ and $P_2 = (3, 10)$. The line L through P_1 and P_2 is

$$y = x + 7.$$

Substituting into the equation for E yields

$$(x + 7)^2 = x^3 + 73,$$

which yields $x^3 - x^2 - 14x + 24 = 0$. Since L intersects E in P_1 and P_2 , we already know two roots, namely $x = 2$ and $x = 3$. Moreover, the sum of the

three roots is minus the coefficient of x^2 (Exercise 1) and therefore equals 1. If x is the third root, then

$$2 + 3 + x = 1,$$

so the third point of intersection has $x = -4$. Since $y = x + 7$, we have $y = 3$, and $Q = (-4, 3)$. Reflect across the x -axis to obtain

$$(2, 0) + (3, 10) = P_3 = (-4, -3).$$

Now suppose we want to add P_3 to itself. The slope of the tangent line to E at P_3 is obtained by implicitly differentiating the equation for E :

$$2y \frac{dy}{dx} = 3x^2, \text{ so } \frac{dy}{dx} = \frac{3x^2}{2y} = -8,$$

where we have substituted $(x, y) = (-4, -3)$ from P_3 . In this case, the line L is $y = -8(x + 4) - 3$. Substituting into the equation for E yields

$$(-8(x + 4) - 3)^2 = x^3 + 73,$$

hence $x^3 - (-8)^2 x^2 + \dots = 0$. The sum of the three roots is 64 (= minus the coefficient of x^2). Because the line L is tangent to E , it follows that $x = -4$ is a double root. Therefore,

$$(-4) + (-4) + x = 64,$$

so the third root is $x = 72$. The corresponding value of y (use the equation of L) is -611 . Changing y to $-y$ yields

$$P_3 + P_3 = (72, 611).$$

What happens if we try to compute $P + \infty$? We make the convention that the lines through ∞ are vertical. Therefore, the line through $P = (x, y)$ and ∞ intersects E in P and also in $(x, -y)$. When we reflect $(x, -y)$ across the x -axis, we get back $P = (x, y)$. Therefore,

$$P + \infty = P.$$

We can also subtract points. First, observe that the line through (x, y) and $(x, -y)$ is vertical, so the third point of intersection with E is ∞ . The reflection across the x -axis is still ∞ (that's what we meant when we said ∞ sits at the top and at the bottom of the y -axis). Therefore,

$$(x, y) + (x, -y) = \infty.$$

Since ∞ plays the role of an additive identity (in the same way that 0 is the identity for addition with integers), we define

$$-(x, y) = (x, -y).$$

To subtract points $P - Q$, simply add P and $-Q$.

Another way to express the addition law is to say that

$$P + Q + R = \infty \Leftrightarrow P, Q, R \text{ are collinear.}$$

(See Exercise 17.)

For computations, we can ignore the geometrical interpretation and work only with formulas, which are as follows:

Addition Law

Let E be given by $y^2 = x^3 + bx + c$ and let

$$P_1 = (x_1, y_1), \quad P_2 = (x_2, y_2).$$

Then

$$P_1 + P_2 = P_3 = (x_3, y_3),$$

where

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

and

$$m = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if } P_1 \neq P_2 \\ (3x_1^2 + b)/(2y_1) & \text{if } P_1 = P_2. \end{cases}$$

If the slope m is infinite, then $P_3 = \infty$. There is one additional law: $\infty + P = P$ for all points P .

It can be shown that the addition law is associative:

$$(P + Q) + R = P + (Q + R).$$

It is also commutative:

$$P + Q = Q + P.$$

When adding several points, it therefore doesn't matter in what order the points are added nor how they are grouped together. In technical terms, we have found that the points of E form an abelian group. The point ∞ is the identity element of this group.

If k is a positive integer and P is a point on an elliptic curve, we can define

$$kP = P + P + \dots + P \quad (k \text{ summands}).$$

We can extend this to negative k . For example, $(-3)P = 3(-P) = (-P) + (-P) + (-P)$, where $-P$ is the reflection of P across the x -axis. The associative law means that we can group the summands in any way we choose when computing a multiple of a point. For example, suppose we want to compute $100P$. We do the additive version of successive squaring that was used in modular exponentiation:

$$\begin{aligned} 2P &= P + P \\ 4P &= 2P + 2P \\ 8P &= 4P + 4P \\ 16P &= 8P + 8P \\ 32P &= 16P + 16P \\ 64P &= 32P + 32P \\ 100P &= 64P + 32P + 4P. \end{aligned}$$

The associative law means, for example, that $4P$ can be computed as $2P + 2P = (P + P) + (P + P)$. It also could have been computed in what might seem to be a

more natural way as $((P + P) + P)$, but this is slower because it requires three additions instead of two.

For more examples, see Examples 41–44 in the Computer Appendices.

21.2 Elliptic Curves Mod p

If p is a prime, we can work with elliptic curves mod p using the aforementioned ideas. For example, consider

$$E : y^2 \equiv x^3 + 4x + 4 \pmod{5}.$$

The points on E are the pairs $(x, y) \pmod{5}$ that satisfy the equation, along with the point at infinity. These can be listed as follows. The possibilities for $x \pmod{5}$ are 0, 1, 2, 3, 4. Substitute each of these into the equation and find the values of y that solve the equation:

$$\begin{aligned} x \equiv 0 &\implies y^2 \equiv 4 && \implies y \equiv 2, 3 \pmod{5} \\ x \equiv 1 &\implies y^2 \equiv 9 \equiv 4 && \implies y \equiv 2, 3 \pmod{5} \\ x \equiv 2 &\implies y^2 \equiv 20 \equiv 0 && \implies y \equiv 0 \pmod{5} \\ x \equiv 3 &\implies y^2 \equiv 43 \equiv 3 && \implies \text{no solutions} \\ x \equiv 4 &\implies y^2 \equiv 84 \equiv 4 && \implies y \equiv 2, 3 \pmod{5} \\ x = \infty &\implies y = \infty. \end{aligned}$$

The points on E are
 $(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (4, 2), (4, 3), (\infty, \infty)$.

The addition of points on an elliptic curve mod p is done via the same formulas as given previously, except that a rational number a/b must be treated as ab^{-1} , where $b^{-1}b \equiv 1 \pmod{p}$. This requires that $\gcd(b, p) = 1$.

More generally, it is possible to develop a theory of elliptic curves mod n for any integer n . In this case, when we encounter a fraction a/b , we need to have $\gcd(b, n) = 1$. The situations where this fails form the key to using elliptic curves for factorization, as we'll see in [Section 21.3](#). There are various technical problems in the general theory that arise when $1 < \gcd(b, n) < n$, but the method to overcome these will not be needed in the following. For details on how to treat this case, see [Washington]. For our purposes, when we encounter an

elliptic curve mod a composite n , we can pretend n is prime. If something goes wrong, we usually obtain useful information about n , for example its factorization.

Example

Let's compute $(1, 2) + (4, 3)$ on the curve just considered. The slope is

$$m \equiv \frac{3 - 2}{4 - 1} \equiv 2 \pmod{5}.$$

Therefore,

$$\begin{aligned} x_3 &\equiv m^2 - x_1 - x_2 \equiv 2^2 - 1 - 4 \equiv 4 \pmod{5} \\ y_3 &\equiv m(x_1 - x_3) - y_1 \equiv 2(1 - 4) - 2 \equiv 2 \pmod{5}. \end{aligned}$$

This means that

$$(1, 2) + (4, 3) = (4, 2).$$

Example

Here is a somewhat larger example. Let $n = 2773$. Let

$$E : y^2 \equiv x^3 + 4x + 4 \pmod{2773}, \text{ and } P = (1, 3).$$

Let's compute $2P = P + P$. To get the slope of the tangent line, we differentiate implicitly and evaluate at $(1, 3)$:

$$2y dy = (3x^2 + 4) dx \Rightarrow \frac{dy}{dx} = \frac{3x^2 + 4}{2y} = \frac{3x^2 + 4}{6}.$$

But we are working mod 2773. Using the extended Euclidean algorithm (see [Section 3.2](#)), we find that $2311 \cdot 6 \equiv 1 \pmod{2773}$, so we can replace $1/6$ by 2311. Therefore,

$$m \equiv \frac{7}{6} \equiv 7 \times 2311 \equiv 2312 \pmod{2773}.$$

The formulas yield

$$\begin{aligned}x_3 &\equiv 2312^2 - 1 - 1 \equiv 1771 \pmod{2773} \\y_3 &\equiv 2312(1 - 1771) - 3 \equiv 705 \pmod{2773}.\end{aligned}$$

The final answer is

$$2P = P + P = (1771, 705).$$

Now that we're done with the example, we mention that 2773 is not prime. When we try to calculate $3P$ in [Section 21.3](#), we'll obtain the factorization of 2773.

21.2.1 Number of Points Mod p

Let $E : y^2 \equiv x^3 + bx + c \pmod{p}$ be an elliptic curve, where $p \geq 5$ is prime. We can list the points on E by letting $x = 0, 1, \dots, p-1$ and seeing when $x^3 + bx + c$ is a square mod p . Since half of the nonzero numbers are squares mod p , we expect that $x^3 + bx + c$ will be a square approximately half the time. When it is a nonzero square, there are two square roots: y and $-y$. Therefore, approximately half the time we get two values of y and half the time we get no y . Therefore, we expect around p points. Including the point ∞ , we expect a total of approximately $p+1$ points. In the 1930s, H. Hasse made this estimate more precise.

Hasse's Theorem

Suppose $E \pmod{p}$ has N points. Then

$$|N - p - 1| < 2\sqrt{p}.$$

The proof of this theorem is well beyond the scope of this book (for a proof, see [Washington]). It can also be shown that whenever N and p satisfy the inequality of

the theorem, there is an elliptic curve $E \bmod p$ with exactly N points.

If p is large, say around 10^{20} , it is infeasible to count the points on an elliptic curve by listing them. More sophisticated algorithms have been developed by Schoof, Atkin, Elkies, and others to deal with this problem. See the Sage Appendix.

21.2.2 Discrete Logarithms on Elliptic Curves

Recall the classical discrete logarithm problem: We know that $x \equiv g^k \pmod{p}$ for some k , and we want to find k . There is an elliptic curve version: Suppose we have points A, B on an elliptic curve E and we know that $B = kA (= A + A + \dots + A)$ for some integer k . We want to find k . This might not look like a logarithm problem, but it is clearly the analog of the classical discrete logarithm problem. Therefore, it is called the **discrete logarithm problem** for elliptic curves.

There is no good general attack on the discrete logarithm problem for elliptic curves. There is an analog of the Pohlig-Hellman attack that works in some situations. Let E be an elliptic curve mod a prime p and let n be the smallest integer such that $nA = \infty$. If n has only small prime factors, then it is possible to calculate the discrete logarithm k mod the prime powers dividing n and then use the Chinese remainder theorem to find k (see [Exercise 25](#)). The Pohlig-Hellman attack can be thwarted by choosing E and A so that n has a large prime factor.

There is no replacement for the index calculus attack described in [Section 10.2](#). This is because there is no good analog of “small.” You might try to use points with small coordinates in place of the “small primes,” but this

doesn't work. When you factor a number by dividing off the prime factors one by one, the quotients get smaller and smaller until you finish. On an elliptic curve, you could have a point with fairly small coordinates, subtract off a small point, and end up with a point with large coordinates (see Computer Problem 5). So there is no good way to know when you are making progress toward expressing a point in terms of the factor base of small points.

The Baby Step, Giant Step attack on discrete logarithms works for elliptic curves (Exercise 13(b)), although it requires too much memory to be practical in most situations. For other attacks, see [Blake et al.] and [Washington].

21.2.3 Representing Plaintext

In most cryptographic systems, we must have a method for mapping our original message into a numerical value upon which we can perform mathematical operations. In order to use elliptic curves, we need a method for mapping a message onto a point on an elliptic curve. Elliptic curve cryptosystems then use elliptic curve operations on that point to yield a new point that will serve as the ciphertext.

The problem of encoding plaintext messages as points on an elliptic curve is not as simple as it was in the conventional case. In particular, there is no known polynomial time, deterministic algorithm for writing down points on an arbitrary elliptic curve $E \pmod{p}$. However, there are fast probabilistic methods for finding points, and these can be used for encoding messages. These methods have the property that with small probability they will fail to produce a point. By appropriately choosing parameters, this probability can be made arbitrarily small, say on the order of $1/2^{30}$.

Here is one method, due to Koblitz. The idea is the following. Let $E : y^2 \equiv x^3 + bx + c \pmod{p}$ be the elliptic curve. The message m (already represented as a number) will be embedded in the x -coordinate of a point. However, the probability is only about $1/2$ that $m^3 + bm + c$ is a square mod p . Therefore, we adjoin a few bits at the end of m and adjust them until we get a number x such that $x^3 + bx + c$ is a square mod p .

More precisely, let K be a large integer so that a failure rate of $1/2^K$ is acceptable when trying to encode a message as a point. Assume that m satisfies

$(m+1)K < p$. The message m will be represented by a number $x = mK + j$, where $0 \leq j < K$. For $j = 0, 1, \dots, K-1$, compute $x^3 + bx + c$ and try to calculate the square root of $x^3 + bx + c \pmod{p}$.

For example, if $p \equiv 3 \pmod{4}$, the method of Section 3.9 can be used. If there is a square root y , then we take $P_m = (x, y)$; otherwise, we increment j by one and try again with the new x . We repeat this until either we find a square root or $j = K$. If j ever equals K , then we fail to map a message to a point. Since $x^3 + bx + c$ is a square approximately half of the time, we have about a $1/2^K$ chance of failure.

In order to recover the message from the point $P_m = (x, y)$ we simply calculate m by

$$m = \lfloor x/K \rfloor,$$

where $\lfloor x/K \rfloor$ denotes the greatest integer less than or equal to x/K .

Example

Let $p = 179$ and suppose that our elliptic curve is $y^2 = x^3 + 2x + 7$. If we are satisfied with a failure rate of $1/2^{10}$, then we may take $K = 10$. Since we need $(m+1)K < 179$, we need $0 \leq m \leq 16$. Suppose

our message is $m = 5$. We consider x of the form $mK + j = 50 + j$. The possible choices for x are 50, 51, \dots , 59. For $x = 51$ we get $x^3 + 2x + 7 \equiv 121 \pmod{179}$, and $11^2 \equiv 121 \pmod{179}$. Thus, we represent the message $m = 5$ by the point $P_m = (51, 11)$. The message m can be recovered by $m = [51/10] = 5$.

21.3 Factoring with Elliptic Curves

Suppose $n = pq$ is a number we wish to factor. Choose a random elliptic curve mod n and a point on the curve. In practice, one chooses several (around 14 for numbers around 50 digits; more for larger integers) curves with points and runs the algorithm in parallel.

How do we choose the curve? First, choose a point P and a coefficient b . Then choose c so that P lies on the curve $y^2 = x^3 + bx + c$. This is much more efficient than choosing b and c and then trying to find a point.

For example, let $n = 2773$. Take $P = (1, 3)$ and $b = 4$. Since we want $3^2 \equiv 1^3 + 4 \cdot 1 + c$, we take $c = 4$. Therefore, our curve is

$$E : y^2 \equiv x^3 + 4x + 4 \pmod{2773}.$$

We calculated $2P = (1771, 705)$ in a previous example. Note that during the calculation, we needed to find $6^{-1} \pmod{2773}$. This required that $\gcd(6, 2773) = 1$ and used the extended Euclidean algorithm, which was essentially a gcd calculation.

Now let's calculate $3P = 2P + P$. The line through the points $2P = (1771, 705)$ and $P = (1, 3)$ has slope $702/1770$. When we try to invert 1770 mod 2773, we find that $\gcd(1770, 2773) = 59$, so we cannot do this. So what do we do? Our original goal was to factor 2773, so we don't need to do anything more. We have found the factor 59, which yields the factorization $2773 = 59 \cdot 47$.

Here's what happened. Using the Chinese remainder theorem, we can regard E as a pair of elliptic curves, one mod 59 and the other mod 47. It turns out that

$3P = \infty \pmod{59}$, while $4P = \infty \pmod{47}$.

Therefore, when we tried to compute $3P$, we had a slope that was infinite mod 59 but finite mod 47. In other words, we had a denominator that was 0 mod 59 but nonzero mod 47. Taking the gcd allowed us to isolate the factor 59.

The same type of idea is the basis for many factoring algorithms. If $n = pq$, you cannot separate p and q as long as they behave identically. But if you can find something that makes them behave slightly differently, then they can be separated. In the example, the multiples of P reached ∞ faster mod 59 than mod 47. Since in general the primes p and q should act fairly independently of each other, one would expect that for most curves $E \pmod{pq}$ and points P , the multiples of P would reach $\infty \pmod{p}$ and \pmod{q} at different times. This will cause the gcd to find either p or q .

Usually, it takes several more steps than 3 or 4 to reach $\infty \pmod{p}$ or \pmod{q} . In practice, one multiplies P by a large number with many small prime factors, for example, $10000!$. This can be done via successive doubling (the additive analog of successive squaring; see [Exercise 21](#)). The hope is that this multiple of P is ∞ either mod p or mod q . This is very much the analog of the $p - 1$ method of factoring. However, recall that the $p - 1$ method (see [Section 9.4](#)) usually doesn't work when $p - 1$ has a large prime factor. The same type of problem could occur in the elliptic curve method just outlined when the number m such that mP equals ∞ has a large prime factor. If this happens (so the method fails to produce a factor after a while), we simply change to a new curve E . This curve will be independent of the previous curve and the value of m such that $mP = \infty$ should have essentially no relation to the previous m . After several tries (or if several curves are treated in parallel), a good curve is often found, and the number $n = pq$ is factored. In contrast, if the $p - 1$ method

fails, there is nothing that can be changed other than using a different factorization method.

Example

We want to factor $n = 455839$. Choose

$$E : y^2 \equiv x^3 + 5x - 5, \quad P = (1, 1).$$

Suppose we try to compute $10!P$. There are many ways to do this. One is to compute

$2!P, 3!P = 3(2!P), 4!P = 4(3!P), \dots$. If we do this, everything is fine through $7!P$, but $8!P$ requires inverting 599 (mod n). Since $\gcd(599, n) = 599$, we can factor n as 599×761 .

Let's examine this more closely. A computation shows that $E \pmod{599}$ has $640 = 2^7 \times 5$ points and $E \pmod{761}$ has $777 = 3 \times 7 \times 37$ points. Moreover, 640 is the smallest positive m such that $mP = \infty$ on $E \pmod{599}$, and 777 is the smallest positive m such that $mP = \infty$ on $E \pmod{761}$. Since 8! is a multiple of 640, it is easy to see that $8!P = \infty$ on $E \pmod{599}$, as we calculated. Since 8! is not a multiple of 777, it follows that $8!P \neq \infty$ on $E \pmod{761}$. Recall that we obtain ∞ when we divide by 0, so calculating $8!P$ asked us to divide by 0 (mod 599). This is why we found the factor 599.

For more examples, see Examples 45 and 46 in the Computer Appendices.

In general, consider an elliptic curve $E \pmod{p}$ for some prime p . The smallest positive m such that $mP = \infty$ on this curve divides the number N of points on $E \pmod{p}$ (if you know group theory, you'll recognize this as a corollary of Lagrange's theorem), so $NP = \infty$. Quite often, m will be N or a large divisor of N . In any case, if N is a product of small primes, then

$B!$ will be a multiple of N for a reasonably small value of B . Therefore, $B!P = \infty$.

A number that has only small prime factors is called **smooth**. More precisely, if all the prime factors of an integer are less than or equal to B , then it is called **B-smooth**. This concept played a role in the $x^2 \equiv y^2$ method and the $p - 1$ factoring method (Section 9.4), and the index calculus attack on discrete logarithms (Section 10.2).

Recall from Hasse's theorem that N is an integer near p . It is possible to show that the density of smooth integers is large enough (we'll leave *small* and *large* undefined here) that if we choose a random elliptic curve $E \pmod{p}$, then there is a reasonable chance that the number N is smooth. This means that the elliptic curve factorization method should find p for this choice of the curve. If we try several curves $E \pmod{n}$, where $n = pq$, then it is likely that at least one of the curves $E \pmod{p}$ or $E \pmod{q}$ will have its number of points being smooth.

In summary, the advantage of the elliptic curve factorization method over the $p - 1$ method is the following. The $p - 1$ method requires that $p - 1$ is smooth. The elliptic curve method requires only that there are enough smooth numbers near p so that at least one of some randomly chosen integers near p is smooth. This means that elliptic curve factorization succeeds much more often than the $p - 1$ method.

The elliptic curve method seems to be best suited for factoring numbers of medium size, say around 40 or 50 digits. These numbers are no longer used for the security of factoring-based systems such as RSA, but it is sometimes useful in other situations to have a fast factorization method for such numbers. Also, the elliptic curve method is effective when a large number has a

small prime factor, say of 10 or 20 decimal digits. For large numbers where the prime factors are large, the quadratic sieve and number field sieve are superior (see [Section 9.4](#)).

21.3.1 Singular Curves

In practice, the case where the cubic polynomial $x^3 + bx + c$ has multiple roots rarely arises. But what happens if it does? Does the factorization algorithm still work? The discriminant $4b^3 + 27c^2$ is zero if and only if there is a multiple root (this is the cubic analog of the fact that $ax^2 + bx + c$ has a double root if and only if $b^2 - 4ac = 0$). Since we are working mod $n = pq$, the result says that there is a multiple root mod n if and only if the discriminant is 0 mod n . Since n is composite, there is also the intermediate case where the gcd of n and the discriminant is neither 1 nor n . But this gives a nontrivial factor of n , so we can stop immediately in this case.

Example

Let's look at an example:

$$y^2 = x^3 - 3x + 2 = (x - 1)^2(x + 2).$$

Given a point $P = (x, y)$ on this curve, we associate the number

$$(y + \sqrt{3}(x - 1))/(y - \sqrt{3}(x - 1)).$$

It can be shown that adding the points on the curve corresponds to multiplying the corresponding numbers. The formulas still work, as long as we don't use the point $(1, 0)$. Where does this come from? The two lines tangent to the curve at $(1, 0)$ are $y + \sqrt{3}(x - 1) = 0$

and $y - \sqrt{3}(x - 1) = 0$. This number is simply the ratio of these two expressions.

Since we need to work mod n , we give an example mod 143. We choose 143 since 3 is a square mod 143; in fact, $82^2 \equiv 3 \pmod{143}$. If this were not the case, things would become more technical with this curve. We could easily rectify the situation by choosing a new curve.

Consider the point $P = (-1, 2)$ on $y^2 = x^3 - 3x + 2 \pmod{143}$. Look at its multiples:

$$P = (-1, 2), \quad 2P = (2, 141), \quad 3P = (112, 101), \quad 4P = (10, 20).$$

When trying to compute $5P$, we find the factor 11 of 143.

Recall that we are assigning numbers to each point on the curve, other than $(1, 1)$. Since we are working mod 143, we use 82 in place of $\sqrt{3}$. Therefore, the number corresponding to $(-1, 2)$ is

$$(2 + 82(-1 - 1))/(2 - 82(-1 - 1)) = 80 \pmod{143}.$$

We can compute the numbers for all the points above:

$$P \leftrightarrow 80, \quad 2P \leftrightarrow 108, \quad 3P \leftrightarrow 60, \quad 4P \leftrightarrow 81.$$

Let's compare with the powers of 80 mod 143:

$$80^1 \equiv 80, \quad 80^2 \equiv 108, \quad 80^3 \equiv 60, \quad 80^4 \equiv 81, \quad 80^5 \equiv 45.$$

We get the same numbers. This is simply the fact mentioned previously that the addition of points on the curve corresponds to multiplication of the corresponding numbers. Moreover, note that $45 \equiv 1 \pmod{11}$, but not mod 13. This corresponds to the fact that 5 times the point $(-1, 2)$ is ∞ mod 11 but not mod 13. Note that 1 is the multiplicative identity for multiplication mod 11, while ∞ is the additive identity for addition on the curve.

It is easy to see from the preceding that factorization using the curve $y^2 = x^3 - 3x + 2$ is essentially the

same as using the classical $p - 1$ factorization method (see Section 9.4).

In the preceding example, the cubic equation had a double root. An even worse possibility is the cubic having a triple root. Consider the curve

$$y^2 = x^3.$$

To a point $(x, y) \neq (0, 0)$ on this curve, associate the number x/y . Let's start with the point $P = (1, 1)$ and compute its multiples:

$$P = (1, 1), \quad 2P = \left(\frac{1}{4}, \frac{1}{8}\right), \quad 3P = \left(\frac{1}{9}, \frac{1}{27}\right), \quad \dots, \quad mP = \left(\frac{1}{m^2}, \frac{1}{m^3}\right).$$

Note that the corresponding numbers x/y are $1, 2, 3, \dots, m$. Adding the points on the curve corresponds to adding the numbers x/y .

If we are using the curve $y^2 = x^3$ to factor n , we need to change the points mP to integers mod n , which requires finding inverses for m^2 and m^3 mod n . This is done by the extended Euclidean algorithm, which is essentially a gcd computation. We find a factor of n when $\gcd(m, n) \neq 1$. Therefore, this method is essentially the same as computing in succession $\gcd(2, n), \gcd(3, n), \gcd(4, n), \dots$ until a factor is found. This is a slow version of trial division, the oldest factorization technique known. Of course, in the elliptic curve factorization algorithm, a large multiple $(B!)P$ of P is usually computed. This is equivalent to factoring by computing $\gcd(B!, n)$, a method that is often used to test for prime factors up to B .

In summary, we see that the $p - 1$ method and trial division are included in the elliptic curve factorization algorithm if we allow singular curves.

21.4 Elliptic Curves in Characteristic 2

Many applications use elliptic curves mod 2, or elliptic curves defined over the finite fields $GF(2^n)$ (these are described in [Section 3.11](#)). This is often because mod 2 adapts well to computers. In 1999, NIST recommended 15 elliptic curves for cryptographic uses (see [[FIPS 186-2](#)]). Of these, 10 are over finite fields $GF(2^n)$.

If we're working mod 2, the equations for elliptic curves need to be modified slightly. There are many reasons for this. For example, the derivative of y^2 is $2yy' = 0$, since 2 is the same as 0. This means that the tangent lines we compute are vertical, so $2P = \infty$ for all points P . A more sophisticated explanation is that the curve $y^2 \equiv x^3 + bx + c \pmod{2}$ has singularities (points where the partial derivatives with respect to x and y simultaneously vanish).

The equations we need are of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a_1, \dots, a_6 are constants. The addition law is slightly more complicated. We still have three points adding to infinity if and only if they lie on a line. Also, the lines through ∞ are vertical. But, as we'll see in the following example, finding $-P$ from P is not the same as before.

Example

Let $E : y^2 + y \equiv x^3 + x \pmod{2}$. As before, we can list the points on E :

$$(0, 0), \quad (0, 1), \quad (1, 0), \quad (1, 1), \quad \infty.$$

Let's compute $(0, 0) + (1, 1)$. The line through these two points is $y = x$. Substituting into the equation for E yields $x^2 + x \equiv x^3 + x$, which can be rewritten as $x^2(x + 1) \equiv 0$. The roots are $x = 0, 0, 1 \pmod{2}$. Therefore, the third point of intersection also has $x = 0$. Since it lies on the line $y = x$, it must be $(0, 0)$. (This might be puzzling. What is happening is that the line is tangent to E at $(0, 0)$ and also intersects E in the point $(1, 1)$.) As before, we now have

$$(0, 0) + (0, 0) + (1, 1) = \infty.$$

To get $(0, 0) + (1, 1)$ we need to compute $\infty - (0, 0)$. This means we need to find P such that $P + (0, 0) = \infty$. A line through ∞ is still a vertical line. In this case, we need one through $(0, 0)$, so we take $x = 0$. This intersects E in the point $P = (0, 1)$. We conclude that $(0, 0) + (0, 1) = \infty$. Putting everything together, we see that

$$(0, 0) + (1, 1) = (0, 1).$$

In most applications, elliptic curves mod 2 are not large enough. Therefore, elliptic curves over finite fields are used. For an introduction to finite fields, see [Section 3.11](#). However, in the present section, we only need the field $GF(4)$, which we now describe.

Let

$$GF(4) = \{0, 1, \omega, \omega^2\},$$

with the following laws:

1. $0 + x = x$ for all x .
2. $x + x = 0$ for all x .
3. $1 \cdot x = x$ for all x .
4. $1 + \omega = \omega^2$.
5. Addition and multiplication are commutative and associative, and the distributive law holds: $x(y + z) = xy + xz$ for all x, y, z .

Since

$$\omega^3 = \omega \cdot \omega^2 = \omega \cdot (1 + \omega) = \omega + \omega^2 = \omega + (1 + \omega) = 1,$$

we see that ω^2 is the multiplicative inverse of ω .

Therefore, every nonzero element of $GF(4)$ has a multiplicative inverse.

Elliptic curves with coefficients in finite fields are treated just like elliptic curves with integer coefficients.

Example

Consider

$$E : y^2 + xy = x^3 + \omega,$$

where $\omega \in GF(4)$ is as before. Let's list the points of E with coordinates in $GF(4)$:

$$\begin{aligned} x = 0 &\Rightarrow y^2 = \omega \Rightarrow y = \omega^2 \\ x = 1 &\Rightarrow y^2 + y = 1 + \omega = \omega^2 \Rightarrow \text{no solutions} \\ x = \omega &\Rightarrow y^2 + \omega y = \omega^2 \Rightarrow y = 1, \omega^2 \\ x = \omega^2 &\Rightarrow y^2 + \omega^2 y = 1 + \omega = \omega^2 \Rightarrow \text{no solutions} \\ x = \infty &\Rightarrow y = \infty. \end{aligned}$$

The points on E are therefore

$$(0, \omega^2), \quad (\omega, 1), \quad (\omega, \omega^2), \quad \infty.$$

Let's compute $(0, \omega^2) + (\omega, \omega^2)$. The line through these two points is $y = \omega^2$. Substitute this into the equation for E :

$$\omega^4 + \omega^2 x = x^3 + \omega,$$

which becomes $x^3 + \omega^2 x = 0$. This has the roots $x = 0, \omega, \omega$. The third point of intersection of the line and E is therefore (ω, ω^2) , so

$$(0, \omega^2) + (\omega, \omega^2) + (\omega, \omega^2) = \infty.$$

We need $-(\omega, \omega^2)$, namely the point P with
 $P + (\omega, \omega^2) = \infty$. The vertical line $x = \omega$ intersects
 E in $P = (\omega, 1)$, so

$$(0, \omega^2) + (\omega, \omega^2) = (\omega, 1).$$

For cryptographic purposes, elliptic curves are used over fields $GF(2^n)$ with n large, say at least 150.

21.5 Elliptic Curve Cryptosystems

Elliptic curve versions exist for many cryptosystems, in particular those involving discrete logarithms. An advantage of elliptic curves compared to working with integers mod p is the following. In the integers, it is possible to use the factorization of integers into primes (especially small primes) to attack the discrete logarithm problem. This is known as the index calculus and is described in [Section 10.2](#). There seems to be no good analog of this method for elliptic curves. Therefore, it is possible to use smaller primes, or smaller finite fields, with elliptic curves and achieve a level of security comparable to that for much larger integers mod p . This allows great savings in hardware implementations, for example.

In the following, we describe three elliptic curve versions of classical algorithms. Here is a general procedure for changing a classical system based on discrete logarithms into one using elliptic curves:

Nonzero numbers mod p	\longleftrightarrow	Points on an elliptic curve
Multiplication mod p	\longleftrightarrow	Elliptic curve addition
1 (multiplicative identity)	\longleftrightarrow	∞ (additive identity)
Division mod p	\longleftrightarrow	Subtraction of points
Exponentiation: g^k	\longleftrightarrow	Integer times a point: $kP = P + \dots + P$
$p - 1$	\longleftrightarrow	$n =$ number of points on the curve

$$\text{Fermat: } a^{p-1} \equiv 1 \quad \longleftrightarrow \quad nP = \infty \text{ (Lagrange's theorem)}$$

$$\text{Discrete log problem:} \quad \longleftrightarrow \quad \text{Elliptic curve discrete log problem:}$$

$$\text{Solve } g^k \equiv h \text{ for } k \quad \longleftrightarrow \quad \text{Solve } kP = Q \text{ for } k$$

Notes:

1. The elliptic curve is an elliptic curve mod some prime, so n , the number of points on the curve, including ∞ , is finite.
2. Addition and subtraction of points on an elliptic curve are of equivalent complexity (if $Q = (x, y)$, then $-Q = (x, -y)$) and $P - Q$ is computed as $P + (-Q)$), but multiplication mod p is much easier than division mod p (via the extended Euclidean algorithm). Both mod p operations are usually simpler than the elliptic curve operations.
3. The elliptic curve discrete log problem is believed to be harder than the mod p discrete log problem.
4. If we fix a number m and look at the set of all integers mod m , then the analogues of the above are: addition mod m , the additive identity 0 , subtraction mod m , multiplying an integer times a number mod m (that is, $ka = a + a + \dots + a \pmod{m}$), m = the number of integers mod m , the relation $ma \equiv 0 \pmod{m}$, and the additive discrete log problem: Solve $ka \equiv b \pmod{m}$ for k , which can be done easily via the Extended Euclidean algorithm. This shows that the difficulty of a discrete log problem depends on the binary operation.

21.5.1 An Elliptic Curve ElGamal Cryptosystem

We recall the non-elliptic curve version. Alice wants to send a message x to Bob, so Bob chooses a large prime p and an integer $\alpha \pmod{p}$. He also chooses a secret integer s and computes $\beta \equiv \alpha^s \pmod{p}$. Bob makes p, α, β public and keeps s secret. Alice chooses a random k and computes y_1 and y_2 , where

$$y_1 \equiv \alpha^k \text{ and } y_2 \equiv x\beta^k \pmod{p}.$$

She sends (y_1, y_2) to Bob, who then decrypts by calculating

$$x \equiv y_2 y_1^{-s} \pmod{p}.$$

Now we describe the elliptic curve version. Bob chooses an elliptic curve $E \pmod{p}$, where p is a large prime. He chooses a point α on E and a secret integer s . He computes

$$\beta = s\alpha \quad (= \alpha + \alpha + \cdots + \alpha).$$

The points α and β are made public, while s is kept secret. Alice expresses her message as a point x on E (see [Section 21.5](#)). She chooses a random integer k , computes

$$y_1 = k\alpha \text{ and } y_2 = x + k\beta,$$

and sends the pair y_1, y_2 to Bob. Bob decrypts by calculating

$$x = y_2 - sy_1.$$

A more workable version of this system is due to Menezes and Vanstone. It is described in [Stinson1, p. 189].

Example

We must first generate a curve. Let's use the prime $p = 8831$, the point $G = (x, y) = (4, 11)$, and $b = 3$. To make G lie on the curve $y^2 \equiv x^3 + bx + c \pmod{p}$, we take $c = 45$. Alice has a message, represented as a point $P_m = (5, 1743)$, that she wishes to send to Bob. Here is how she does it.

Bob has chosen a secret random number $s_B = 3$ and has published the point $s_B G = (413, 1808)$.

Alice downloads this and chooses a random number $k = 8$. She sends Bob $kG = (5415, 6321)$ and

$P_m + k(s_B G) = (6626, 3576)$. He first calculates $s_B(kG) = 3(5415, 6321) = (673, 146)$. He now subtracts this from $(6626, 3576)$:

$$(6626, 3576) - (673, 146) = (6626, 3576) + (673, -146) = (5, 1743).$$

Note that we subtracted points by using the rule $P - Q = P + (-Q)$ from [Section 21.1](#).

For another example, see [Example 47](#) in the Computer Appendices.

21.5.2 Elliptic Curve Diffie-Hellman Key Exchange

Alice and Bob want to exchange a key. In order to do so, they agree on a public basepoint G on an elliptic curve E : $y^2 \equiv x^3 + bx + c \pmod{p}$. Let's choose $p = 7211$ and $b = 1$ and $G = (3, 5)$. This forces us to choose $c = 7206$ in order to have the point on the curve. Alice chooses N_A randomly and Bob chooses N_B randomly. Let's suppose $N_A = 12$ and $N_B = 23$. They keep these private to themselves but publish $N_A G$ and $N_B G$. In our case, we have

$$N_A G = (1794, 6375) \text{ and } N_B G = (3861, 1242).$$

Alice now takes $N_B G$ and multiplies by N_A to get the key:

$$N_A(N_B G) = 12(3861, 1242) = (1472, 2098).$$

Similarly, Bob takes $N_A G$ and multiplies by N_B to get the key:

$$N_B(N_A G) = 23(1794, 6375) = (1472, 2098).$$

Notice that they have the same key.

For another example, see [Example 48](#) in the Computer Appendices.

21.5.3 ElGamal Digital Signatures

There is an elliptic curve analog of the procedure described in [Section 13.2](#). A few modifications are needed to account for the fact that we are working with both integers and points on an elliptic curve.

Alice wants to sign a message m (which might actually be the hash of a long message). We assume m is an integer. She fixes an elliptic curve $E \pmod{p}$, where p is a large prime, and a point A on E . We assume that the number of points N on E has been calculated and assume $0 \leq m < N$ (if not, choose a larger p). Alice also chooses a private integer a and computes $B = aA$. The prime p , the curve E , the integer n , and the points A and B are made public. To sign the message, Alice does the following:

1. Chooses a random integer k with $1 \leq k < N$ and $\gcd(k, N) = 1$, and computes $R = kA = (x, y)$
2. Computes $s \equiv k^{-1}(m - ax) \pmod{N}$
3. Sends the signed message (m, R, s) to Bob

Note that R is a point on E , and m and s are integers.

Bob verifies the signature as follows:

1. Downloads Alice's public information p, E, A, B
2. Computes $V_1 = xB + sR$ and $V_2 = mA$
3. Declares the signature valid if $V_1 = V_2$

The verification procedure works because

$$V_1 = xB + sR = xaA + k^{-1}(m - ax)(kA) = xaA + (m - ax)A = mA = V_2.$$

There is a subtle point that should be mentioned. We have used k^{-1} in this verification equation as the integer mod N satisfying $k^{-1}k \equiv 1 \pmod{N}$. Therefore, $k^{-1}k$

is not 1, but rather an integer congruent to 1 mod N . So $k^{-1}k = 1 + tN$ for some integer t . It can be shown that $NA = \infty$. Therefore,

$$k^{-1}kA = (1 + tN)A = A + t(NA) = A + t\infty = A.$$

This shows that k^{-1} and k cancel each other in the verification equation, as we implicitly assumed above.

The classical ElGamal scheme and the present elliptic curve version are analogs of each other. The integers mod p are replaced with the elliptic curve E , and the number $p - 1$ becomes N . Note that the calculations in the classical scheme work with integers that are nonzero mod p , and there are $p - 1$ such congruence classes. The elliptic curve version works with points on the elliptic curve that are multiples of A , and the number of such points is a divisor of N .

The use of the x -coordinate of R in the elliptic version is somewhat arbitrary. Any method of assigning integers to points on the curve would work. Using the x -coordinate is an easy choice. Similarly, in the classical ElGamal scheme, the use of the integer r in the mod $p - 1$ equation for s might seem a little unnatural, since r was originally defined mod p . However, any method of assigning integers to the integers mod p would work (see [Exercise 16 in Chapter 13](#)). The use of r itself is an easy choice.

There is an elliptic curve version of the Digital Signature Algorithm that is similar to the preceding ([Exercise 24](#)).

21.6 Exercises

1. 1. Let $x^3 + ax^2 + bx + c$ be a cubic polynomial with roots r_1, r_2, r_3 . Show that $r_1 + r_2 + r_3 = -a$.

2. Write $x = x_1 - a/3$. Show that

$$x^3 + ax^2 + bx + c = x_1^3 + b'x_1 + c',$$

with $b' = b - (1/3)a^2$ and $c' = c - (1/3)ab + (2/27)a^3$. (Remark: This shows that a simple change of variables allows us to consider the case where the coefficient of x^2 is 0.)

2. Let E be the elliptic curve $y^2 \equiv x^3 - x + 4 \pmod{5}$.

1. List the points on E (don't forget ∞).
2. Evaluate the elliptic curve addition $(2, 0) + (4, 3)$.

3. 1. List the points on the elliptic curve E : $y^2 \equiv x^3 - 2 \pmod{7}$.

2. Find the sum $(3, 2) + (5, 5)$ on E .
3. Find the sum $(3, 2) + (3, 2)$ on E .

4. Let E be the elliptic curve $y^2 \equiv x^3 + x + 2 \pmod{13}$.

1. Evaluate $(1, 2) + (2, 5)$.
2. Evaluate $2(1, 2)$.
3. Evaluate $(1, 2) + \infty$.

5. 1. Find the sum of the points $(1, 2)$ and $(6, 3)$ on the elliptic curve $y^2 \equiv x^3 + 3 \pmod{7}$.

2. Eve tries to find the sum of the points $(1, 2)$ and $(6, 3)$ on the elliptic curve $y^2 \equiv x^3 + 3 \pmod{35}$. What information does she obtain?

6. Show that if $P = (x, 0)$ is a point on an elliptic curve, then $2P = \infty$.

7. Find an elliptic curve mod 101 such that $(43, 21)$ is a point on the curve.

8. The point $(3, 5)$ lies on the elliptic curve $y^2 = x^3 - 2$ defined over the rational numbers. use the addition law to find another point with positive rational coordinates that lies on this curve.
9. 1. Show that $Q = (2, 3)$ on $y^2 = x^3 + 1$ satisfies $6Q = \infty$. (Hint: Compute $3Q$, then use [Exercise 6](#).)
2. Your computations in (a) probably have shown that $2Q \neq \infty$ and $3Q \neq \infty$. Use this to show that the points $\infty, Q, 2Q, 3Q, 4Q, 5Q$ are distinct.
10. 1. Factor $n = 35$ by the elliptic curve method by using the elliptic curve $y^2 \equiv x^3 + 26$ and calculating 3 times the point $P = (10, 9)$.
2. Factor $n = 35$ by the elliptic curve method by using the elliptic curve $y^2 \equiv x^3 + 5x + 8$ and the point $P = (1, 28)$.
11. Suppose you want to factor a composite integer n by using the elliptic curve method. You start with the curve $y^2 = x^3 - 4x \pmod{n}$ and the point $(2, 0)$. Why will this not yield the factorization of n ?
12. Devise an analog of the procedure in [Exercise 11\(a\)](#) in [Chapter 10](#) that uses elliptic curves.
13. Let $p = 999983$. The elliptic curve $E : y^2 \equiv x^3 + 1 \pmod{p}$ has 999984 points. Suppose you are given points P and Q on E and are told that there is an integer k such that $Q = kP$.
1. Describe a birthday attack that is expected to find k .
 2. Describe how the Baby Step, Giant Step method (see [Section 10.2](#)) finds k .
14. Let P and Q be points on an elliptic curve E . Peggy claims that she knows an integer k such that $kP = Q$ and she wants to convince Victor that she knows k without giving Victor any information about k . They perform a zero-knowledge protocol. The first step is the following:
1. Peggy chooses a random integer r_1 and lets $r_2 = k - r_1$. She computes $X_1 = r_1P$ and $X_2 = r_2P$ and sends them to Victor.
- Give the remaining steps. Victor wants to be at least 99 % sure that Peggy knows k . (Technical note: You may regard r_1 and r_2 as numbers mod n , where $nP = \infty$. Without congruences, Victor obtains some information about the size of k .

Nontechnical note: The “Technical note” may be ignored when solving the problem.)

15. Find all values of $y \bmod 35$ such that $(1, y)$ is a point on the curve $y^2 \equiv x^3 + 3x + 12 \pmod{35}$.

16. Suppose n is a product of two large primes and let $E : y^2 \equiv x^3 + bx + c \pmod{n}$. Bob wants to find some points on E .

1. Bob tries choosing a random x , computing $x^3 + bx + c$, and finding the square root of this number mod n , when the square root exists. Why will this strategy probably fail if Bob does not know p and q ?
2. Suppose Bob knows p and q . Explain how Bob can use the method of part (a) successfully? (Hint: He needs to use the Chinese Remainder Theorem.)

17. Show that if P, Q, R are points on an elliptic curve, then

$$P + Q + R = \infty \Leftrightarrow P, Q, R \text{ are collinear.}$$

- 18.
1. Eve is trying to find an elliptic curve discrete log: She has points A and B on an elliptic curve E such that $B = kA$ for some k . There are approximately 10^{20} points on E , so assume that $1 \leq k \leq 10^{20}$. She makes two lists and looks for a match. The first list is jA for N randomly chosen values of j . The second is $B - \ell A$ for N randomly chosen values of ℓ . How big should N be so that there is a good chance of a match?
 2. Give a classical (that is, not elliptic curve) version of the procedure in part (a).

19. Let P be a point on the elliptic curve E mod a prime p .

1. Show that there are only finitely many points on E , so P has only finitely many distinct multiples.
2. Show that there are integers i, j with $i > j$ such that $iP = jP$. Conclude that $(i - j)P = \infty$.
3. The smallest positive integer k such that $kP = \infty$ is called the **order** of P . Let m be an integer such that $mP = \infty$. Show that k divides m . (Hint: Imitate the proof of Exercise 53(c, d) in Chapter 3.)
4. (for those who know some group theory) Use Lagrange’s theorem from group theory to show that the number of points on E is a multiple of the order of P . (Combined with Hasse’s theorem, this gives a way of finding the

number of points on E . See Computer Problems 1 and 4.)

20. Let P be a point on the elliptic curve E . Suppose you know a positive integer k such that $kP = \infty$. You want to prove (or disprove) that k is the order of P .

1. Show that if $(k/p)P = \infty$ for some prime factor p of k , then k is not the order of P .
2. Suppose $m|k$ and $1 \leq m < k$. Show that $m|(k/p)$ for some prime divisor p of k .
3. Suppose that $(k/p)P \neq \infty$ for each prime factor of k .
Use Exercise 11(c) to show that the order of P is k .
(Compare with Exercise 54 in Chapter 3. For an example, see Computer Problem 4.)

21. 1. Let $x = b_1b_2 \dots b_w$ be an integer written in binary. Let P be a point on the elliptic curve E . Perform the following procedure:

1. Start with $k = 1$ and $S_1 = \infty$.
2. If $b_k = 1$, let $R_k = S_k + P$. If $b_k = 0$, let $R_k = S_k$.
3. Let $S_{k+1} = 2R_k$.
4. If $k = w$, stop. If $k < w$, add 1 to k and go to step 2.

Show that $R_w = xP$. (Compare with Exercise 56(a) in Chapter 3.)

2. Let x be a positive integer and let P be a point on an elliptic curve. Show that the following procedure computes xP .

1. Start with $a = x$, $B = \infty$, $C = P$.
2. If a is even, let $a = a/2$, and let $B = B$, $C = 2C$.
3. If a is odd, let $a = a - 1$, and let $B = B + C$, $C = C$.
4. If $a \neq 0$, go to step 2.
5. Output B .

(Compare with Exercise 56(b) in Chapter 3.)

22. Let E be an elliptic curve mod n (where n is some integer) and let P and Q be points on E with $2P = Q$. The curve E and the point Q are public and are known to everyone. The point P is secret. Peggy wants to convince Victor that she knows P . They do the following procedure:

1. Peggy chooses a random point R_1 on E and lets $R_2 = P - R_1$.
 2. Peggy computes $H_1 = 2R_1$ and $H_2 = 2R_2$ and sends H_1, H_2 to Victor.
 3. Victor checks that $H_1 + H_2 = Q$.
 4. Victor makes a request and Peggy responds.
 5. Victor now does something else.
 6. They repeat steps 1 through 5 several times.
1. Describe what is done in steps 4 and 5.
 2. Give a classical (non-elliptic curve) version of this protocol that yields a zero-knowledge proof that Peggy knows a solution x to $x^2 \equiv s \pmod{n}$.

23. Let E be an elliptic curve mod a large prime, let N be the number of points on E , and let P and Q be points on E . Peggy claims to know an integer s such that $sP = Q$. She wants to prove this to Victor by the following procedure. Victor knows E , P , and Q , but he does not know s and should receive no information about s .

1. Peggy chooses a random integer $r_1 \pmod{N}$ and lets $r_2 \equiv s - r_1 \pmod{N}$. (Don't worry about why it's mod N . It's for technical reasons.)
 2. Peggy computes $Y_1 = r_1 P$ and $Y_2 = r_2 P$ and sends Y_1 and Y_2 to Victor.
 3. Victor checks something.
 4. Victor randomly chooses $i = 1$ or 2 and asks Peggy for r_i .
 5. Peggy sends r_i to Victor.
 6. Victor checks something.
 7. Step (7).
1. What does Victor check in step (3)?
 2. What does Victor check in step (6)?

3. What should step (7) be if Victor wants to be at least 99.9% sure that Peggy knows s ?
24. Here is an elliptic curve version of the Digital Signature Algorithm. Alice wants to sign a message m , which is an integer. She chooses a prime p and an elliptic curve $E \pmod{p}$. The number of points n on E is computed and a large prime factor q of n is found. A point $A (\neq \infty)$ is chosen such that $qA = \infty$. (In fact, n is not needed. Choose a point A' on E and find an integer n' with $n'A' = \infty$. There are ways of doing this, though it is not easy. Let q be a large prime factor of n' , if it exists, and let $A = (n'/q)A'$. Then $qA = \infty$.) It is assumed that the message satisfies $0 \leq m < q$. Alice chooses her secret integer a and computes $B = aA$. The public information is p, E, q, A, B . Alice does the following:
1. Chooses a random integer k with $1 \leq k < q$ and computes $R = kA = (x, y)$
 2. Computes $s \equiv k^{-1}(m + ax) \pmod{q}$
 3. Sends the signed message (m, R, s) to Bob
- Bob verifies the signature as follows:
1. Computes $u_1 \equiv s^{-1}m \pmod{q}$ and $u_2 \equiv s^{-1}x \pmod{q}$
 2. Computes $V = u_1A + u_2B$
 3. Declares the signature valid if $V = R$
 1. Show that the verification equation holds for a correctly signed message. Where is the fact that $qA = \infty$ used (see the “subtle point” mentioned in the ElGamal scheme in [Section 21.5](#))?
 2. Why does $k^{-1} \pmod{q}$ exist?
 3. If q is large, why is there very little chance that s^{-1} does not exist mod q ? How do we recognize the case when it doesn’t exist? (Of course, in this case, Alice should start over by choosing a new k .)
 4. How many computations “(large integer) \times (point on E)” are made in the verification process here? How many are made in the verification process for the elliptic ElGamal scheme described in the text? (Compare with the end of [Section 13.5](#).)
25. Let A and B be points on an elliptic curve and suppose $B = kA$ for some integer k . Suppose also that $2^nA = \infty$ for some integer

n , but $T = 2^{n-1}A \neq \infty$.

1. Show that if $k \equiv k' \pmod{2^n}$, then $B = k'A$.
Therefore, we may assume that $0 \leq k < 2^n$.
2. Let j be an integer. Show that $jT = \infty$ when j is even
and $jT \neq \infty$ when j is odd.
3. Write $k = x_0 + 2x_1 + 4x_2 + \dots + 2^{n-1}x_{n-1}$,
where each x_i is 0 or 1 (binary expansion of k). Show
that $x_0 = 0$ if and only if $2^{n-1}B = \infty$.
4. Suppose that for some $m < n$ we know
 x_0, \dots, x_{m-1} . Let
 $Q_m = B - (x_0 + \dots + 2^{m-1}x_{m-1})A$. Show that
 $2^{n-m-1}Q_m = \infty$ if and only if $x_m = 0$. This allows us
to find x_m . Continuing in this way, we obtain
 x_0, \dots, x_{n-1} , and therefore we can compute k . This
technique can be extended to the case where $sA = \infty$,
where s is an integer with only small prime factors. This
is the analog of the Pohlig-Hellman algorithm (see
[Section 10.2](#)).

21.7 Computer Problems

1. Let E be the elliptic curve $y^2 \equiv x^3 + 2x + 3 \pmod{19}$.
 1. Find the sum $(1, 5) + (9, 3)$.
 2. Find the sum $(9, 3) + (9, -3)$.
 3. Using the result of part (b), find the difference $(1, 5) - (9, 3)$.
 4. Find an integer k such that $k(1, 5) = (9, 3)$.
 5. Show that $(1, 5)$ has exactly 20 distinct multiples, including ∞ .
 6. Using (e) and [Exercise 19\(d\)](#), show that the number of points on E is a multiple of 20. Use Hasse's theorem to show that E has exactly 20 points.
2. You want to represent the message 12345 as a point (x, y) on the curve $y^2 \equiv x^3 + 7x + 11 \pmod{593899}$. Write $x = 12345_\underline{\hspace{2em}}$ and find a value of the missing last digit of x such that there is a point on the curve with this x -coordinate.
 1. Factor 3900353 using elliptic curves.
 2. Try to factor 3900353 using the $p - 1$ method of [Section 9.4](#). Using the knowledge of the prime factors obtained from part (a), explain why the $p - 1$ method does not work well for this problem.
4. Let $P = (2, 3)$ be a point on the elliptic curve $y^2 \equiv x^3 - 10x + 21 \pmod{557}$.
 1. Show that $189P = \infty$, but $63P \neq \infty$ and $27P \neq \infty$.
 2. Use [Exercise 20](#) to show that P has order 189.
 3. Use [Exercise 19\(d\)](#) and Hasse's theorem to show that the elliptic curve has 567 points.
5. Compute the difference $(5, 9) - (1, 1)$ on the elliptic curve $y^2 \equiv x^3 - 11x + 11 \pmod{593899}$. Note that the answer involves large integers, even though the original points have small coordinates.

Chapter 22 Pairing-Based Cryptography

As we pointed out in the previous chapter, elliptic curves have various advantages in cryptosystems based on discrete logs. However, as we'll see in this chapter, they also open up exciting new vistas for cryptographic applications. The existence of a bilinear pairing on certain elliptic curves is what makes this possible.

First, we'll describe one of these pairings. Then we'll give various applications, including identity-based encryption, digital signatures, and encrypted keyword searches.

22.1 Bilinear Pairings

Although most of this chapter could be done in the context of cyclic groups of prime order, the primary examples of pairings in cryptography are based on elliptic curves or closely related situations. Therefore, for concreteness, we use only the following situation.

Let p be a prime of the form $6q - 1$, where q is also prime. Let E be the elliptic curve $y^2 \equiv x^3 + 1 \pmod{p}$. We need the following facts about E .

1. There are exactly $p + 1 = 6q$ points on E .
2. There is a point $P_0 \neq \infty$ such that $qP_0 = \infty$. In fact, if we take a random point P , then, with very high probability, $6P \neq \infty$ and $6P$ is a multiple of P_0 .
3. There is a function \tilde{e} that maps pairs of points (aP_0, bP_0) to q th roots of unity for all integers a, b . It satisfies the **bilinearity property**

$$\tilde{e}(aP_0, bP_0) = \tilde{e}(P_0, P_0)^{ab}$$

for all a, b . This implies that

$$\tilde{e}(aP, bQ) = \tilde{e}(P, Q)^{ab}$$

for all points P, Q that are multiples of P_0 . (See [Exercise 2](#).)

4. If we are given two points P and Q that are multiples of P_0 , then $\tilde{e}(P, Q)$ can be computed quickly from the coordinates of P and Q .
5. $\tilde{e}(P_0, P_0) \neq 1$, so it is a nontrivial q th root of unity.

We also make the following assumption:

1. If we are given a random point $P \neq \infty$ on E and a random q th root of unity $\omega \neq 1$, it is computationally infeasible to find a point Q on E with $\tilde{e}(Q, P) = \omega$ and it is computationally infeasible to find Q' with $\tilde{e}(P, Q') = \omega$.

Remarks

Properties (1) and (2) are fairly easy to verify (see [Exercises 4](#) and [5](#)). The existence of \tilde{e} satisfying (3), (4), (5) is deep. In fact, \tilde{e} is a modification of what is known as the Weil pairing in the theory of elliptic curves. The usual Weil pairing e satisfies $e(P_0, P_0) = 1$, but the present version is modified using special properties of E to obtain (5). It is generally assumed that (A) is true if the prime p is large enough, but this is not known. See [Exercise 10](#).

The fact that $\tilde{e}(P, Q)$ can be computed quickly needs some more explanation. The two points P, Q satisfy $P = aP_0$ and $Q = bP_0$ for some a, b . However, to find a and b requires solving a discrete log problem, which could take a long time. Therefore, the obvious solution of choosing a random q th root of unity for $\tilde{e}(P_0, P_0)$ and then using the bilinearity property to define \tilde{e} does not work, since it cannot be computed quickly. Instead, $\tilde{e}(P, Q)$ is computed directly in terms of the coordinates of the points P, Q .

Although we will not need to know this, the q th roots of unity lie in the finite field with p^2 elements (see [Section 3.11](#)).

For more about the definition of \tilde{e} , see [Boneh-Franklin] or [Washington].

The curve E is an example of a **supersingular** elliptic curve, namely one where the number of points is congruent to $1 \pmod p$. (See [Exercise 4](#).) For a while, these curves were regarded as desirable for cryptographic purposes because computations can be done quickly on them. But then it was shown that the discrete logarithm problem for them is only slightly more difficult than the classical discrete logarithm mod p (see [Section 22.2](#)), so they fell out of favor (after all, they are slower

computationally than simple multiplication mod p , and they provide no security advantage). Because of the existence of the pairing \tilde{e} , they have become popular again.

22.2 The MOV Attack

Let E be the elliptic curve from [Section 22.1](#). Suppose $Q = kP_0$, where k is some integer and P_0 is the point from [Section 22.1](#). The Elliptic Curve Discrete Log Problem asks us to find k . Menezes, Okamoto, and Vanstone showed the following method of reducing this problem to the discrete log problem in the field with p^2 elements. Observe that

$$\tilde{e}(Q, P_0) = \tilde{e}(kP_0, P_0) = \tilde{e}(P_0, P_0)^k.$$

Therefore, solving the discrete log problem $\tilde{e}(Q, P_0) = \tilde{e}(P_0, P_0)^k$ for k yields the answer. Note that this latter discrete log problem is not on the elliptic curve, but instead in the finite field with p^2 elements. There are analogues of the index calculus for this situation, so usually this is an easier discrete log problem.

For a randomly chosen (not necessarily supersingular) elliptic curve, the method still works in theory. But the values of the pairing usually lie in a field much larger than the field with p^2 elements. This slows down the computations enough that the MOV attack is infeasible for most non-supersingular curves.

The MOV attack shows that cryptosystems based on the elliptic curve discrete log problem for supersingular curves gives no substantial advantage over the classical discrete log problem mod a prime. For this reason, supersingular curves were avoided for cryptographic purposes until these curves occurred in applications where pairings needed to be computed quickly, as in the next few sections.

22.3 Tripartite Diffie-Hellman

Alice, Bob, and Carlos want to agree on a common key (for a symmetric cryptosystem). All communications among them are public. If there were only two people, Diffie-Hellman could be used. A slight extension of this procedure works for three people:

1. Alice, Bob, and Carlos agree on a large prime p and a primitive root α .
2. Alice chooses a secret integer a , Bob chooses a secret integer b , and Carlos chooses a secret integer c .
3. Alice computes $A \equiv \alpha^a \pmod{p}$, Bob computes $B \equiv \alpha^b \pmod{p}$, and Carlos computes $C \equiv \alpha^c \pmod{p}$.
4. Alice sends A to Bob, Bob sends B to Carlos, and Carlos sends C to Alice.
5. Alice computes $A' \equiv C^a$, Bob computes $B' \equiv A^b$, and Carlos computes $C' \equiv B^c$.
6. Alice sends A' to Bob, Bob sends B' to Carlos, and Carlos sends C' to Alice.
7. Alice computes $A'' \equiv C'^a$, Bob computes $B'' \equiv A'^b$, and Carlos computes $C'' \equiv B'^c$. Note that $A'' = B'' = C'' \equiv \alpha^{abc} \pmod{p}$.
8. Alice, Bob, and Carlos use some agreed-upon method to obtain keys from A'' , B'' , C'' . For example, they could use some standard hash function and apply it to $\alpha^{abc} \pmod{p}$.

This protocol could also be used with p and α replaced by an elliptic curve E and a point P_0 , so Alice computes aP_0 , etc., and the final result is $abcP_0$.

In 2000, Joux showed how to use pairings to obtain a more efficient protocol, one in which there is only one round instead of two:

1. Alice, Bob, and Carlos choose a supersingular elliptic curve E and a point P_0 , as in Section 22.1.

2. Alice chooses a secret integer a , Bob chooses a secret integer b , and Carlos chooses a secret integer c .
3. Alice computes $A = aP_0$, Bob computes $B = bP_0$, and Carlos computes $C = cP_0$.
4. Alice makes A public, Bob makes B public, and Carlos makes C public.
5. Alice computes $\tilde{e}(B, C)^a$, Bob computes $\tilde{e}(A, C)^b$, and Carlos computes $\tilde{e}(A, B)^c$. Note that each person has computed $\tilde{e}(P_0, P_0)^{abc}$.
6. Alice, Bob, and Carlos use some agreed-upon method to obtain keys from $\tilde{e}(P_0, P_0)^{abc}$. For example, they could apply some standard hash function to this number.

The eavesdropper Eve sees E and the points

P_0, aP_0, bP_0, cP_0 and needs to compute $\tilde{e}(P_0, P_0)^{abc}$.

This computation is called the **Bilinear Diffie-**

Hellman Problem. It is not known how difficult it is.

However, if Eve can solve the Computational Diffie-

Hellman Problem (see [Section 10.4](#)), then she uses

E, P_0, aP_0, bP_0 to obtain abP_0 and computes

$\tilde{e}(abP_0, cP_0) = \tilde{e}(P_0, P_0)^{abc}$. Therefore, the Bilinear

Diffie-Hellman Problem is no harder than the

Computational Diffie-Hellman Problem.

Joux's result showed that pairings could be used in a constructive way in cryptography, rather than only in a destructive method such as the MOV attack, and this led to pairings being considered for applications such as those in the next few sections. It also meant that supersingular curves again became useful in cryptography, with the added requirement that when a curve mod p is used, the prime p must be chosen large enough that the classical discrete logarithm problem (solve $\alpha^x \equiv \beta \pmod{p}$ for x) is intractable.

22.4 Identity-Based Encryption

In most public key systems, when Alice wants to send a message to Bob, she looks up his public key in a directory and then encrypts her message. However, she needs some type of authentication – perhaps the directory has been modified by Eve, and the public key listed for Bob was actually created by Eve. Alice wants to avoid this situation. It was suggested by Shamir in 1984 that it would be nice to have an identity-based system, where Bob's public identification information (for example, his email address) serves as his public key. Such a system was finally designed in 2001 by Boneh and Franklin.

Of course, some type of authentication of each user is still needed. In the present system, this occurs in the initial setup of the system during the communications between the Trusted Authority and the User. In the following, we give the basic idea of the system. For more details and improvements, see [Boneh-Franklin].

We need two public hash functions:

1. H_1 maps arbitrary length binary strings to multiples of P_0 . A little care is needed in defining H_1 , since no one should be able, given a binary string b , to find k with $H_1(b) = kP_0$. See [Exercise 7](#).
2. H_2 maps q th roots of unity to binary strings of length n , where n is the length of the messages that will be sent. Since H_2 must be specified before the system is set up, this limits the lengths of the messages that can be sent. However, the message could be, for example, a DES key that is used to encrypt a much longer message, so this length requirement is not a severe restriction.

To set up the system we need a Trusted Authority. Let's call him Arthur. Arthur does the following.

1. He chooses, once and for all, a secret integer s . He computes $P_1 = sP_0$, which is made public.
2. For each User, Arthur finds the user's identification ID (written as a binary string) and computes

$$D_{\text{User}} = s H_1(ID).$$

Recall that $H_1(ID)$ is a point on E , so D_{User} is s times this point.

3. Arthur uses a secure channel to send D_{User} to the user, who keeps it secret. Arthur does not need to store D_{User} , so he discards it.

The system is now ready to operate, but first let's review what is known:

Public: E, p, P_0, P_1, H_1, H_2

Secret: s (known only to Arthur), D_{User} (one for each User; it is known only by that User)

Alice wants to send an email message m (of binary length n) to Bob, who is one of the Users. She knows Bob's address, which is `bob@computer.com`. This is his ID . Alice does the following.

1. She computes $g = \tilde{e}(H_1(\text{bob@computer.com}), P_1)$. This is a q th root of unity.
2. She chooses a random $r \not\equiv 0 \pmod{q}$ and computes

$$t = m \oplus H_2(g^r).$$

3. She sends Bob the ciphertext

$$c = (rP_0, t).$$

Note that rP_0 is a point on E , and t is a binary string of length n .

If Bob receives a pair (U, v) , where U is a point on E and v is a binary string of length n , then he does the following.

1. He computes $h = \tilde{e}(D_{\text{Bob}}, U)$, which is a q th root of unity.
2. He recovers the message as

$$m = v \oplus H_2(h).$$

Why does this yield the message? If the encryption is performed correctly, Bob receives $U = rP_0$ and $v = t = m \oplus H_2(g^r)$. Since $D_{\text{Bob}} = sH_1(\text{bob@computer.com})$,

$$h = \tilde{e}(D_{\text{Bob}}, rP_0) = \tilde{e}(H_1, P_0)^{sr} = \tilde{e}(H_1, sP_0)^r = g^r. \quad (22.1)$$

Therefore,

$$t \oplus H_2(h) = t \oplus H_2(g^r) = m \oplus H_2(g^r) \oplus H_2(g^r) = m,$$

as desired. Note that the main step is [Equation \(22.1\)](#), which removes the secret s from the D_{Bob} in the first argument of \tilde{e} and puts it on the P_0 in the second argument. This follows from the bilinearity property of the function \tilde{e} . Almost all of the cryptographic uses of pairings have a similar idea of moving from one masking of the secret to another. The pairing allows this to be done without knowing the value of s .

It is very important that s be kept secret. If Eve obtains s , then she can compute the points D_{User} for each user and read every email. Since $P_1 = sP_0$, the security of s is compromised if Eve can compute discrete logs on the elliptic curve. Moreover, the ciphertext contains rP_0 . If Eve can compute a discrete log and find r , then she can compute g^r and use this to find $H_2(g^r)$ and also m . Therefore, for the security of the system, it is vital that p be chosen large enough that discrete logs are computationally infeasible.

22.5 Signatures

22.5.1 BLS Signatures

Alice wants to sign a document m . In earlier chapters, we have seen how to do this with RSA and ElGamal signatures. The BLS method, due to Boneh, Lynn, and Schacham, uses pairings.

We use a supersingular elliptic curve E_0 and point P_0 , as in [Section 22.1](#). To set up the signature scheme, we'll need a public hash function H that maps arbitrary length binary strings to multiples of P_0 . A little care is needed in defining H , since no one should be able, given a binary string b , to find k with $H(b) = kP_0$. See [Exercise 7](#).

To set up the system, Alice chooses, once and for all, a secret integer a and computes $K_{\text{Alice}} = aP_0$, which is made public.

Alice's signature for the message m is $S = aH(m)$, which is a point on E .

To verify that (m, S) is a valid signed message, Bob checks

$$\tilde{e}(S, P_0) \stackrel{?}{=} \tilde{e}(H(m), K_{\text{Alice}}).$$

If this equation holds, Bob says that the signature is valid.

If Alice signs correctly,

$$\tilde{e}(S, P_0) = \tilde{e}(aH(m), P_0) = \tilde{e}(H(m), P_0)^a = \tilde{e}(H(m), aP_0) = \tilde{e}(H(m), K_{\text{Alice}}),$$

so the verification equation holds.

Suppose Eve wants to forge Alice's signature on a document m . The values of $H(m)$, K_{Alice} , and P_0 are then already determined, so the verification equation says that Eve needs to find S satisfying $\tilde{e}(S, P_0) = a$ known quantity. Assumption (A) from [Section 22.1](#) says that (we hope) this is computationally infeasible.

22.5.2 A Variation

The BLS signature scheme uses a hash function whose values are points on an elliptic curve. This might seem less natural than using a standard hash function with values that are binary strings (that is, numbers). The following method of Zhang, Safavi-Naini, and Susilo remedies this situation. Let H be a standard hash function such as SHA-3 or SHA-256 that maps binary strings of arbitrary length to binary strings of fixed length. Regard the output of H as an integer. Alice's key is the same as in BLS, namely, $K_{\text{Alice}} = aP_0$. But the signature is computed as

$$S = (H(m) + a)^{-1}P_0,$$

where $(H(m) + a)^{-1}$ is the modular multiplicative inverse of $(H(m) + a) \bmod q$ (where q is the order of P_0 , as in [Section 22.1](#)).

The verification equation for the signed message (m, S) is

$$\tilde{e}(H(m)P_0 + K_{\text{Alice}}, S) \stackrel{?}{=} \tilde{e}(P_0, P_0).$$

Since $H(m)P_0 + K_{\text{Alice}} = (H(m) + a)P_0$, the left side of the verification equation equals the right side when Alice signs correctly. Again, assumption (A) from [Section 22.1](#) says that it should be hard for Eve to forge Alice's signature.

22.5.3 Identity-Based Signatures

When Alice uses one of the above methods or uses RSA or ElGamal signatures to sign a document, and Bob wants to verify the signature, he looks up Alice's key on a web-page, for example, and uses it in the verification process. This means he must trust the web-page to be correct, not a fake one made by Eve. It would be preferable to use something closely associated with Alice such as her email address as the public key. This is of course the same problem that was solved in the previous section for encryption, and similar techniques work in the present situation.

The following method of Hess gives an identity-based signature scheme.

We use a supersingular elliptic curve E and point P_0 as in [Section 22.1](#). We need two public hash functions:

1. H_1 maps arbitrary length binary strings to multiples of P_0 . A little care is needed in defining H_1 , since no one should be able, given a binary string b , to find k with $H_1(b) = kP_0$. See [Exercise 7](#).
2. H_2 maps binary strings of arbitrary length to binary strings of fixed length; for example, H_2 can be a standard hash function such as SHA-3 or SHA-256.

To set up the system, we need a Trusted Authority. Let's call him Arthur. Arthur does the following.

1. He chooses, once and for all, a secret integer s . He computes $P_1 = sP_0$, which is made public.
2. For each User, Arthur finds the user's identification ID (written as a binary string) and computes

$$D_{\text{User}} = s H_1(ID).$$

Recall that $H_1(ID)$ is a point on E , so D_{User} is s times this point.

3. Arthur uses a secure channel to send D_{User} to the user, who keeps it secret. Arthur does not need to store D_{User} , so he discards it.

The system is now ready to operate, but first let's review what is known:

Public: E, p, P_0, P_1, H_1, H_2

Secret: s (known only to Arthur), D_{User} (one for each User; it is known only by that User)

To sign m , Alice does the following.

1. She chooses a random point $P \neq \infty$ on E .
2. She computes $r = \tilde{e}(P, P_0)$.
3. She computes $h = H_2(m || r)$.
4. She computes $U = hD_{\text{Alice}} + P$.
5. The signed message is (m, U, h) .

If Bob receives a triple (m, U, h) , where U is a point on E and h is a binary string, then he does the following.

1. He computes $v_1 = \tilde{e}(H_1(ID_{\text{Alice}}), P_1)^{-h}$, which is a q th root of unity.
2. He computes $v_2 = v_1 \cdot \tilde{e}(U, P_0)$.
3. If $h = H_2(m || v_2)$, then Bob says the signature is valid.

Let's suppose Alice signed correctly. Then, writing H_1 for $H_1(ID_{\text{Alice}})$, we have

$$\begin{aligned} v_1 &= \tilde{e}(H_1, P_1)^{-h} = \tilde{e}(H_1, P_0)^{-hs} = \tilde{e}(sH_1, P_0)^{-h} \\ &= \tilde{e}(D_{\text{Alice}}, P_0)^{-h} = \tilde{e}(-h D_{\text{Alice}}, P_0). \end{aligned}$$

Also,

$$\begin{aligned} v_2 &= v_1 \cdot \tilde{e}(U, P_0) = \tilde{e}(-h D_{\text{Alice}}, P_0) \cdot \tilde{e}(U, P_0) \\ &= \tilde{e}(-h D_{\text{Alice}} + U, P_0) \\ &= \tilde{e}(P, P_0) = r. \end{aligned}$$

Therefore,

$$H_2(m || v_2) = H_2(m || r) = h,$$

so the signature is declared valid.

Suppose Eve has a document m and she wants to forge Alice's signature so that (m, U, h) is a valid signature. She cannot choose h arbitrarily since Bob is going to compute v_2 and see whether $h = H_2(m \parallel v_2)$. Therefore, if H_2 is a good hash function, Eve's best strategy is to choose a value v'_2 and compute $h = H(m \parallel v'_2)$. Since H_2 is collision resistant and $H_2(m \parallel v'_2) = h = H_2(m \parallel v_2)$, this v'_2 should equal $v_2 = v_1 \cdot \tilde{e}(U, P_0)$. But v_1 is completely determined by Alice's ID and h . This means that in order to satisfy the verification equation, Eve must find U such that $\tilde{e}(U, P_0)$ equals a given quantity. Assumption (A) says this should be hard to do.

22.6 Keyword Search

Alice runs a large corporation. Various employees send in reports containing secret information. These reports need to be sorted and routed to various departments, depending on the subject of the message. Of course, each report could have a set of keywords attached, and the keywords could determine which departments receive the reports. But maybe the keywords are sensitive information, too. Therefore, the keywords need to be encrypted. But if the person who sorts the reports decrypts the keywords, then this person sees the sensitive keywords. It would be good to have the keywords encrypted in a way that an authorized person can search for a certain keyword in a message and determine either that the keyword is present or not, and receive no other information about other keywords. A solution to this problem was found by Boneh, Di Crescenzo, Ostrovsky, and Persiano.

Start with a supersingular elliptic curve E and point P_0 as in [Section 22.1](#). We need two public hash functions:

1. H_1 maps arbitrary length binary strings to multiples of P_0 . A little care is needed in defining H_1 , since no one should be able, given a binary string b , to find k with $H_1(b) = kP_0$. See [Exercise 7](#).
2. H_2 maps the q th roots of unity (in the finite field with p^2 elements) to binary strings of fixed length; for example, if the roots of unity are expressed in binary, H_2 can be a standard hash function.

Alice sets up the system as follows. Incoming reports (encrypted) will have attachments consisting of encrypted keywords. Each department will have a set of keyword searches that it is allowed to do. If it finds one of its allotted keywords, then it saves the report.

1. Alice chooses a secret integer a and computes $P_1 = aP_0$.

2. Alice computes

$$T_w = a H_1(w).$$

The point T_w is sent via secure channels to each department that is authorized to search for w .

When Daphne writes a report, she attaches the relevant encrypted keywords to the documents. These encrypted keywords are produced as follows:

1. Let w be a keyword. Daphne chooses a random integer $r \not\equiv 0 \pmod{q}$ and computes

$$t = \tilde{e}(H_1(w), rP_1).$$

2. The encryption of the keyword w is the pair

$$[rP_0, H_2(t)].$$

A searcher looks at the encrypted keywords $[A, b]$ attached to a document and checks

$$H_2(\tilde{e}(T_w, A)) \stackrel{?}{=} b.$$

If yes, then the searcher concludes that w is a keyword for that document. If no, then w is not a keyword for it.

Why does this work? If $[A, b]$ is the encrypted form of the keyword w , then

$$A = rP_0 \text{ and } t = \tilde{e}(H_1(w), rP_1)$$

for some r . Therefore,

$$\tilde{e}(T_w, A) = \tilde{e}(a H_1(w), rP_0) = \tilde{e}(H_1(w), rP_0)^a = \tilde{e}(H_1(w), rP_1) = t,$$

$$\text{so } H_2(\tilde{e}(T_w, A)) = H_2(t) = b.$$

Suppose conversely, that w' is another keyword. Is it possible that $[A, b]$ corresponds to both w and w' ? Since the same value of r could be used in the encryptions of w and w' , it is possible that A occurs in encryptions of both w and w' . However, we'll show that in this case the number b comes from at most one of w and w' . Since H_1 is collision resistant and $T_{w'} = a H_1(w')$ and

$T_w = a H_1(w)$, we expect that $T_{w'} \neq T_w$ and $\tilde{e}(T_{w'}, A) \neq \tilde{e}(T_w, A)$. (See [Exercise 1](#).) Since H_2 is collision resistant, we expect that

$$H_2(\tilde{e}(T_{w'}, A)) \neq H_2(\tilde{e}(T_w, A)).$$

Therefore, $[A, b]$ passes the verification equation for at most one keyword w .

Each time that Daphne encrypts the keyword, a different r should be used. Otherwise, the encryption of the keyword will be the same and this can lead to information being leaked. For example, someone could notice that certain keywords occur frequently and make some guesses as to their meanings.

There are many potential applications of this keyword search scheme. For example, suppose a medical researcher wants to find out how many patients at a certain hospital were treated for disease X in the previous year. For privacy reasons, the administration does not want the researcher to obtain any other information about the patients, for example, gender, race, age, and other diseases. The administration could give the researcher T_X . Then the researcher could search the encrypted medical records for keyword X without obtaining any information other than the presence or absence of X .

22.7 Exercises

1. Let E be the supersingular elliptic curve from [Section 22.1](#).

1. Let $P \neq \infty$ and $Q \neq \infty$ be multiples of P_0 . Show that $\tilde{e}(P, Q) \neq 1$. (Hint: Use the fact that $\tilde{e}(P_0, P_0) = \omega$ is a q th root of unity and that $\omega^x = 1$ if and only if $x \equiv 0 \pmod{q}$.)
2. Let $Q \neq \infty$ be a multiple of P_0 and let P_1, P_2 be multiples of P_0 . Show that if $\tilde{e}(P_1, Q) = \tilde{e}(P_2, Q)$, then $P_1 = P_2$.

2. Let E be the supersingular elliptic curve from [Section 22.1](#).

1. Show that

$$\tilde{e}(aP, bQ) = \tilde{e}(P, Q)^{ab}$$

for all points P, Q that are multiples of P_0 .

2. Show that

$$\tilde{e}(P + Q, R) = \tilde{e}(P, R) \tilde{e}(Q, R)$$

for all P, Q, R that are multiples of P_0 .

3. Let E be the supersingular elliptic curve from [Section 22.1](#).

Suppose you have points A, B on E that are multiples of P_0 and are not equal to ∞ . Let a and b be two secret integers. Suppose you are given the points aA and bB . Find a way to use \tilde{e} to decide whether or not $a \equiv b \pmod{q}$.

4. Let $p \equiv -1 \pmod{3}$ be prime.

1. Show that there exists d with $3d \equiv 1 \pmod{p-1}$.

2. Show that if $a^3 \equiv b \pmod{p}$ if and only if $a \equiv b^d \pmod{p}$. This shows that every integer mod p has a unique cube root.

3. Show that $y^2 \equiv x^3 + 1 \pmod{p}$ has exactly $p+1$ points (including the point ∞). (Hint: Apply part (b) to $y^2 - 1$.) (Remark: A curve mod p whose number of points is congruent to 1 mod p is called *supersingular*.)

5. (for those who know some group theory)

1. In the situation of [Exercise 4](#), suppose that $p = 6q - 1$ with q also prime. Show that there exists a point $P_0 \neq \infty$ such that $qP_0 = \infty$.
2. Let $Q = (2, 3)$, as in [Exercise 9](#) in [Chapter 21](#). Show that if $P \not\in \{\infty, Q, 2Q, 3Q, 4Q, 5Q\}$, then $6P \neq \infty$ and $6P$ is a multiple of P_0 . (For simplicity, assume that $q > 3$.)
6. Let H_0 be a hash function that takes a binary string of arbitrary length as input and then outputs an integer mod p . Let $p = 6q - 1$ be prime with q also prime. Show how to use H_0 to construct a hash function H_1 that takes a binary string of arbitrary length as input and outputs a point on the elliptic curve $y^2 \equiv x^3 + 1 \pmod{p}$ that is a multiple of the point P_0 of [Section 22.1](#). (Hint: Use the technique of [Exercise 4](#) to find y , then x . Then use [Exercise 5\(b\)](#).)
7.
 1. In the identity-based encryption system of [Section 22.4](#), suppose Eve can compute k such that $H_1(\text{bob@computer.edu}) = kP_0$. Show that Eve can compute g^r and therefore read Bob's messages.
 2. In the BLS signature scheme of [Section 22.5.1](#), suppose Eve can compute k such that $H(m) = kP_0$. Show that Eve can compute S such that (m, S) is a valid signed document.
 3. In the identity-based signature scheme of [Section 22.5.1](#), suppose Eve can compute k such that $H_1(ID_{\text{Alice}}) = kP_0$. Show that Eve can compute D_{Alice} and therefore forge Alice's signature on documents.
 4. In the keyword search scheme of [Section 22.6](#), suppose Eve can compute k such that $H_1(w) = kP_0$. Show that Eve can compute T_w and therefore find the occurrences of encrypted w on documents.
8. Let E and P_0 be as in [Section 22.1](#). Show that an analogue of the Decision Diffie-Hellman problem can be solved for E . Namely, if we are given aP_0, bP_0, cP_0 , show how we can decide whether $abP_0 = cP_0$.
9. Suppose you try to set up an identity-based cryptosystem as follows. Arthur chooses large primes p and q and forms $n = pq$, which is made public. For each User, he converts the User's identification ID to a number e_{User} by some public method and then computes d with $de_{\text{User}} \equiv 1 \pmod{\phi(n)}$. Arthur gives d to the User. The integer n is the same for all users. When Alice wants to send an email to Bob, she uses the public method to convert his email address to e_{Bob} and then uses this to encrypt messages with

RSA. Bob knows d , so he can decrypt. Explain why this system is not secure.

10. You are given a point $P \neq \infty$ on the curve E of [Section 22.1](#) and you are given a q th root of unity ω . Suppose you can solve discrete log problems for the q th roots of unity. That is, if $\alpha \neq 1$ and β are q th roots of unity, you can find k so that $\alpha^k = \beta$. Show how to find a point Q on E with $\tilde{e}(Q, P) = \omega$.

Chapter 23 Lattice Methods

Lattices have become an important tool for the cryptanalyst. In this chapter, we give a sampling of some of the techniques. In particular, we use lattice reduction techniques to attack RSA in certain cases. Also, we describe the NTRU public key system and show how it relates to lattices. For a more detailed survey of cryptographic applications of lattices, see [Nguyen-Stern].

23.1 Lattices

Let v_1, \dots, v_n be linearly independent vectors in n -dimensional real space \mathbf{R}^n . This means that every n -dimensional real vector v can be written in the form

$$v = a_1v_1 + \dots + a_nv_n$$

with real numbers a_1, \dots, a_n that are uniquely determined by v . The **lattice** generated by v_1, \dots, v_n is the set of vectors of the form

$$m_1v_1 + \dots + m_nv_n$$

where m_1, \dots, m_n are integers. The set $\{v_1, \dots, v_n\}$ is called a **basis** of the lattice. A lattice has infinitely many possible bases. For example, suppose $\{v_1, v_2\}$ is a basis of a lattice. Let k be an integer and let $w_1 = v_1 + kv_2$ and $w_2 = v_2$. Then $\{w_1, w_2\}$ is also a basis of the lattice: Any vector of the form $m_1v_1 + m_2v_2$ can be written as $m'_1w_1 + m'_2w_2$ with $m'_1 = m_1$ and $m'_2 = m_2 - km_1$, and similarly any integer linear combination of w_1 and w_2 can be written as an integer linear combination of v_1 and v_2 .

Example

Let $v_1 = (1, 0)$ and $v_2 = (0, 1)$. The lattice generated by v_1 and v_2 is the set of all pairs (x, y) with x, y integers. Another basis for this lattice is

$\{(1, 5), (0, 1)\}$. A third basis is $\{(5, 16), (6, 19)\}$.

More generally, if $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is a matrix with determinant ± 1 , then $\{(a, b), (c, d)\}$ is a basis of this lattice (Exercise 4).

The length of a vector $v = (x_1, \dots, x_n)$ is

$$\| v \| = (x_1^2 + \cdots + x_n^2)^{1/2}.$$

Many problems can be related to finding a shortest nonzero vector in a lattice. In general, the **shortest vector problem** is hard to solve, especially when the dimension of the lattice is large. In the following section, we give some methods that work well in small dimensions.

Example

A shortest vector in the lattice generated by

$$(31, 59) \text{ and } (37, 70)$$

is $(3, -1)$ (another shortest vector is $(-3, 1)$). How do we find this vector? This is the subject of the next section. For the moment, we verify that $(3, -1)$ is in the lattice by writing

$$(3, -1) = -19(31, 59) + 16(37, 70).$$

In fact, $\{(3, -1), (1, 4)\}$ is a basis of the lattice. For most purposes, this latter basis is much easier to work with than the original basis since the two vectors $(3, -1)$ and $(1, 4)$ are almost orthogonal (their dot product is $(3, -1) \cdot (1, 4) = -1$, which is small). In contrast, the two vectors of the original basis are nearly parallel and have a very large dot product. The methods of the next section show how to replace a basis of a lattice with a new basis whose vectors are almost orthogonal.

23.2 Lattice Reduction

23.2.1 Two-Dimensional Lattices

Let v_1, v_2 form the basis of a two-dimensional lattice. Our first goal is to replace this basis with what will be called a reduced basis.

If $\|v_1\| > \|v_2\|$, then swap v_1 and v_2 , so we may assume that $\|v_1\| \leq \|v_2\|$. Ideally, we would like to replace v_2 with a vector v_2^* perpendicular to v_1 . As in the Gram-Schmidt process from linear algebra, the vector

$$v_2^* = v_2 - \frac{v_1 \cdot v_2}{v_1 \cdot v_1} v_1$$

(23.1)

is perpendicular to v_1 . But this vector might not lie in the lattice. Instead, let t be the closest integer to $(v_1 \cdot v_2) / (v_1 \cdot v_1)$ (for definiteness, take 0 to be the closest integer to $\pm \frac{1}{2}$, and ± 1 to be closest to $\pm \frac{3}{2}$, etc.). Then we replace the basis $\{v_1, v_2\}$ with the basis

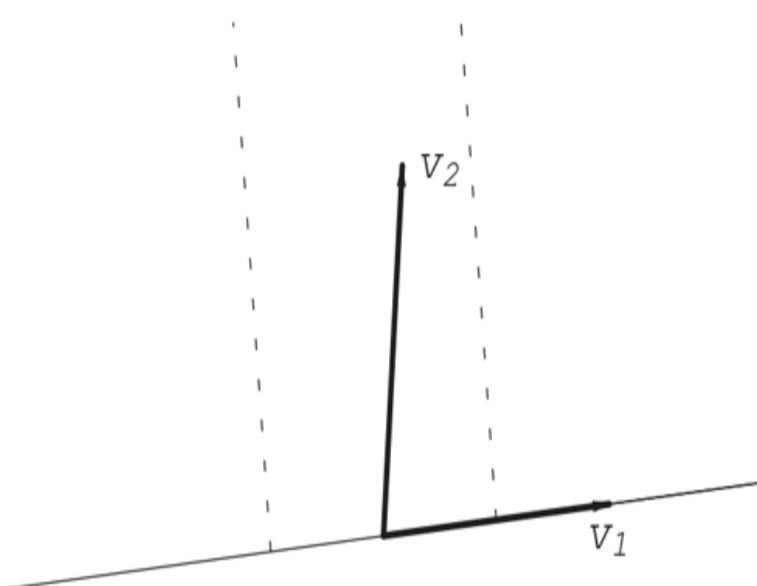
$$\{v_1, v_2 - tv_1\}.$$

We then repeat the process with this new basis.

We say that the basis $\{v_1, v_2\}$ is **reduced** if

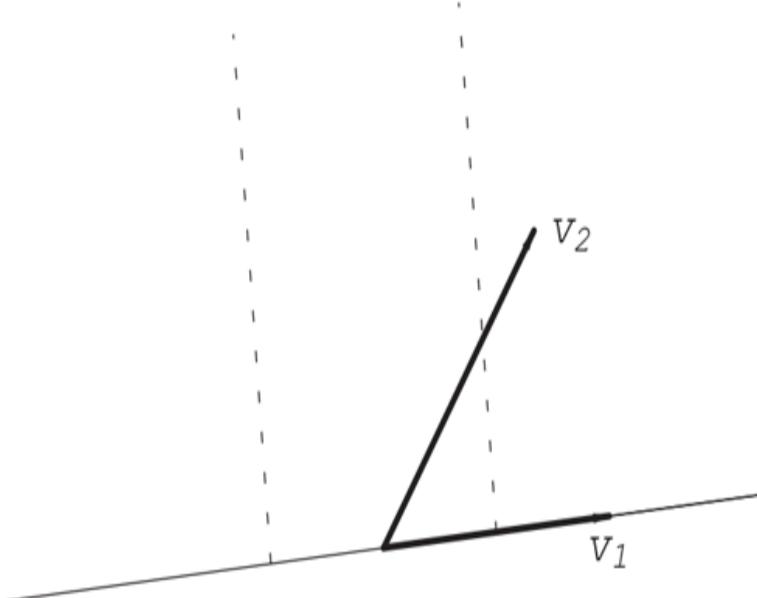
$$\|v_1\| \leq \|v_2\| \text{ and } -\frac{1}{2} \leq \frac{v_1 \cdot v_2}{v_1 \cdot v_1} \leq \frac{1}{2}.$$

The above reduction process stops exactly when we obtain a reduced basis, since this means that $t = 0$.



A reduced basis

23.2-1 Full Alternative Text



A nonreduced basis

23.2-2 Full Alternative Text

In the figures, the first basis is reduced because v_2 is longer than v_1 and the projection of v_2 onto v_1 is less than half the length of v_1 . The second basis is nonreduced because the projection of v_2 onto v_1 is too long. It is easy to see that a basis $\{v_1, v_2\}$ is reduced

when v_2 is at least as long as v_1 and v_2 lies within the dotted lines of the figures.

Example

Let's start with $v_1 = (31, 59)$ and $v_2 = (37, 70)$. We have $\|v_1\| < \|v_2\|$, so we do not swap the two vectors. Since

$$\frac{v_1 \cdot v_2}{v_1 \cdot v_1} = \frac{5277}{4442},$$

we take $t = 1$. The new basis is

$$v'_1 = v_1 = (31, 59) \quad \text{and} \quad v'_2 = v_2 - v_1 = (6, 11).$$

Swap v'_1 and v'_2 and rename the vectors to obtain a basis

$$v''_1 = (6, 11) \quad \text{and} \quad v''_2 = (31, 59).$$

We have

$$\frac{v''_1 \cdot v''_2}{v''_1 \cdot v''_1} = \frac{835}{157},$$

so we take $t = 5$. This yields vectors

$$(6, 11) \quad \text{and} \quad (1, 4) = (31, 59) - 5 \cdot (6, 11).$$

Swap these and name them $v_1^{(3)} = (1, 4)$ and $v_2^{(3)} = (6, 11)$. We have

$$\frac{v_1^{(3)} \cdot v_2^{(3)}}{v_1^{(3)} \cdot v_1^{(3)}} = \frac{50}{17},$$

so $t = 3$. This yields, after a swap,

$$v_1^r = (3, -1) \quad \text{and} \quad v_2^r = (1, 4).$$

Since $\|v_1^r\| \leq \|v_2^r\|$ and

$$\frac{v_1^r \cdot v_2^r}{v_1^r \cdot v_1^r} = -\frac{1}{10},$$

the basis $\{v_1^r, v_2^r\}$ is reduced.

A natural question is whether this process always produces a reduced basis. The answer is yes, as we prove in the following theorem. Moreover, the first vector in the reduced basis is a shortest vector for the lattice.

We summarize the discussion in the following.

Theorem

Let $\{v_1, v_2\}$ be a basis for a two-dimensional lattice in \mathbf{R}^2 . Perform the following algorithm:

1. If $\|v_1\| > \|v_2\|$, swap v_1 and v_2 so that $\|v_1\| \leq \|v_2\|$.
2. Let t be the closest integer to $(v_1 \cdot v_2) / (v_1 \cdot v_1)$.
3. If $t = 0$, stop. If $t \neq 0$, replace v_2 with $v_2 - tv_1$ and return to step 1.

The algorithm stops in finite time and yields a reduced basis $\{v_1^r, v_2^r\}$ of the lattice. The vector v_1^r is a shortest nonzero vector for the lattice.

Proof

First we prove that the algorithm eventually stops. As in [Equation 23.1](#), let $\mu = (v_1 \cdot v_2) / (v_1 \cdot v_1)$ and let $v_2^* = v_2 - \mu v_1$. Then

$$v_2 - tv_1 = v_2^* + (\mu - t)v_1.$$

Since v_1 and v_2^* are orthogonal, the Pythagorean theorem yields

$$\|v_2 - tv_1\|^2 = \|v_2^*\|^2 + \|(\mu - t)v_1\|^2 = \|v_2^*\|^2 + (\mu - t)^2 \|v_1\|^2.$$

Also, since $v_2 = v_2^* + \mu v_1$, and again since v_1 and v_2^* are orthogonal,

$$\|v_2\|^2 = \|v_2^*\|^2 + \mu^2 \|v_1\|^2.$$

Note that if $-1/2 \leq \mu \leq 1/2$ then $t = 0$ and $\mu - t = \mu$. Otherwise, $|\mu - t| \leq 1/2t|\mu|$. Therefore, if $t \neq 0$, we have $|\mu - t| < |\mu|$, which implies that

$$\|v_2 - tv_1\|^2 = \|v_2^*\|^2 + (\mu - t)^2 \|v_1\|^2 < \|v_2^*\|^2 + \mu^2 \|v_1\|^2 = \|v_2\|^2.$$

Therefore, if the process continues forever without yielding a reduced basis, then the lengths of the vectors decrease indefinitely. However, there are only finitely many vectors in the lattice that are shorter than the original basis vector v_2 . Therefore, the lengths cannot decrease forever, and a reduced basis must be found eventually.

To prove that the vector v_1 in a reduced basis is a shortest nonzero vector for the lattice, let $av_1 + bv_2$ be any nonzero vector in the lattice, where a and b are integers. Then

$$\|av_1 + bv_2\|^2 = (av_1 + bv_2) \cdot (av_1 + bv_2) = a^2 \|v_1\|^2 + b^2 \|v_2\|^2 + 2ab v_1 \cdot v_2.$$

Because $\{v_1, v_2\}$ is reduced,

$$-\frac{1}{2}v_1 \cdot v_1 \leq v_1 \cdot v_2 \leq \frac{1}{2}v_1 \cdot v_1,$$

which implies that $2ab v_1 \cdot v_2 \geq -|ab| \|v_1\|^2$.

Therefore,

$$\begin{aligned} \|av_1 + bv_2\|^2 &= (av_1 + bv_2) \cdot (av_1 + bv_2) \\ &= a^2 \|v_1\|^2 + 2ab v_1 \cdot v_2 + b^2 \|v_2\|^2 \\ &\geq a^2 \|v_1\|^2 - |ab| \|v_1\|^2 + b^2 \|v_2\|^2 \\ &\geq a^2 \|v_1\|^2 - |ab| \|v_1\|^2 + b^2 \|v_1\|^2, \end{aligned}$$

since $\|v_2\|^2 \geq \|v_1\|^2$ by assumption. Therefore,

$$\|av_1 + bv_2\|^2 \geq (a^2 - |ab| + b^2) \|v_1\|^2.$$

But $a^2 - |ab| + b^2$ is an integer. Writing it as $(|a| - \frac{1}{2}|b|)^2 + \frac{1}{4}|b|^2$, we see that it is nonnegative, and it equals 0 if and only if $a = b = 0$. Since $av_1 + bv_2 \neq 0$, we must have $a^2 - |ab| + b^2 \geq 1$. Therefore,

$$\| av_1 + bv_2 \| \geq \| v_1 \|,$$

so v_1 is a shortest nonzero vector.

23.2.2 The LLL algorithm

Lattice reduction in dimensions higher than two is much more difficult. One of the most successful algorithms was invented by A. Lenstra, H. Lenstra, and L. Lovász and is called the *LLL* algorithm. In many problems, a short vector is needed, and it is not necessary that the vector be the shortest. The *LLL* algorithm takes this approach and looks for short vectors that are almost as short as possible. This modified approach makes the algorithm run very quickly (in what is known as polynomial time). The algorithm performs calculations similar to those in the two-dimensional case, but the steps are more technical, so we omit details, which can be found in [Cohen], for example. The result is the following.

Theorem

Let L be the n -dimensional lattice generated by v_1, \dots, v_n in \mathbf{R}^n . Define the determinant of the lattice to be

$$D = |\det(v_1, \dots, v_n)|.$$

(This can be shown to be independent of the choice of basis. It is the volume of the parallelepiped spanned by v_1, \dots, v_n .) Let λ be the length of a shortest nonzero vector in L . The *LLL* algorithm produces a basis $\{b_1, \dots, b_n\}$ of L satisfying

1. $\| b_1 \| \leq 2^{(n-1)/4} D^{1/n}$
2. $\| b_1 \| \leq 2^{(n-1)/2} \lambda$
3. $\| b_1 \| \| b_2 \| \cdots \| b_n \| \leq 2^{n(n-1)/4} D$.

Statement (2) says that b_1 is close to being a shortest vector, at least when the dimension n is small. Statement (3) says that the new basis vectors are in some sense close to being orthogonal. More precisely, if the vectors b_1, \dots, b_n are orthogonal, then the volume D equals the product $\|b_1\| \|b_2\| \cdots \|b_n\|$. The fact that this product is no more than $2^{n(n-1)/4}$ times D says that the vectors are mostly close to orthogonal.

The running time of the *LLL* algorithm is less than a constant times $n^6 \log^3 B$, where n is the dimension and B is a bound on the lengths of the original basis vectors. In practice it is much faster than this bound. This estimate shows that the running time is quite good with respect to the size of the vectors, but potentially not efficient when the dimension gets large.

Example

Let's consider the lattice generated by (31, 59) and (37, 70), which we considered earlier when looking at the two-dimensional algorithm. The *LLL* algorithm yields the same result, namely $b_1 = (3, -1)$ and $b_2 = (1, 4)$. We have $D = 13$ and $\lambda = \sqrt{10}$ (given by $\|(3, -1)\|$, for example). The statements of the theorem are

1. $\|b_1\| = \sqrt{10} \leq 2^{1/4}\sqrt{13}$
2. $\|b_1\| = \sqrt{10} \leq 2^{1/2}\sqrt{10}$
3. $\|b_1\| \|b_2\| = \sqrt{10}\sqrt{17} \leq 2^{1/2}13$.

23.3 An Attack on RSA

Alice wants to send Bob a message of the form

*The answer is ***

or

*The password for your new account is *****.*

In these cases, the message is of the form

$$m = B + x, \text{ where } B \text{ is fixed and } |x| \leq Y$$

for some integer Y . We'll present an attack that works when the encryption exponent is small.

Suppose Bob has public RSA key $(n, e) = (n, 3)$. Then the ciphertext is

$$c \equiv (B + x)^3 \pmod{n}.$$

We assume that Eve knows B , Y , and n , so she only needs to find x . She forms the polynomial

$$\begin{aligned} f(T) &= (B + T)^3 - c = T^3 + 3BT^2 + 3B^2T + B^3 - c \\ &\equiv T^3 + a_2T^2 + a_1T + a_0 \pmod{n}. \end{aligned}$$

Eve is looking for $|x| \leq Y$ such that $f(x) \equiv 0 \pmod{n}$. In other words, she is looking for a small solution to a polynomial congruence $f(T) \equiv 0 \pmod{n}$.

Eve applies the *LLL* algorithm to the lattice generated by the vectors

$$\begin{aligned} v_1 &= (n, 0, 0, 0), & v_2 &= (0, Yn, 0, 0), & v_3 &= (0, 0, Y^2n, 0), \\ v_4 &= (a_0, a_1Y, a_2Y^2, Y^3). \end{aligned}$$

This yields a new basis b_1, \dots, b_4 , but we need only b_1 . The theorem in Subsection 23.2.2 tells us that

$$\| b_1 \| \leq 2^{3/4} \det(v_1, \dots, v_4)^{1/4}$$

(23.2)

$$= 2^{3/4}(n^3Y^6)^{1/4} = 2^{3/4}n^{3/4}Y^{3/2}.$$

(23.3)

We can write

$$b_1 = c_1v_1 + \cdots + c_4v_4 = (e_0, Ye_1, Y^2e_2, Y^3e_3)$$

with integers c_i and with

$$\begin{aligned} e_0 &= c_1n + c_4a_0 \\ e_1 &= c_2n + c_4a_1 \\ e_2 &= c_3n + c_4a_2 \\ e_3 &= c_4. \end{aligned}$$

It is easy to see that

$$e_i \equiv c_4a_i \pmod{n}, \quad 0 \leq i \leq 2.$$

Form the polynomial

$$g(T) = e_3T^3 + e_2T^2 + e_1T + e_0.$$

Then, since the integer x satisfies $f(x) \equiv 0 \pmod{n}$
and since the coefficients of $c_4f(T)$ and $g(T)$ are
congruent mod n ,

$$0 \equiv c_4f(x) \equiv g(x) \pmod{n}.$$

Assume now that

$$Y < 2^{-7/6}n^{1/6}.$$

(23.4)

Then

$$\begin{aligned} |g(x)| &\leq |e_0| + |e_1x| + |e_2x^2| + |e_3x^3| \\ &\leq |e_0| + |e_1|Y + |e_2|Y^2 + |e_3|Y^3 \\ &= (1, 1, 1, 1) \cdot (|e_0|, |e_1Y|, |e_2Y^2|, |e_3Y^3|) \\ &\leq \| (1, 1, 1, 1) \| ((|e_0|, \dots, |e_3Y^3|)) \\ &= 2 \| b_1 \|, \end{aligned}$$

where the last inequality used the Cauchy-Schwarz
inequality for dot products (that is, $v \cdot w \leq \|v\| \|w\|$). Since, by (17.3) and (17.4),

$$\| b_1 \| \leq 2^{3/4} n^{3/4} Y^{3/2} < 2^{3/4} n^{3/4} \left(2^{-7/6} n^{1/6} \right)^{3/2} = 2^{-1} n,$$

we obtain

$$|g(x)| < n.$$

Since $g(x) \equiv 0 \pmod{n}$, we must have $g(x) = 0$. The zeros of $g(T)$ may be determined numerically, and we obtain at most three candidates for x . Each of these may be tried to see if it gives the correct ciphertext. Therefore, Eve can find x .

Note that the above method replaces the problem of finding a solution to the congruence $f(T) \equiv 0 \pmod{n}$ with the exact, non-congruence, equation $g(T) = 0$.

Solving a congruence often requires factoring n , but solving exact equations can be done by numerical procedures such as Newton's method.

In exactly the same way, we can find small solutions (if they exist) to a polynomial congruence of degree d , using a lattice of dimension $d + 1$. Of course, d must be small enough that *LLL* will run in a reasonable time.

Improvements to this method exist. Coppersmith ([Coppersmith2]) gave an algorithm using higher-dimensional lattices that looks for small solutions x to a monic (that is, the highest-degree coefficient equals 1) polynomial equation $f(T) \equiv 0 \pmod{n}$ of degree d . If $|x| \leq n^{1/d}$, then the algorithm runs in time polynomial in $\log n$ and d .

Example

Let

$$n = 1927841055428697487157594258917$$

(which happens to be the product of the primes
 $p = 757285757575769$ and $q = 2545724696579693$

, but Eve does not know this). Alice is sending the message

*The answer is **,*

where ** denotes a two-digit number. Therefore the message is $m = B + x$ where

$B = 200805000114192305180009190000$ and
 $0 \leq x < 100$. Suppose Alice sends the ciphertext
 $c \equiv (B + x)^3 \equiv 30326308498619648559464058932 \pmod{n}$
. Eve forms the polynomial

$$f(T) = (B + T)^3 - c \equiv T^3 + a_2 T^2 + a_1 T + a_0 \pmod{n},$$

where

$$\begin{aligned} a_2 &= 602415000342576915540027570000 \\ a_1 &= 1123549124004247469362171467964 \\ a_0 &= 587324114445679876954457927616. \end{aligned}$$

Note that $a_0 \equiv B^3 - c \pmod{n}$.

Eve uses *LLL* to find a root of $f(T) \pmod{n}$. She lets $Y = 100$ and forms the vectors

$$\begin{aligned} v_1 &= (n, 0, 0, 0), \quad v_2 = (0, 100n, 0, 0), \quad v_3 = (0, 0, 10^4 n, 0) \\ v_4 &= (a_0, 100a_1, 10^4 a_2, 10^6). \end{aligned}$$

The *LLL* algorithm produces the vector

$$\begin{aligned} &308331465484476402v_1 + 589837092377839611v_2 \\ &+ 316253828707108264v_3 - 1012071602751202635v_4 \\ &= (246073430665887186108474, -577816087453534232385300, \\ &\quad 405848565585194400880000, -1012071602751202635000000). \end{aligned}$$

Eve then looks at the polynomial

$$\begin{aligned} g(T) &= -1012071602751202635T^3 + 40584856558519440088T^2 \\ &\quad - 5778160874535342323853T + 246073430665887186108474. \end{aligned}$$

The roots of $g(T)$ are computed numerically to be

$$42.000000000, -0.949612039 \pm 76.079608511i.$$

It is easily checked that $g(42) = 0$, so the plaintext is

The answer is 42.

Of course, a brute force search through all possibilities for the two-digit number x could have been used to find the answer in this case. However, if n is taken to be a 200-digit number, then Y can have around 33 digits. A brute force search would usually not succeed in this situation.

23.4 NTRU

If the dimension n is large, say $n \geq 100$, the *LLL* algorithm is not effective in finding short vectors. This allows lattices to be used in cryptographic constructions. Several cryptosystems based on lattices have been proposed. One of the most successful current systems is NTRU (rumored to stand for either “Number Theorists aRe Us” or “Number Theorists aRe Useful”). It is a public key system. In the following, we describe the algorithm for transmitting messages using a public key. There is also a related signature scheme, which we won’t discuss. Although the initial description of NTRU does not involve lattices, we’ll see later that it also has a lattice interpretation.

First, we need some preliminaries. Choose an integer N . We will work with the set of polynomials of degree less than N . Let

$$f = a_{N-1}X^{N-1} + \cdots + a_0 \quad \text{and} \quad g = b_{N-1}X^{N-1} + \cdots + b_0$$

be two such polynomials. Define

$$h = f * g = c_{N-1}X^{N-1} + \cdots + c_0,$$

where

$$c_i = \sum_{j+k=i} a_j b_k.$$

The summation is over all pairs j, k with $j + k \equiv i \pmod{N}$.

For example, let $N = 3$, let $f = X^2 + 7X + 9$, and let $g = 3X^2 + 2X + 5$. Then the coefficient c_1 of X in $f * g$ is

$$a_0b_1 + a_1b_0 + a_2b_2 = 9 \cdot 2 + 7 \cdot 5 + 1 \cdot 3 = 56,$$

and

$$f * g = 46X^2 + 56X + 68.$$

From a slightly more advanced viewpoint, $f * g$ is simply multiplication of polynomials mod $X^N - 1$ (see [Exercise 5](#) and [Section 3.11](#)).

NTRU works with certain sets of polynomials with small coefficients, so it is convenient to have a notation for them. Let

$$\begin{aligned} L(j, k) = & \text{ the set of polynomials of degree } < N \\ & \text{with } j \text{ coefficients equal to } +1 \\ & \text{and } k \text{ coefficients equal to } -1. \\ & \text{The remaining coefficients are 0.} \end{aligned}$$

We can now describe the NTRU algorithm. Alice wants to send a message to Bob, so Bob needs to set up his public key. He chooses three integers N, p, q with the requirements that $\gcd(p, q) = 1$ and that p is much smaller than q . Recommended choices are

$$(N, p, q) = (107, 3, 64)$$

for moderate security and

$$(N, p, q) = (503, 3, 256)$$

for very high security. Of course, these parameters will need to be adjusted as attacks improve. Bob then chooses two secret polynomials f and g with small coefficients (we'll say more about how to choose them later).

Moreover, f should be invertible mod p and mod q , which means that there exist polynomials F_p and F_q of degree less than N such that

$$F_p * f \equiv 1 \pmod{p}, \quad F_q * f \equiv 1 \pmod{q}.$$

Bob calculates

$$h \equiv F_q * g \pmod{q}.$$

Bob's public key is

$$(N, p, q, h).$$

His private key is f . Although F_p can be calculated easily from f , he should store (secretly) F_p since he will need it in the decryption process. What about g ? Since $g \equiv f * h \pmod{q}$, he is not losing information by not storing it (and he does not need it in decryption).

Alice can now send her message. She represents the message, by some prearranged procedure, as a polynomial m of degree less than N with coefficients of absolute value at most $(p - 1)/2$. When $p = 3$, this means that m has coefficients $-1, 0, 1$. Alice then chooses a small polynomial ϕ (“small” will be made more precise shortly) and computes

$$c \equiv p\phi * h + m \pmod{q}.$$

She sends the ciphertext c to Bob.

Bob decrypts by first computing

$$a \equiv f * c \pmod{q}$$

with all coefficients of the polynomial a of absolute value at most $q/2$, then (usually) recovering the message as

$$m \equiv F_p * a \pmod{p}.$$

Why should this work? In fact, sometimes it doesn't, but experiments with the parameter choices given below indicate that the probability of decryption errors is less than 5×10^{-5} . But here is why the decryption is usually correct. We have

$$\begin{aligned} a &\equiv f * c \equiv f * (p\phi * h + m) \\ &\equiv f * p\phi * F_q * g + f * m \\ &\equiv p\phi * g + f * m \pmod{q}. \end{aligned}$$

Since ϕ, g, f, m have small coefficients and p is much smaller than q , it is very probable that $p\phi * g + f * m$, before reducing mod q , has coefficients of absolute value less than $q/2$. In this case, we have equality

$$a = p\phi * g + f * m.$$

Then

$$F_p * a = pF_p * \phi * g + F_p * f * m \equiv 0 + 1 * m \equiv m \pmod{p},$$

so the decryption works.

For $(N, p, q) = (107, 3, 64)$, the recommended choices for f, g, ϕ are

$$f \in L(15, 14), \quad g \in L(12, 12), \quad \phi \in L(5, 5)$$

(recall that this means that the coefficients of f are fifteen 1s, fourteen -1 s, and the remaining 78 coefficients are 0).

For $(N, p, q) = (503, 3, 256)$, the recommended choices for f, g, ϕ are

$$f \in L(216, 215), \quad g \in L(72, 72), \quad \phi \in L(55, 55).$$

With these choices of parameters, the polynomials f, g, ϕ are small enough that the decryption works with very high probability.

The reason f has a different number of 1s and -1 s is so that $f(1) \neq 0$. It can be shown that if $f(1) = 0$, then f cannot be invertible.

Example

Let $(N, p, q) = (5, 3, 16)$ (this choice of N is much too small for any security; we use it only in order to give an explicit example). Take $f = X^4 + X - 1$ and $g = X^3 - X$. Since

$$(X^3 + X^2 - 1) * (X^4 + X - 1) \equiv 1 \pmod{3},$$

we have

$$F_p = X^3 + X^2 - 1.$$

Also,

$$\begin{aligned} F_q &= X^3 + X^2 - 1 \\ h &= -X^4 - 2X^3 + 2X + 1 \equiv F_q * g \pmod{16}. \end{aligned}$$

Bob's public key is

$$(N, p, q, h) = (5, 3, 16, -X^4 - 2X^3 + 2X + 1).$$

His private key is

$$f = X^4 + X - 1.$$

Alice takes her message to be $m = X^2 - X + 1$. She chooses $\phi = X - 1$. Then the ciphertext is

$$c \equiv 3\phi * h + m \equiv -3X^4 + 6X^3 + 7X^2 - 4X - 5 \pmod{16}.$$

Bob decrypts by first computing

$$a \equiv f * c \equiv 4X^4 - 2X^3 - 5X^2 + 6X - 2 \pmod{16},$$

then

$$F_p * a \equiv X^2 - X + 1 \pmod{3}.$$

Therefore, Bob has obtained the message.

23.4.1 An Attack on NTRU

Let $h = h_{N-1}X^{N-1} + \dots + h_0$. Form the $N \times N$ matrix

$$H = \begin{pmatrix} h_0 & h_1 & \cdots & h_{N-1} \\ h_{N-1} & h_0 & \cdots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_0 \end{pmatrix}.$$

If we represent $f = f_{N-1}X^{N-1} + \dots + f_0$ and $g = g_{N-1}X^{N-1} + \dots + g_0$ by the row vectors

$$\bar{f} = (f_0, \dots, f_{N-1}) \text{ and } \bar{g} = (g_0, \dots, g_{N-1}),$$

then we see that $\bar{f}H \equiv \bar{g} \pmod{q}$.

Let I be the $N \times N$ identity matrix. Form the $2N \times 2N$ matrix

$$M = \begin{pmatrix} I & H \\ 0 & qI \end{pmatrix}.$$

Let L be the lattice generated by the rows of M . Since $g \equiv f * h \pmod{q}$, we can write $g = f * h + qy$ for some polynomial y . Represent y as an N -dimensional row vector \bar{y} , so (\bar{f}, \bar{y}) is a $2N$ -dimensional row vector. Then

$$(\bar{f}, \bar{y}) M = (\bar{f}, \bar{g}),$$

so (\bar{f}, \bar{g}) is in the lattice L (see Exercise 3). Since f and g have small coefficients, (\bar{f}, \bar{g}) is a small vector in the lattice L . Therefore, the secret information for the key can be represented as a short vector in a lattice. An attacker can try to apply a lattice reduction algorithm to find short vectors, and possibly obtain (\bar{f}, \bar{g}) . Once the attacker has found f and g , the system is broken.

To stop lattice attacks, we need to make the lattice have high enough dimension that lattice reduction algorithms are inefficient. This is easily achieved by making N sufficiently large. However, if N is too large, the encryption and decryption algorithms become slow. The suggested values of N were chosen to achieve security while keeping the cryptographic algorithms efficient.

Lattice reduction methods have the best success when the shortest vector is small (more precisely, small when compared to the $2N$ th root of the determinant of the $2N$ -dimensional lattice). Improvements in the above lattice attack can be obtained by replacing I in the upper left block of M by αI for a suitably chosen real number α . This makes the resulting short vector $(\alpha \bar{f}, \bar{g})$ comparatively shorter and thus easier to find. The

parameters in NTRU, especially the sizes of f and g , have been chosen so as to limit the effect of these lattice attacks.

So far, the NTRU cryptosystem appears to be strong; however, as with many new cryptosystems, the security is still being studied. If no successful attacks are found, NTRU will have the advantage of providing security comparable to RSA and other public key methods, but with smaller key size and with faster encryption and decryption times.

23.5 Another Lattice-Based Cryptosystem

Another lattice-based public key cryptosystem was developed by Goldreich, Goldwasser, and Halevi in 1997. Let L be a 300-dimensional lattice that is a subset of the points with integral coordinates in 300-dimensional real space \mathbb{R}^{300} .

The private key is a “good” basis of L , given by the columns of a 300×300 matrix G , where “good” means that the entries of G are small.

The public key is a “bad basis” of L , given by the columns of a 300×300 matrix $B = GU$, where U is a secret 300×300 matrix with integral entries and with determinant 1. The determinant condition implies that the entries of U^{-1} are also integers. “Bad” means that B has many large entries.

A message is a 300-dimensional vector \vec{m} with integral entries, which is encrypted to obtain the ciphertext

$$\vec{c} = B\vec{m} + \vec{e},$$

where \vec{e} is a 300-dimensional vector whose entries are chosen randomly from $\{0, \pm 1, \pm 2, \pm 3\}$.

The decryption is carried out by computing

$$B^{-1}G[\vec{c}],$$

where $[\vec{v}]$ for a vector means we round off each entry to the nearest integer (and .5 goes whichever way you specify). Why does this decryption work? First, $U = G^{-1}B$, so

$$G^{-1}\vec{c} = G^{-1}B\vec{m} + G^{-1}\vec{e} = U\vec{m} + G^{-1}\vec{e}.$$

Since U and \vec{m} have integral entries, so does the $U\vec{m}$.

Since G is good, the entries of $G^{-1}\vec{e}$ tend to be small fractions, so they disappear in the rounding. Therefore, $\lfloor G^{-1}\vec{c} \rfloor$ probably equals $U\vec{m}$, so

$$B^{-1}G\lfloor G^{-1}\vec{c} \rfloor = B^{-1}GU\vec{m} = B^{-1}GG^{-1}B\vec{m} = \vec{m}.$$

Example

To keep things simple, we'll work in two-dimensional, rather than 300-dimensional, space. Let

$$G = \begin{pmatrix} 5 & 2 \\ 1 & 4 \end{pmatrix} \quad \text{and} \quad B = GU = \begin{pmatrix} 5 & 2 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} 17 & 18 \\ 16 & 17 \end{pmatrix} = \begin{pmatrix} 117 & 124 \\ 81 & 86 \end{pmatrix}$$

and

$$\vec{m} = \begin{pmatrix} 59 \\ 37 \end{pmatrix}.$$

Then

$$\vec{c} = \begin{pmatrix} 117 & 124 \\ 81 & 86 \end{pmatrix} \begin{pmatrix} 59 \\ 37 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 11492 \\ 7960 \end{pmatrix}.$$

To decrypt, we first compute

$$\lfloor G^{-1}\vec{c} \rfloor = \left\lfloor \begin{pmatrix} 2/9 & -1/9 \\ -1/18 & 5/18 \end{pmatrix} \begin{pmatrix} 11492 \\ 7960 \end{pmatrix} \right\rfloor = \left\lfloor \begin{pmatrix} 1669.33 \\ 1572.67 \end{pmatrix} \right\rfloor = \begin{pmatrix} 1669 \\ 1573 \end{pmatrix}.$$

Therefore, the decryption is

$$B^{-1}G\lfloor G^{-1}\vec{c} \rfloor = U^{-1}\lfloor G^{-1}\vec{c} \rfloor = \begin{pmatrix} 17 & -18 \\ -16 & 17 \end{pmatrix} \begin{pmatrix} 1669 \\ 1573 \end{pmatrix} = \begin{pmatrix} 59 \\ 37 \end{pmatrix} = \vec{m}.$$

Suppose, instead, that we tried to decrypt by computing $\lfloor B^{-1}\vec{c} \rfloor$? In the present example,

$$B^{-1} = \begin{pmatrix} 43/9 & -62/9 \\ -9/2 & 13/2 \end{pmatrix}$$

and

$$B^{-1}\vec{c} = \begin{pmatrix} 43/9 & -62/9 \\ -9/2 & 13/2 \end{pmatrix} \begin{pmatrix} 11492 \\ 7960 \end{pmatrix} = \begin{pmatrix} 212/3 \\ 26 \end{pmatrix}.$$

This rounds off to $(71, 26)$, which is nowhere close to the original message. The problem is that the entries of B^{-1} are much larger than those of G^{-1} , so the small error introduced by \vec{e} is amplified by B^{-1} .

Attacking this system involves the **Closest Vector**

Problem: Given a point P in \mathbf{R}^n , find the point in L closest to P .

We have $B\vec{m} \in L$, and \vec{c} is close to $B\vec{m}$ since it is moved off the lattice by the small vector \vec{e} .

For general lattices, the Closest Vector Problem is very hard. But it seems to be easier if the point is very close to a lattice point, which is the case in this cryptosystem. So the actual level of security is not yet clear.

23.6 Post-Quantum Cryptography?

If a quantum computer is built (see [Chapter 25](#)), cryptosystems based on factorization or discrete logs will become less secure. An active area of current research involves designing systems that cannot be broken by a quantum computer. Some of the most promising candidates seem to be lattice-based systems since their security does not depend on the difficulty of computing discrete logs or factoring, and no attack with a quantum computer has been found. Similarly, the McEliece cryptosystem, which is based on error-correcting codes (see [Section 24.10](#)) and is similar to the system in [Section 23.5](#), seems to be a possibility.

One of the potential difficulties with using many of these lattice-based systems is the key size: In the system in [Section 23.5](#), the public key requires integer entries. Many of the entries of A should be large, so let's say that we use 100 bits to specify each one. This means that the key requires $100n^2$ bits, much more than is used in current public key cryptosystems.

For more on this subject, see [Bernstein et al.].

23.7 Exercises

1. Find a reduced basis and a shortest nonzero vector in the lattice generated by the vectors $(58, 19)$, $(168, 55)$.
2.
 1. Find a reduced basis for the lattice generated by the vectors $(53, 88)$, $(107, 205)$.
 2. Find the vector in the lattice of part (a) that is closest to the vector $(151, 33)$. (Remark: This is an example of the **closest vector problem**. It is fairly easy to solve when a reduced basis is known, but difficult in general. For cryptosystems based on the closest vector problem, see [Nguyen-Stern].)
3. Let v_1, \dots, v_n be linearly independent row vectors in \mathbf{R}^n . Form the matrix M whose rows are the vectors v_i . Let $a = (a_1, \dots, a_n)$ be a row by v_1, \dots, v_n , and show that every vector in the lattice can be written in this way.
4. Let $\{v_1, v_2\}$ be a basis of a lattice. Let a, b, c, d be integers with $ad - bc = \pm 1$, and let

$$w_1 = av_1 + bv_2, \quad w_2 = cv_1 + dv_2.$$

1. Show that

$$v_1 = \pm(dw_1 - bw_2), \quad v_2 = \pm(-cw_1 + aw_2).$$

2. Show that $\{w_1, w_2\}$ is also a basis of the lattice.

5. Let N be a positive integer.

1. Show that if $j + k \equiv i \pmod{N}$, then $X^{j+k} - X^i$ is a multiple of $X^N - 1$.
2. Let $0 \leq i < N$. Let $a_0, \dots, a_{N-1}, b_0, \dots, b_{N-1}$ be integers and let

$$c_i = \sum_{j+k \equiv i} a_j b_k,$$

where the sum is over pairs j, k with $j + k \equiv i \pmod{N}$. Show that

$$c_i X^i - \sum_{j+k \equiv i} a_j b_k X^{j+k}$$

is a multiple of $X^N - 1$.

3. Let f and g be polynomials of degree less than N . Let fg be the usual product of f and g and let $f * g$ be defined as in [Section 23.4](#). Show that $fg - f * g$ is a multiple of $X^N - 1$.
6. Let N and p be positive integers. Suppose that there is a polynomial $F(X)$ such that $f(X) * F(X) \equiv 1 \pmod{p}$. Show that $f(1) \not\equiv 0 \pmod{p}$. (Hint: Use [Exercise 5\(c\)](#).)
7.
 1. In the NTRU cryptosystem, suppose we ignore q and let $c = p\phi * h + m$. Show how an attacker can obtain the message quickly.
 2. In the NTRU cryptosystem, suppose q is a multiple of p . Show how an attacker can obtain the message quickly.

Chapter 24 Error Correcting Codes

In a good cryptographic system, changing one bit in the ciphertext changes enough bits in the corresponding plaintext to make it unreadable. Therefore, we need a way of detecting and correcting errors that could occur when ciphertext is transmitted.

Many noncryptographic situations also require error correction; for example, fax machines, computer hard drives, CD players, and anything that works with digitally represented data. Error correcting codes solve this problem.

Though coding theory (communication over noisy channels) is technically not part of cryptology (communication over nonsecure channels), in Section [24.10](#) we describe how error correcting codes can be used to construct a public key cryptosystem.

24.1 Introduction

All communication channels contain some degree of noise, namely interference caused by various sources such as neighboring channels, electric impulses, deterioration of the equipment, etc. This noise can interfere with data transmission. Just as holding a conversation in a noisy room becomes more difficult as the noise becomes louder, so too does data transmission become more difficult as the communication channel becomes noisier. In order to hold a conversation in a loud room, you either raise your voice, or you are forced to repeat yourself. The second method is the one that will concern us; namely, we need to add some redundancy to the transmission in order for the recipient to be able to reconstruct the message. In the following, we give several examples of techniques that can be used. In each case, the symbols in the original message are replaced by *codewords* that have some redundancy built into them.

Example 1. (repetition codes)

Consider an alphabet $\{A, B, C, D\}$. We want to send a letter across a noisy channel that has a probability $p = 0.1$ of error. If we want to send C , for example, then there is a 90% chance that the symbol received is C . This leaves too large a chance of error. Instead, we repeat the symbol three times, thus sending CCC . Suppose an error occurs and the received word is CBC . We take the symbol that occurs most frequently as the message, namely C . The probability of the correct message being found is the probability that all three letters are correct plus the probability that exactly one of the three letters is wrong:

$$(0.9)^3 + 3(0.9)^2(0.1) = 0.972,$$

which leaves a significantly smaller chance of error.

Two of the most important concepts for codes are error detection and error correction. If there are at most two errors, this repetition code allows us to detect that errors have occurred. If the received message is *CBC*, then there could be either one error from *CCC* or two errors from *BBB*; we cannot tell which. If at most one error has occurred, then we can correct the error and deduce that the message was *CCC*. Note that if we used only two repetitions instead of three, we could detect the existence of one error, but we could not correct it (did *CB* come from *BB* or *CC*?).

This example was chosen to point out that error correcting codes can use arbitrary sets of symbols. Typically, however, the symbols that are used are mathematical objects such as integers mod a prime or binary strings. For example, we can replace the letters *A, B, C, D* by 2-bit strings: 00, 01, 10, 11. The preceding procedure (repeating three times) then gives us the codewords

000000, 010101, 101010, 111111.

Example 2. (parity check)

Suppose we want to send a message of seven bits. Add an eighth bit so that the number of nonzero bits is even. For example, the message 0110010 becomes 01100101, and the message 1100110 becomes 11001100. An error of one bit during transmission is immediately discovered since the message received will have an odd number of nonzero bits. However, it is impossible to tell which bit is incorrect, since an error in any bit could have yielded the odd number of nonzero bits. When an error is detected, the best thing to do is resend the message.

Example 3. (two-dimensional parity code)

The parity check code of the previous example can be used to design a code that can correct an error of one bit. The two-dimensional parity code arranges the data into a two-dimensional array, and then parity bits are computed along each row and column.

To demonstrate the code, suppose we want to encode the 20 data bits 10011011001100101011. We arrange the bits into a 4×5 matrix

$$\begin{array}{ccccc} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}$$

and calculate the parity bits along the rows and columns. We define the last bit in the lower right corner of the extended matrix by calculating the parity of the parity bits that were calculated along the columns. This results in the 5×6 matrix

$$\begin{array}{cccccc} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1. \end{array}$$

Suppose that this extended matrix of bits is transmitted and that a bit error occurs at the bit in the third row and fourth column. The receiver arranges the received bits into a 5×6 matrix and obtains

$$\begin{array}{cccccc} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1. \end{array}$$

The parities of the third row and fourth column are odd, so this locates the error as occurring at the third row and

fourth column.

If two errors occur, this code can detect their existence. For example, if bit errors occur at the second and third bits of the second row, then the parity checks of the second and third columns will indicate the existence of two bit errors. However, in this case it is not possible to correct the errors, since there are several possible locations for them. For example, if the second and third bits of the fifth row were incorrect instead, then the parity checks would be the same as when these errors occurred in the second row.

Example 4. (Hamming [7, 4] code)

The original message consists of blocks of four binary bits. These are replaced by codewords, which are blocks of seven bits, by multiplying (mod 2) on the right by the matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

For example, the message 1100 becomes

$$(1, 1, 0, 0) \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \equiv (1, 1, 0, 0, 0, 1, 1) \text{ (mod 2).}$$

Since the first four columns of G are the identity matrix, the first four entries of the output are the original message. The remaining three bits provide the redundancy that allows error detection and correction. In fact, as we'll see, we can easily correct an error if it affects only one of the seven bits in a codeword.

Suppose, for example, that the codeword 1100011 is sent but is received as 1100001 . How do we detect and correct the error? Write G in the form $[I_4, P]$, where P is a 4×3 matrix. Form the matrix $H = [P^T, I_3]$, where P^T is the transpose of P . Multiply the message received times the transpose of H :

$$(1, 1, 0, 0, 0, 0, 1) \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^T$$

$$\equiv (1, 1, 0, 0, 0, 0, 1) \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv (0, 1, 0) \pmod{2}.$$

This is the 6th row of H^T , which means there was an error in the 6th bit of the message received. Therefore, the correct codeword was 1100011 . The first four bits give the original message 1100 . If there had been no errors, the result of multiplying by H^T would have been $(0, 0, 0)$, so we would have recognized that no correction was needed. This rather mysterious procedure will be explained when we discuss Hamming codes in [Section 24.5](#). For the moment, note that it allows us to correct errors of one bit fairly efficiently.

The Hamming $[7, 4]$ code is a significant improvement over the repetition code. In the Hamming code, if we want to send four bits of information, we transmit seven bits. Up to two errors can be detected and up to one error can be corrected. For a repetition code to achieve this level of error detection and correction, we need to transmit 12 bits in order to send a 4-bit message. Later, we'll express this mathematically by saying that the code rate of this Hamming code is $4/7$, while the code rate of the repetition code is $4/12 = 1/3$. Generally, a higher code rate is better, as long as not too much error correcting capability is lost. For example, sending a 4-bit

message as itself has a code rate of 1 but is unsatisfactory in most situations since there is no error correction capability.

Example 5. (ISBN code)

The International Standard Book Number (ISBN) provides another example of an error detecting code. The ISBN is a 10-digit codeword that is assigned to each book when it is published. For example, the first edition of this book had ISBN number 0-13-061814-4. The first digit represents the language that is used; 0 indicates English. The next two digits represent the publisher. For example, 13 is associated with Pearson/Prentice Hall. The next six numbers correspond to a book identity number that is assigned by the publisher. The tenth digit is chosen so that the ISBN number $a_1a_2 \cdots a_{10}$ satisfies

$$\sum_{j=1}^{10} ja_j \equiv 0 \pmod{11}.$$

Notice that the equation is done modulo 11. The first nine numbers $a_1a_2 \cdots a_9$ are taken from $\{0, 1, \dots, 9\}$ but a_{10} may be 10, in which case it is represented by the symbol X . Books published in 2007 or later use a 13-digit ISBN, which uses a slightly different sum and works mod 10.

Suppose that the ISBN number $a_1a_2 \cdots a_{10}$ is sent over a noisy channel, or is written on a book order form, and is received as $x_1x_2 \cdots x_{10}$. The ISBN code can detect a single error, or a double error that occurs due to the transposition of the digits. To accomplish this, the receiver calculates the weighted checksum

$$S = \sum_{j=1}^{10} jx_j \pmod{11}.$$

If $S \equiv 0 \pmod{11}$, then we do not detect any errors, though there is a small chance that an error occurred and was undetected. Otherwise, we have detected an error. However, we cannot correct it (see [Exercise 2](#)).

If $x_1x_2 \cdots x_{10}$ is the same as $a_1a_2 \cdots a_{10}$ except in one place x_k , we may write $x_k = a_k + e$ where $e \neq 0$.

Calculating S gives

$$S = \sum_{j=1}^{10} ja_j + ke \equiv ke \pmod{11}.$$

Thus, if a single error occurs, we can detect it. The other type of error that can be *reliably* detected is when a_k and a_l have been transposed. This is one of the most common errors that occur when someone is copying numbers. In this case $x_l = a_k$ and $x_k = a_l$. Calculating S gives

$$\begin{aligned} S &= \sum_{j=1}^{10} jx_j = \sum_{j=1}^{10} ja_j + (k-l)a_l + (l-k)a_k \pmod{11} \\ &\equiv (k-l)(a_l - a_k) \pmod{11} \end{aligned}$$

If $a_l \neq a_k$, then the checksum is not equal to 0, and an error is detected.

Example 6. (Hadamard code)

This code was used by the *Mariner* spacecraft in 1969 as it sent pictures back to Earth. There are 64 codewords; 32 are represented by the rows of the 32×32 matrix

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & -1 & 1 & -1 & \cdots & -1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & -1 & -1 & 1 & \cdots & -1 \end{pmatrix}.$$

The matrix is constructed as follows. Number the rows and columns from 0 to 31. To obtain the entry h_{ij} in the

i th row and j th column, write $i = a_4a_3a_2a_1a_0$ and $j = b_4b_3b_2b_1b_0$ in binary. Then

$$h_{ij} = (-1)^{a_0b_0 + a_1b_1 + \dots + a_4b_4}.$$

For example, when $i = 31$ and $j = 3$, we have $i = 11111$ and $j = 00011$. Therefore,
 $h_{31, 3} = (-1)^2 = 1$.

The other 32 codewords are obtained by using the rows of $-H$. Note that the dot product of any two rows of H is 0, unless the two rows are equal, in which case the dot product is 32.

When *Mariner* sent a picture, each pixel had a darkness given by a 6-bit number. This was changed to one of the 64 codewords and transmitted. A received message (that is, a string of 1s and -1 s of length 32) can be decoded (that is, corrected to a codeword) as follows. Take the dot product of the message with each row of H . If the message is correct, it will have dot product 0 with all rows except one, and it will have dot product ± 32 with that row. If the dot product is 32, the codeword is that row of H . If it is -32 , the codeword is the corresponding row of $-H$. If the message has one error, the dot products will all be ± 2 , except for one, which will be ± 30 . This again gives the correct row of H or $-H$. If there are two errors, the dot products will all be 0, ± 2 , ± 4 , except for one, which will be ± 32 , ± 30 , or ± 28 . Continuing, we see that if there are seven errors, all the dot products will be between -14 and 14 , except for one between -30 and -16 or between 16 and 30 , which yields the correct codeword. With eight or more errors, the dot products start overlapping, so correction is not possible. However, detection is possible for up to 15 errors, since it takes 16 errors to change one codeword to another.

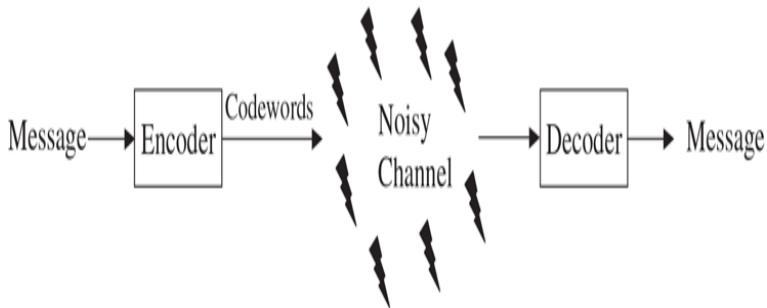
This code has a relatively low code rate of $6/32$, since it uses 32 bits to send a 6-bit message. However, this is

balanced by a high error correction rate. Since the messages from *Mariner* were fairly weak, the potential for errors was high, so high error correction capability was needed. The other option would have been to increase the strength of the signal and use a code with a higher code rate and less error correction. The transmission would have taken less time and therefore potentially have used less energy. However, in this case, it turned out that using a weaker signal more than offset the loss in speed. This issue (technically known as **coding gain**) is an important engineering consideration in the choice of which code to use in a given application.

24.2 Error Correcting Codes

A sender starts with a message and **encodes** it to obtain codewords consisting of sequences of symbols. These are transmitted over a noisy channel, depicted in [Figure 24.1](#), to the receiver. Often the sequences of symbols that are received contain errors and therefore might not be codewords. The receiver must **decode**, which means correct the errors in order to change what is received back to codewords and then recover the original message.

Figure 24.1 Encoding and Decoding



[Figure 24.1 Full Alternative Text](#)

The symbols used to construct the codewords belong to an alphabet. When the alphabet consists of the binary bits 0 and 1, the code is called a **binary code**. A code that uses sequences of three symbols, often represented as integers mod 3, is called a **ternary code**. In general, a code that uses an alphabet consisting of q symbols is called a **q -ary code**.

Definition

Let A be an alphabet and let A^n denote the set of n -tuples of elements of A . A **code of length n** is a nonempty subset of A^n .

The n -tuples that make up a code are called **codewords**, or code vectors. For example, in a binary repetition code where each symbol is repeated three times, the alphabet is the set $A = \{0, 1\}$ and the code is the set $\{(0, 0, 0), (1, 1, 1)\} \subset A^3$.

Strictly speaking, the codes in the definition are called **block codes**. Other codes exist where the codewords can have varying lengths. These will be mentioned briefly at the end of this chapter, but otherwise we focus only on block codes.

For a code that is a random subset of A^n , decoding could be a time-consuming procedure. Therefore, most useful codes are subsets of A^n satisfying additional conditions. The most common is to require that A be a finite field, so that A^n is a vector space, and require that the code be a subspace of this vector space. Such codes are called *linear* and will be discussed in [Section 24.4](#).

For the rest of this section, however, we work with arbitrary, possibly nonlinear, codes. We always assume that our codewords are n -dimensional vectors.

In order to decode, it will be useful to put a measure on how close two vectors are to each other. This is provided by the Hamming distance. Let u, v be two vectors in A^n . The **Hamming distance** $d(u, v)$ is the number of places where the two vectors differ. For example, if we use binary vectors and have the vectors $u = (1, 0, 1, 0, 1, 0, 1, 0)$ and $v = (1, 0, 1, 1, 1, 0, 0, 0)$, then u and v differ in two places (the 4th and the 7th) so $d(u, v) = 2$. As another example, suppose we are working with the usual English alphabet. Then $d(\text{fourth}, \text{eighth}) = 4$ since the two strings differ in four places.

The importance of the Hamming distance $d(u, v)$ is that it measures the minimum number of “errors” needed for u to be changed to v . The following gives some of its basic properties.

Proposition

$d(u, v)$ is a metric on A^n , which means that it satisfies

1. $d(u, v) \geq 0$, and $d(u, v) = 0$ if and only if $u = v$
2. $d(u, v) = d(v, u)$ for all u, v
3. $d(u, v) \leq d(u, w) + d(w, v)$ for all u, v, w .

The third property is often called the triangle inequality.

Proof. (1) $d(u, v) = 0$ is exactly the same as saying that u and v differ in no places, which means that $u = v$.

Part (2) is obvious. For part (3), observe that if u and v differ in a place, then either u and w differ at that place, or v and w differ at that place, or both. Therefore, the number of places where u and v differ is less than or equal to the number of places where u and w differ, plus the number of places where v and w differ.

For a code C , one can calculate the Hamming distance between any two distinct codewords. Out of this table of distances, there is a minimum value $d(C)$, which is called the **minimum distance** of C . In other words,

$$d(C) = \min \{d(u, v) | u, v \in C, u \neq v\}.$$

The minimum distance of C is very important number, since it gives the smallest number of errors needed to change one codeword into another.

When a codeword is transmitted over a noisy channel, errors are introduced into some of the entries of the vector. We correct these errors by finding the codeword whose Hamming distance from the received vector is as

small as possible. In other words, we change the received vector to a codeword by changing the fewest places possible. This is called **nearest neighbor decoding**.

We say that a code can **detect** up to s errors if changing a codeword in at most s places cannot change it to another codeword. The code can **correct** up to t errors if, whenever changes are made at t or fewer places in a codeword c , then the closest codeword is still c . This definition says nothing about an efficient algorithm for correcting the errors. It simply requires that nearest neighbor decoding gives the correct answer when there are at most t errors. An important result from the theory of error correcting codes is the following.

Theorem

1. A code C can detect up to s errors if $d(C) \geq s + 1$.
2. A code C can correct up to t errors if $d(C) \geq 2t + 1$.

Proof.

1. Suppose that $d(C) \geq s + 1$. If a codeword c is sent and s or fewer errors occur, then the received message r cannot be a different codeword. Hence, an error is detected.
2. Suppose that $d(C) \geq 2t + 1$. Assume that the codeword c is sent and the received word r has t or fewer errors; that is, $d(c, r) \leq t$. If c_1 is any other codeword besides c , we claim that $d(c_1, r) \geq t + 1$. To see this, suppose that $d(c_1, r) \leq t$. Then, by applying the triangle inequality, we have

$$2t + 1 \leq d(C) \leq d(c, c_1) \leq d(c, r) + d(c_1, r) \leq t + t = 2t.$$

This is a contradiction, so $d(c_1, r) \geq t + 1$. Since r has t or fewer errors, nearest neighbor decoding successfully decodes r to c .

How does one find the nearest neighbor? One way is to calculate the distance between the received message r and each of the codewords, then select the codeword with the smallest Hamming distance. In practice, this is impractical for large codes. In general, the problem of

decoding is challenging, and considerable research effort is devoted to looking for fast decoding algorithms. In later sections, we'll discuss a few decoding techniques that have been developed for special classes of codes.

Before continuing, it is convenient to introduce some notation.

Notation

A code of length n , with M codewords, and with minimum distance $d = d(C)$, is called an **(n, M, d) code**.

When we discuss linear codes, we'll have a similar notation, namely, an $[n, k, d]$ code. Note that this latter notation uses square brackets, while the present one uses curved parentheses. (These two similar notations cause less confusion than one might expect!) The relation is that an $[n, k, d]$ binary linear code is an $(n, 2^k, d)$ code.

The binary repetition code $\{(0, 0, 0), (1, 1, 1)\}$ is a $(3, 2, 3)$ code. The Hadamard code of [Exercise 6](#), [Section 24.1](#), is a $(32, 64, 16)$ code (it could correct up to 7 errors because $16 \geq 2 \cdot 7 + 1$).

If we have a q -ary (n, M, d) code, then we define the **code rate**, or **information rate**, R by

$$R = \frac{\log_q M}{n}.$$

For example, for the Hadamard code, $R = \log_2(64)/32 = 6/32$. The code rate represents the ratio of the number of input data symbols to the number of transmitted code symbols. It is an important parameter to consider when implementing real-world systems, as it represents what fraction of the bandwidth

is being used to transmit actual data. The code rate was mentioned in Examples 4 and 6 in [Section 24.1](#). A few limitations on the code rate will be discussed in [Section 24.3](#).

Given a code, it is possible to construct other codes that are essentially the same. Suppose that we have a codeword c that is expressed as $c = (c_1, c_1, \dots, c_n)$. Then we may define a positional permutation of c by permuting the order of the entries of c . For example, the new vector $c' = (c_2, c_3, c_1)$ is a positional permutation of $c = (c_1, c_2, c_3)$. Another type of operation that can be done is a symbol permutation. Suppose that we have a permutation of the q -ary symbols. Then we may fix a position and apply this permutation of symbols to that fixed position for every codeword. For example, suppose that we have the following permutation of the ternary symbols $\{0 \rightarrow 2, 1 \rightarrow 0, 2 \rightarrow 1\}$, and that we have the following codewords: $(0, 1, 2)$, $(0, 2, 1)$, and $(2, 0, 1)$. Then applying the permutation to the second position of all of the codewords gives the following vectors: $(0, 0, 2)$, $(0, 1, 1)$, and $(2, 2, 1)$.

Formally, we say that two codes are **equivalent** if one code can be obtained from the other by a series of the following operations:

1. Permuting the positions of the code
2. Permuting the symbols appearing in a fixed position of all codewords

It is easy to see that all codes equivalent to an (n, M, d) code are also (n, M, d) codes. However, for certain choices of n, M, d , there can be several inequivalent (n, M, d) codes.

24.3 Bounds on General Codes

We have shown that an (n, M, d) code can correct t errors if $d \geq 2t + 1$. Hence, we would like the minimum distance d to be large so that we can correct as many errors as possible. But we also would like for M to be large so that the code rate R will be as close to 1 as possible. This would allow us to use bandwidth efficiently when transmitting messages over noisy channels. Unfortunately, increasing d tends to increase n or decrease M .

In this section, we study the restrictions on n , M , and d without worrying about practical aspects such as whether the codes with good parameters have efficient decoding algorithms. It is still useful to have results such as the ones we'll discuss since they give us some idea of how good an actual code is, compared to the theoretical limits.

First, we treat upper bounds for M in terms of n and d . Then we show that there exist codes with M larger than certain lower bounds. Finally, we see how some of our examples compare with these bounds.

24.3.1 Upper Bounds

Our first result was given by R. Singleton in 1964 and is known as the **Singleton bound**.

Theorem

Let C be a q -ary (n, M, d) code. Then

$$M \leq q^{n-d+1}.$$

Proof. For a codeword $c = (a_1, \dots, a_n)$, let $c' = (a_d, \dots, a_n)$. If $c_1 \neq c_2$ are two codewords, then they differ in at least d places. Since c_1' and c_2' are obtained by removing $d - 1$ entries from c_1 and c_2 , they must differ in at least one place, so $c_1' \neq c_2'$. Therefore, the number M of codewords c equals the number of vectors c' obtained in this way. There are at most q^{n-d+1} vectors c' since there are $n - d + 1$ positions in these vectors. This implies that $M \leq q^{n-d+1}$, as desired.

Corollary

The code rate of a q -ary (n, M, d) code is at most $1 - \frac{d-1}{n}$.

Proof. The corollary follows immediately from the definition of code rate.

The corollary implies that if the **relative minimum distance** d/n is large, the code rate is forced to be small.

A code that satisfies the Singleton bound with equality is called an **MDS code** (maximum distance separable). The Singleton bound can be rewritten as $q^d \leq q^{n+1}/M$, so an MDS code has the largest possible value of d for a given n and M . The Reed-Solomon codes (Section 24.9) are an important class of MDS codes.

Before deriving another upper bound, we need to introduce a geometric interpretation that is useful in error correction. A **Hamming sphere** of radius t centered at a codeword c is denoted by $B(c, t)$ and is defined to be all vectors that are at most a Hamming

distance of t from the codeword c . That is, a vector u belongs to the Hamming sphere $B(c, t)$ if $d(c, u) \leq t$. We calculate the number of vectors in $B(c, t)$ in the following lemma.

Lemma

A sphere $B(c, r)$ in n -dimensional q -ary space has

$$\binom{n}{0} + \binom{n}{1}(q-1) + \binom{n}{2}(q-1)^2 + \cdots + \binom{n}{r}(q-1)^r$$

elements.

Proof. First we calculate the number of vectors that are a distance 1 from c . These vectors are the ones that differ from c in exactly one location. There are n possible locations and $q - 1$ ways to make an entry different. Thus the number of vectors that have a Hamming distance of 1 from c is $n(q - 1)$. Now let's calculate the number of vectors that have Hamming distance m from c . There are $\binom{n}{m}$ ways in which we can choose m locations to differ from the values of c . For each of these m locations, there are $q - 1$ choices for symbols different from the corresponding symbol from c . Hence, there are

$$\binom{n}{m}(q-1)^m$$

vectors that have a Hamming distance of m from c . Including the vector c itself, and using the identity $\binom{n}{0} = 1$, we get the result:

$$\binom{n}{0} + \binom{n}{1}(q-1) + \binom{n}{2}(q-1)^2 + \cdots + \binom{n}{r}(q-1)^r.$$

We may now state the **Hamming bound**, which is also called the **sphere packing bound**.

Theorem

Let C be a q -ary (n, M, d) code with $d \geq 2t + 1$. Then

$$M \leq \frac{q^n}{\sum_{j=0}^t \binom{n}{j} (q-1)^j}.$$

Proof. Around each codeword c we place a Hamming sphere of radius t . Since the minimum distance of the code is $d \geq 2t + 1$, these spheres do not overlap. The total number of vectors in all of the Hamming spheres cannot be greater than q^n . Thus, we get

$$\begin{aligned} & (\text{number of codewords}) \times (\text{number of elements per sphere}) \\ &= M \sum_{j=0}^t \binom{n}{j} (q-1)^j \leq q^n. \end{aligned}$$

This yields the desired inequality for M .

An (n, M, d) code with $d = 2t + 1$ that satisfies the Hamming bound with equality is called a **perfect code**. A perfect t -error correcting code is one such that the M Hamming spheres of radius t with centers at the codewords cover the entire space of q -ary n -tuples. The Hamming codes ([Section 24.5](#)) and the Golay code G_{23} ([Section 24.6](#)) are perfect. Other examples of perfect codes are the trivial $(n, q^n, 1)$ code obtained by taking all n -tuples, and the binary repetition codes of odd length ([Exercise 15](#)).

Perfect codes have been studied a lot, and they are interesting from many viewpoints. The complete list of perfect codes is now known. It includes the preceding examples, plus a ternary $[11, 6, 5]$ code constructed by Golay. We leave the reader a caveat. A name like *perfect codes* might lead one to assume that perfect codes are the best error correcting codes. This, however, is not true, as there are error correcting codes, such as Reed-Solomon codes, that are not perfect codes yet have better

error correcting capabilities for certain situations than perfect codes.

24.3.2 Lower Bounds

One of the problems central to the theory of error correcting codes is to find the largest code of a given length and given minimum distance d . This leads to the following definition.

Definition

Let the alphabet A have q elements. Given n and d with $d \leq n$, the largest M such that an (n, M, d) code exists is denoted $A_q(n, d)$.

We can always find at least one (n, M, d) code: Fix an element a_0 of A . Let C be the set of all vectors $(a, a, \dots, a, a_0, \dots, a_0)$ (with d copies of a and $n - d$ copies of a_0) with $a \in A$. There are q such vectors, and they are at distance d from each other, so we have an (n, q, d) code. This gives the trivial lower bound $A_q(n, d) \geq q$. We'll obtain much better bounds later.

It is easy to see that $A_q(n, 1) = q^n$: When a code has minimum distance $d = 1$, we can take the code to be all q -ary n -tuples. At the other extreme, $A_q(n, n) = q$ (Exercise 7).

The following lower bound, known as the **Gilbert-Varshamov bound**, was discovered in the 1950s.

Theorem

Given n, d with $n \geq d$, there exists a q -ary (n, M, d) code with

$$M \geq \frac{q^n}{\sum_{j=0}^{d-1} \binom{n}{j} (q-1)^j}.$$

This means that

$$A_q(n, d) \geq \frac{q^n}{\sum_{j=0}^{d-1} \binom{n}{j} (q-1)^j}.$$

Proof. Start with a vector c_1 and remove all vectors in A^n (where A is an alphabet with q symbols) that are in a Hamming sphere of radius $d - 1$ about that vector. Now choose another vector c_2 from those that remain. Since all vectors with distance at most $d - 1$ from c_1 have been removed, $d(c_2, c_1) \geq d$. Now remove all vectors that have distance at most $d - 1$ from c_2 , and choose c_3 from those that remain. We cannot have $d(c_3, c_1) \leq d - 1$ or $d(c_3, c_2) \leq d - 1$, since all vectors satisfying these inequalities have been removed. Therefore, $d(c_3, c_i) \geq d$ for $i = 1, 2$. Continuing in this way, choose c_4, c_5, \dots , until there are no more vectors.

The selection of a vector removes at most

$$\sum_{j=0}^{d-1} \binom{n}{j} (q-1)^j$$

vectors from the space. If we have chosen M vectors c_1, \dots, c_M , then we have removed at most

$$M \sum_{j=1}^{d-1} \binom{n}{j} (q-1)^j$$

vectors, by the preceding lemma. We can continue until all q^n vectors are removed, which means we can continue at least until

$$M \sum_{j=1}^{d-1} \binom{n}{j} (q-1)^j \geq q^n.$$

Therefore, there exists a code $\{c_1, \dots, c_M\}$ with M satisfying the preceding inequality.

Since $A_q(n, d)$ is the largest such M , it also satisfies the inequality.

There is one minor technicality that should be mentioned. We actually have constructed an (n, M, e) code with $e \geq d$. However, by modifying a few entries of c_2 if necessary, we can arrange that $d(c_2, c_1) = d$. The remaining vectors are then chosen by the above procedure. This produces a code where the minimal distance is exactly d .

If we want to send codewords with n bits over a noisy channel, and there is a probability p that any given bit will be corrupted, then we expect the number of errors to be approximately pn when n is large. Therefore, we need an (n, M, d) code with $d > 2pn$. We therefore need to consider (n, M, d) codes with $d/n \approx x > 0$, for some given $x > 0$. How does this affect M and the code rate?

Here is what happens. Fix q and choose x with $0 < x < 1 - 1/q$. The **asymptotic Gilbert-Varshamov bound** says that there is a sequence of q -ary (n, M, d) codes with $n \rightarrow \infty$ and $d/n \rightarrow x$ such that the code rate approaches a limit $\geq H_q(x)$, where

$$H_q(x) = 1 - x \log_q (q-1) + x \log_q (x) + (1-x) \log_q (1-x).$$

The graph of $H_2(x)$ is as in Figure 24.2. Of course, we would like to have codes with high error correction (that is, high x), and with high code rate ($= k/n$). The asymptotic result says that there are codes with error correction and code rate good enough to lie arbitrarily close to, or above, the graph.

Figure 24.2 The Graph of $H_2(x)$

Code Rate

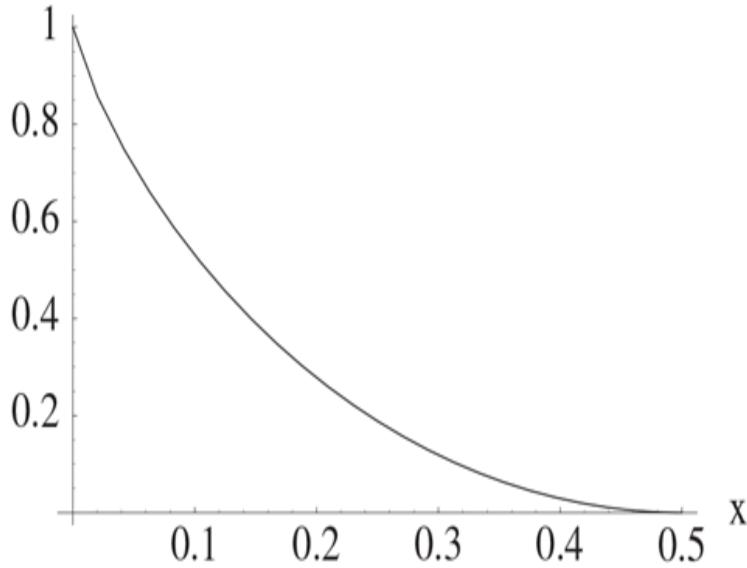


Figure 24.2 Full Alternative Text

The existence of certain sequences of codes having code rate limit strictly larger than $H_q(x)$ (for certain x and q) was proved in 1982 by Tsfasman, Vladut, and Zink using Goppa codes arising from algebraic geometry.

Examples

Consider the binary repetition code C of length 3 with the two vectors $(0, 0, 0)$ and $(1, 1, 1)$. It is a $(3, 2, 3)$ code. The Singleton bound says that $2 = M \leq 2$, so C is an MDS code. The Hamming bound says that

$$2 = M \leq \frac{2^3}{\binom{3}{0} + \binom{3}{1}} = 2,$$

so C is also perfect. The Gilbert-Varshamov bound says that there exists a $(3, M, 3)$ binary code with

$$M \geq \frac{2^3}{\binom{3}{0} + \binom{3}{1} + \binom{3}{2}} = \frac{8}{7},$$

which means $M \geq 2$.

The Hamming $[7, 4]$ code has $M = 16$ and $d = 3$, so it is a $(7, 16, 3)$ code. The Singleton bound says that $16 = M \leq 2^5$, so it is not an MDS code. The Hamming bound says that

$$16 = M \leq \frac{2^7}{\binom{7}{0} + \binom{7}{1}} = 16,$$

so the code is perfect. The Gilbert-Varshamov bound says that there exists a $(7, M, 3)$ code with

$$M \geq \frac{2^7}{\binom{7}{0} + \binom{7}{1} + \binom{7}{2}} = \frac{128}{29} \approx 4.4,$$

so the Hamming code is much better than this lower bound. Codes that have efficient error correction algorithms and also exceed the Gilbert-Varshamov bound are currently relatively rare.

The Hadamard code from [Section 24.1](#) is a binary (because there are two symbols) $(32, 64, 16)$ code. The Singleton bound says that $64 = M \leq 2^{17}$, so it is not very sharp in this case. The Hamming bound says that

$$64 = M \leq \frac{2^{32}}{\sum_{j=0}^7 \binom{32}{j}} \approx 951.3.$$

The Gilbert-Varshamov bound says there exists a binary $(32, M, 16)$ code with

$$M \geq \frac{2^{32}}{\sum_{j=0}^{15} \binom{32}{j}} \approx 2.3.$$

24.4 Linear Codes

When you are having a conversation with a friend over a cellular phone, your voice is turned into digital data that has an error correcting code applied to it before it is sent. When your friend receives the data, the errors in transmission must be accounted for by decoding the error correcting code. Only after decoding are the data turned into sound that represents your voice.

The amount of delay it takes for a packet of data to be decoded is critical in such an application. If it took several seconds, then the delay would become aggravating and make holding a conversation difficult.

The problem of efficiently decoding a code is therefore of critical importance. In order to decode quickly, it is helpful to have some structure in the code rather than taking the code to be a random subset of A^n . This is one of the primary reasons for studying linear codes. For the remainder of this chapter, we restrict our attention to linear codes.

Henceforth, the alphabet A will be a finite field \mathbf{F} . For an introduction to finite fields, see [Section 3.11](#). For much of what we do, the reader can assume that \mathbf{F} is $\mathbf{Z}_2 = \{0, 1\}$ = the integers mod 2, in which case we are working with binary vectors. Another concrete example of a finite field is \mathbf{Z}_p = the integers mod a prime p . For other examples, see [Section 3.11](#). In particular, as is pointed out there, \mathbf{F} must be one of the finite fields $GF(q)$; but the present notation is more compact. Since we are working with arbitrary finite fields, we'll use “=” instead of “ \equiv ” in our equations. If you want to think of \mathbf{F} as being \mathbf{Z}_2 , just replace all equalities between elements of \mathbf{F} with congruences mod 2.

The set of n -dimensional vectors with entries in \mathbf{F} is denoted by \mathbf{F}^n . They form a vector space over \mathbf{F} . Recall that a subspace of \mathbf{F}^n is a nonempty subset S that is closed under linear combinations, which means that if s_1, s_2 are in S and a_1, a_2 are in \mathbf{F} , then $a_1s_1 + a_2s_2 \in S$. By taking $a_1 = a_2 = 0$, for example, we see that $(0, 0, \dots, 0) \in S$.

Definition

A **linear code** of dimension k and length n over a field \mathbf{F} is a k -dimensional subspace of \mathbf{F}^n . Such a code is called an **$[n, k]$ code**. If the minimum distance of the code is d , then the code is called an **$[n, k, d]$ code**.

When $\mathbf{F} = \mathbf{Z}_2$, the definition can be given more simply. A binary code of length n and dimension k is a set of 2^k binary n -tuples (the codewords) such that the sum of any two codewords is always a codeword.

Many of the codes we have met are linear codes. For example, the binary repetition code $\{(0, 0, 0), (1, 1, 1)\}$ is a one-dimensional subspace of \mathbf{Z}_2^3 . The parity check code from [Exercise 2 in Section 24.1](#) is a linear code of dimension 7 and length 8. It consists of those binary vectors of length 8 such that the sum of the entries is 0 mod 2. It is not hard to show that the set of such vectors forms a subspace. The vectors

$$(1, 0, 0, 0, 0, 0, 0, 1), (0, 1, 0, 0, 0, 0, 0, 1), \dots, (0, 0, 0, 0, 0, 0, 1, 1)$$

form a basis of this subspace. Since there are seven basis vectors, the subspace is seven-dimensional.

The Hamming $[7, 4]$ code from [Example 4 of Section 24.1](#) is a linear code of dimension 4 and length 7. Every codeword is a linear combination of the four rows of the matrix G . Since these four rows span the code and are linearly independent, they form a basis.

The ISBN code ([Example 5 of Section 24.1](#)) is not linear. It consists of a set of 10-dimensional vectors with entries in \mathbf{Z}_{11} . However, it is not closed under linear combinations since X is not allowed as one of the first nine entries.

Let C be a linear code of dimension k over a field \mathbf{F} . If \mathbf{F} has q elements, then C has q^k elements. This may be seen as follows. There is a basis of C with k elements; call them v_1, \dots, v_k . Every element of C can be written uniquely in the form $a_1v_1 + \dots + a_kv_k$, with $a_1, \dots, a_k \in \mathbf{F}$. There are q choices for each a_i and there are k numbers a_i . This means there are q^k elements of C , as claimed. Therefore, an $[n, k, d]$ linear code is an (n, q^k, d) code in the notation of [Section 24.2](#).

For an arbitrary, possibly nonlinear, code, computing the minimum distance could require computing $d(u, v)$ for every pair of codewords. For a linear code, the computation is much easier. Define the **Hamming weight** $wt(u)$ of a vector u to be the number of nonzero places in u . It equals $d(u, 0)$, where 0 denotes the vector $(0, 0, \dots, 0)$.

Proposition

Let C be a linear code. Then $d(C)$ equals the smallest Hamming weight of all nonzero code vectors:

$$d(C) = \min \{wt(u) \mid 0 \neq u \in C\}.$$

Proof. Since $wt(u) = d(u, 0)$ is the distance between two codewords, we have $wt(u) \geq d(C)$ for all codewords u . It remains to show that there is a codeword with weight equal to $d(C)$. Note that $d(v, w) = wt(v - w)$ for any two vectors v, w . This is because an entry of $v - w$ is nonzero, and hence gets counted in $wt(v - w)$, if and only if v and w differ in

that entry. Choose v and w to be distinct codewords such that $d(v, w) = d(C)$. Then $wt(v - w) = d(C)$, so the minimum weight of the nonzero codewords equals $d(C)$.

To construct a linear $[n, k]$ code, we have to construct a k -dimensional subspace of \mathbf{F}^n . The easiest way to do this is to choose k linearly independent vectors and take their span. This can be done by choosing a $k \times n$ **generating matrix** G of rank k , with entries in \mathbf{F} . The set of vectors of the form vG , where v runs through all row vectors in \mathbf{F}^k , then gives the subspace.

For our purposes, we'll usually take $G = [I_k, P]$, where I_k is the $k \times k$ identity matrix and P is a $k \times (n - k)$ matrix. The rows of G are the basis for a k -dimensional subspace of the space of all vectors of length n . This subspace is our linear code C . In other words, every codeword is uniquely expressible as a linear combination of rows of G . If we use a matrix $G = [I_k, P]$ to construct a code, the first k columns determine the codewords. The remaining $n - k$ columns provide the redundancy.

The code in the second half of [Example 1, Section 24.1](#), has

$$G = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

The codewords 101010 and 010101 appear as rows in the matrix and the codeword 111111 is the sum of these two rows. This is a $[6, 2]$ code.

The code in [Example 2](#) has

$$G = \begin{matrix} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{matrix}.$$

For example, the codeword 11001001 is the sum mod 2 of the first, second, and fifth rows, and hence is obtained by multiplying $(1, 1, 0, 0, 1, 0, 0)$ times G . This is an $[8, 7]$ code.

In [Exercise 4](#), the matrix G is given in the description of the code. As you can guess from its name, it is a $[7, 4]$ code.

As mentioned previously, we could start with any $k \times n$ matrix of rank k . Its rows would generate an $[n, k]$ code. However, row and column operations can be used to transform the matrix to the form of G we are using, so we usually do not work with the more general situation. A code described by a matrix $G = [I_k, P]$ as before is said to be **systematic**. In this case, the first k bits are the **information symbols** and the last $n - k$ symbols are the **check symbols**.

Suppose we have $G = [I_k, P]$ as the generating matrix for a code C . Let

$$H = [-P^T, I_{n-k}],$$

where P^T is the transpose of P . In [Exercise 4](#) of [Section 24.1](#), this is the matrix that was used to correct errors. For [Exercise 2](#), we have $H = [1, 1, 1, 1, 1, 1, 1]$. Note that in this case a binary string v is a codeword if and only if the number of nonzero bits is even, which is the same as saying that its dot product with H is zero. This can be rewritten as $vH^T = 0$, where H^T is the transpose of H .

More generally, suppose we have a linear code $C \subset \mathbf{F}^n$. A matrix H is called a **parity check matrix** for C if H has the property that a vector $v \in \mathbf{F}^n$ is in C if and only if $vH^T = 0$. We have the following useful result.

Theorem

If $G = [I_k, P]$ is the generating matrix for a code C , then $H = [-P^T, I_{n-k}]$ is a parity check matrix for C .

Proof. Consider the i th row of G , which has the form

$$v_i = (0, \dots, 1, \dots, 0, p_{i,1}, \dots, p_{i,n-k}),$$

where the 1 is in the i th position. This is a vector of the code C . The j th column of H^T is the vector

$$(-p_{1,j}, \dots, -p_{n-k,j}, 0, \dots, 1, \dots, 0),$$

where the 1 is in the $(n - k + j)$ th position. To obtain the j th element of $v_i H^T$, take the dot product of these two vectors, which yields

$$1 \cdot (-p_{i,j}) + p_{i,j} \cdot 1 = 0.$$

Therefore, H^T annihilates every row v_i of G . Since every element of C is a sum of rows of G , we find that $vH^T = 0$ for all $v \in C$.

Recall the following fact from linear algebra: The left null space of an $m \times n$ matrix of rank r has dimension $n - r$. Since H^T contains I_{n-k} as a submatrix, it has rank $n - k$. Therefore, its left null space has dimension k . But we have just proved that C is contained in this null space. Since C also has dimension k , it must equal the null space, which is what the theorem claims.

We now have a way of detecting errors: If v is received during a transmission and $vH^T \neq 0$, then there is an error. If $vH^T = 0$, we cannot conclude that there is no error, but we do know that v is a codeword. Since it is more likely that no errors occurred than enough errors occurred to change one codeword into another codeword, the best guess is that an error did not occur.

We can also use a parity check matrix to make the task of decoding easier. First, let's look at an example.

Example

Let C be the binary linear code with generator matrix

$$G = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

We are going to make a table of all binary vectors of length 4 according to the following procedure. First, list the four elements of the code in the first row, starting with $(0, 0, 0, 0)$. Then, among the 12 remaining vectors, choose one of smallest weight (there might be several choices). Add this vector to the first row to obtain the second row. From the remaining eight vectors, again choose one with smallest weight and add it to the first row to obtain the third row. Finally, choose a vector with smallest weight from the remaining four vectors, add it to the first row, and obtain the fourth row. We obtain the following:

$$\begin{array}{cccc} (0, 0, 0, 0) & (1, 0, 1, 1) & (0, 1, 1, 0) & (1, 1, 0, 1) \\ (1, 0, 0, 0) & (0, 0, 1, 1) & (1, 1, 1, 0) & (0, 1, 0, 1) \\ (0, 1, 0, 0) & (1, 1, 1, 1) & (0, 0, 1, 0) & (1, 0, 0, 1) \\ (0, 0, 0, 1) & (1, 0, 1, 0) & (0, 1, 1, 1) & (1, 1, 0, 0) \end{array}$$

This can be used as a decoding table. When we receive a vector, find it in the table. Decode by changing the vector to the one at the top of its column. The error that is removed is first element of its row. For example, suppose we receive $(0, 1, 0, 1)$. It is the last element of the second row. Decode it to $(1, 1, 0, 1)$, which means removing the error $(1, 0, 0, 0)$. In this small example, this is not exactly the same as nearest neighbor decoding, since $(0, 0, 1, 0)$ decodes as $(0, 1, 1, 0)$ when it has an equally close neighbor $(0, 0, 0, 0)$. The problem is that the minimum distance of the code is 2, so general error correction is not possible. However, if we use a code that can correct up to t errors, this procedure correctly decodes all vectors that are distance at most t from a codeword.

In a large example, finding the vector in the table can be tedious. In fact, writing the table can be rather difficult (that's why we used such a small example). This is where a parity check matrix H comes to the rescue.

The first vector v in a row is called the **coset leader**. Let r be any vector in the same row as v . Then $r = v + c$ for some codeword c , since this is how the table was constructed. Therefore,

$$rH^T = vH^T + cH^T = vH^T,$$

since $cH^T = 0$ by the definition of a parity check matrix. The vector $S(r) = rH^T$ is called the **syndrome** of r . What we have shown is that two vectors in the same row have the same syndrome. Replace the preceding table with the following much smaller table.

Coset Leader	Syndrome
(0, 0, 0, 0)	(0, 0)
(1, 0, 0, 0)	(1, 1)
(0, 1, 0, 0)	(1, 0)
(0, 0, 0, 1)	(0, 1)

This table may be used for decoding as follows. For a received vector r , calculate its syndrome $S(r) = rH^T$. Find this syndrome on the list and subtract the corresponding coset leader from r . This gives the same decoding as above. For example, if $r = (0, 1, 0, 1)$, then

$$S(r) = (0, 1, 0, 1) \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = (1, 1).$$

This is the syndrome for the second row. Subtract the coset leader $(1, 0, 0, 0)$ from r to obtain the codeword $(1, 1, 0, 1)$.

We now consider the general situation. The method of the example leads us to two definitions.

Definition

Let C be a linear code and let u be an n -dimensional vector. The set $u + C$ given by

$$u + C = \{u + c \mid c \in C\}$$

is called a **coset** of C .

It is easy to see that if $v \in u + C$, then the sets $v + C$ and $u + C$ are the same (Exercise 9).

Definition

A vector having minimum Hamming weight in a coset is called a **coset leader**.

The **syndrome** of a vector u is defined to be $S(u) = uH^T$. The following lemma allows us to determine the cosets easily.

Lemma

Two vectors u and v belong to the same coset if and only if they have the same syndrome.

Proof. Two vectors u and v to belong to the same coset if and only if their difference belongs to the code C ; that is, $u - v \in C$. This happens if and only if

$(u - v)H^T = 0$, which is equivalent to
 $S(u) = uH^T = vH^T = S(v)$.

Decoding can be achieved by building a syndrome lookup table, which consists of the coset leaders and their corresponding syndromes. With a syndrome lookup table, we can decode with the following steps:

1. For a received vector r , calculate its syndrome $S(r) = rH^T$.
2. Next, find the coset leader with the same syndrome as $S(r)$. Call the coset leader c_0 .
3. Decode r as $r - c_0$.

Syndrome decoding requires significantly fewer steps than searching for the nearest codeword to a received vector. However, for large codes it is still too inefficient to be practical. In general, the problem of finding the nearest neighbor in a general linear code is hard; in fact, it is what is known as an NP-complete problem.

However, for certain special types of codes, efficient decoding is possible. We treat some examples in the next few sections.

24.4.1 Dual Codes

The vector space \mathbf{F}^n has a dot product, defined in the usual way:

$$(a_1, \dots, a_n) \cdot (b_0, \dots, b_n) = a_0b_0 + \dots + a_nb_n.$$

For example, if $\mathbf{F} = \mathbf{Z}_2$, then

$$(0, 1, 0, 1, 1, 1) \cdot (0, 1, 0, 1, 1, 1) = 0,$$

so we find the possibly surprising fact that the dot product of a nonzero vector with itself can sometimes be 0, in contrast to the situation with real numbers.

Therefore, the dot product does not tell us the length of a vector. But it is still a useful concept.

If C is a linear $[n, k]$ code, define the **dual code**

$$C^\perp = \{u \in \mathbf{F}^n \mid u \cdot c = 0 \text{ for all } c \in C\}.$$

Proposition

If C is a linear $[n, k]$ code with generating matrix $G = [I_k, P]$, then C^\perp is a linear $[n, n - k]$ code with generating matrix $H = [-P^T, I_{n-k}]$. Moreover, G is a parity check matrix for C^\perp .

Proof. Since every element of C is a linear combination of the rows of G , a vector u is in C^\perp if and only if $uG^T = 0$. This means that C^\perp is the left null space of G^T . Also, we see that G is a parity check matrix for C^\perp . Since G has rank k , so does G^T . The left null space of G^T therefore has dimension $n - k$, so C^\perp has dimension $n - k$. Because H is a parity check matrix for C , and the rows of G are in C , we have $GH^T = 0$.

Taking the transpose of this relation, and recalling that transpose reverses order ($(AB)^T = B^T A^T$), we find $HG^T = 0$. This means that the rows of H are in the left null space of G^T ; therefore, in C^\perp . Since H has rank $n - k$, the span of its rows has dimension $n - k$, which is the same as the dimension of C^\perp . It follows that the rows of H span C^\perp , so H is a generating matrix for C^\perp

A code C is called **self-dual** if $C = C^\perp$. The Golay code G_{24} of [Section 24.6](#) is an important example of a self-dual code.

Example

Let $C = \{(0, 0, 0), (1, 1, 1)\}$ be the binary repetition code. Since $u \cdot (0, 0, 0) = 0$ for every u , a vector u is in C^\perp if and only if $u \cdot (1, 1, 1) = 0$. This means that C^\perp

is a parity check code: $(a_1, a_2, a_3) \in C^\perp$ if and only if $a_1 + a_2 + a_3 = 0$.

Example

Let C be the binary code with generating matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

The proposition says that C^\perp has generating matrix

$$H = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

This is G with the rows switched, so the rows of G and the rows of H generate the same subspace. Therefore, $C = C^\perp$, which says that C is self-dual.

24.5 Hamming Codes

The Hamming codes are an important class of single error correcting codes that can easily encode and decode. They were originally used in controlling errors in long-distance telephone calls. Binary Hamming codes have the following parameters:

1. Code length: $n = 2^m - 1$
2. Dimension: $k = 2^m - m - 1$
3. Minimum distance: $d = 3$

The easiest way to describe a Hamming code is through its parity check matrix. For a binary Hamming code of length $n = 2^m - 1$, first construct an $m \times n$ matrix whose columns are all nonzero binary m -tuples. For example, for a $[7, 4]$ binary Hamming code we take $m = 3$, so $n = 7$ and $k = 4$, and start with

$$\begin{array}{ccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array} .$$

In order to obtain a parity check matrix for a code in systematic form, we move the appropriate columns to the end so that the matrix ends with the $m \times m$ identity matrix. The order of the other columns is irrelevant. The result is the parity check matrix H for a Hamming $[n, k]$ code. In our example, we move the 4th, 2nd, and 1st columns to the end to obtain

$$H = \begin{array}{ccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array} ,$$

which is the matrix H from [Exercise 3](#).

We can easily calculate a generator matrix G from the parity check matrix H . Since Hamming codes are single error correcting codes, the syndrome method for decoding can be simplified. In particular, the error vector e is allowed to have weight at most 1, and therefore will be zero or will have all zeros except for a single 1 in the j th position.

The Hamming decoding algorithm, which corrects up to one bit error, is as follows:

1. Compute the syndrome $s = yH^T$ for the received vector y . If $s = 0$, then there are no errors. Return the received vector and exit.
2. Otherwise, determine the position j of the column of H that is the transpose of the syndrome.
3. Change the j th bit in the received word, and output the resulting code.

As long as there is at most one bit error in the received vector, the result will be the codeword that was sent.

Example

The [15, 11] binary Hamming code has parity check matrix

$$\begin{matrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{matrix}.$$

Assume the received vector is

$$y = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1).$$

The syndrome $s = yH^T$ is calculated to be $s = (1, 1, 1, 1)$. Notice that s is the transpose of the 11th column of H , so we change the 11th bit of y to get the decoded word as

$(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1)$.

Since the first 11 bits give the information, the original message was

$(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$.

Therefore, we have detected and corrected the error.

24.6 Golay Codes

Two of the most famous binary codes are the Golay codes G_{23} and G_{24} . The $[24, 12, 8]$ extended Golay code G_{24} was used by the *Voyager I* and *Voyager II* space crafts during 1979–1981 to provide error correction for transmission back to Earth of color pictures of Jupiter and Saturn. The (nonextended) Golay code G_{23} , which is a $[23, 12, 7]$ code, is closely related to G_{24} . We shall construct G_{24} first, then modify it to obtain G_{23} . There are many other ways to construct the Golay codes. See [MacWilliams-Sloane].

The generating matrix for G_{24} is the 12×24 matrix

$G =$

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}.$$

All entries of G are integers mod 2. The first 12 columns of G are the 12×12 identity matrix. The last 11 columns are obtained as follows. The squares mod 11 are 0, 1, 3, 4, 5, 9 (for example, $4^2 \equiv 3$ and $7^2 \equiv 5$). Take the vector

$(x_0, \dots, x_{10}) = (1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0)$, with a 1 in positions 0, 1, 3, 4, 5, 9. This gives the last 11 entries in the first row of G . The last 11 elements of the other rows, except the last, are obtained by cyclically

permuting the entries in this vector. (Note: The entries are integers mod 2, not mod 11. The squares mod 11 are used only to determine which positions receive a 1.) The 13th column and the 12th row are included because they can be; they increase k and d and help give the code some of its nice properties. The basic properties of G_{24} are given in the following theorem.

Theorem

G_{24} is a self-dual [24, 12, 8] binary code. The weights of all vectors in G_{24} are multiples of 4.

Proof. The rows in G have length 24. Since the 12×12 identity matrix is contained in G , the 12 rows of G are linearly independent. Therefore, G_{24} has dimension 12, so it is a [24, 12, d] code for some d . The main work will be to show that $d = 8$. Along the way, we'll show that G_{24} is self-dual and that the weights of its codewords are 0 (mod 4).

Of course, it would be possible to have a computer list all $2^{12} = 4096$ elements of G_{24} and their weights. We would then verify the claims of the theorem. However, we prefer to give a more theoretical proof.

Let r_1 be the first row of G and let $r \neq r_1$ be any of the other first 11 rows. An easy check shows that r_1 and r have exactly four 1s in common, and each has four 1s that are matched with 0s in the other vector. In the sum $r_1 + r$, the four common 1s cancel mod 2, and the remaining four 1s from each row give a total of eight 1s in the sum. Therefore, $r_1 + r$ has weight 8. Also, the dot product $r_1 \cdot r$ receives contributions only from the common 1s, so

$$r_1 \cdot r = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 4 \equiv 0 \pmod{2}$$

.

Now let u and v be any two distinct rows of G , other than the last row. The first 12 entries and the last 11 entries of v are cyclic permutations of the corresponding parts of u and also of the corresponding parts of the first row. Since a permutation of the entries does not change the weights of vectors or the value of dot products, the preceding calculation of $r_1 + r$ and $r_1 \cdot r$ applies to u and v . Therefore,

1. $\text{wt}(u + v) = 8$
2. $u \cdot v \equiv 0 \pmod{2}$.

Any easy check shows that (1) and (2) also hold if u or v is the last row of G , so we see that (1) and (2) hold for any two distinct rows u, v of G . Also, each row of G has an even number of 1s, so (2) holds even when $u = v$.

Now let c_1 and c_2 be arbitrary elements of G_{24} . Then c_1 and c_2 are linear combinations of rows of G , so $c_1 \cdot c_2$ is a linear combination of numbers of the form $u \cdot v$ for various rows u and v of G . Each of these dot products is $0 \pmod{2}$, so $r_1 \cdot r_2 \equiv 0 \pmod{2}$. This implies that $C \subseteq C^\perp$. Since C is a 12-dimensional subspace of 24-dimensional space, C^\perp has dimension $24 - 12 = 12$. Therefore, C and C^\perp have the same dimension, and one is contained in the other. Therefore, $C = C^\perp$, which says that C is self-dual.

Observe that the weight of each row of G is a multiple of 4. The following lemma will be used to show that every element of G_{24} has a weight that is a multiple of 4.

Lemma

Let v_1 and v_2 be binary vectors of the same length. Then

$$\text{wt}(v_1 + v_2) = \text{wt}(v_1) + \text{wt}(v_2) - 2[v_1 \cdot v_2],$$

where the notation $[v_1 \cdot v_2]$ means that the dot product is regarded as a usual integer, not mod 2 (for example, $[(1, 0, 1, 1) \cdot (1, 1, 1, 1)] = 3$, rather than 1).

Proof. The nonzero entries of $v_1 + v_2$ occur when exactly one of the vectors v_1, v_2 has an entry 1 and the other has a 0 as its corresponding entry. When both vectors have a 1, these numbers add to 0 mod 2 in the sum. Note that $wt(v_1) + wt(v_2)$ counts the total number of 1s in v_1 and v_2 and therefore includes these 1s that canceled each other. The contributions to $[v_1 \cdot v_2]$ are caused exactly by these 1s that are common to the two vectors. So there are $[v_1 \cdot v_2]$ entries in v_1 and the same number in v_2 that are included in $wt(v_1) + wt(v_2)$, but do not contribute to $wt(v_1 + v_2)$. Putting everything together yields the equation in the lemma.

We now return to the proof of the theorem. Consider a

vector g in G_{24} . It can be written as a sum

$g \equiv u_1 + \cdots + u_k \pmod{2}$, where u_1, \dots, u_k are distinct rows of G . We'll prove that

$wt(g) \equiv 0 \pmod{4}$ by induction on k . Looking at G , we see that the weights of all rows of G are multiples of 4, so the case $k = 1$ is true. Suppose, by induction, that all vectors that can be expressed as a sum of $k - 1$ rows of G have weight $\equiv 0 \pmod{4}$. In particular,

$u = u_1 + \cdots + u_{k-1}$ has weight a multiple of 4. By the lemma,

$$wt(g) = wt(u + u_k) = wt(u) + wt(u_k) - 2[u \cdot u_k] \equiv 0 + 0 - 2[u \cdot u_k] \pmod{4}.$$

But $u \cdot u_k \equiv 0 \pmod{2}$, as we proved. Therefore,

$2[u \cdot u_k] \equiv 0 \pmod{4}$. We have proved that

$wt(g) \equiv 0 \pmod{4}$ whenever g is a sum of k rows. By induction, all sums of rows of G have weight $\equiv 0 \pmod{4}$. This proves that all weights of G_{24} are multiples of 4.

Finally, we prove that the minimum weight in G_{24} is 8. This is true for the rows of G , but we also must show it for sums of rows of G . Since the weights of codewords are multiples of 4, we must show that there is no codeword of weight 4, since the weights must then be at least 8. In fact, 8 is then the minimum, because the first row of G , for example, has weight 8.

We need the following lemma.

Lemma

The rows of the 12×12 matrix B formed from the last 12 columns of G are linearly independent mod 2. The rows of the 11×11 matrix A formed from the last 11 elements of the first 11 rows of G are linearly dependent mod 2. The only linear dependence relation is that the sum of all 11 rows of A is 0 mod 2.

Proof. Since G_{24} is self-dual, the dot product of any two rows of G is 0. This means that the matrix product $GG^T = 0$. Since $G = [I|B]$ (that is, I followed by the matrix B), this may be rewritten as

$$I^2 + B B^T = 0,$$

which implies that $B^{-1} = B^T$ (we're working mod 2, so the minus signs disappear). This means that B is invertible, so the rows are linearly independent.

The sum of the rows of A is 0 mod 2, so this is a dependence relation. Let $v_1 = (1, \dots, 1)^T$ be an 11-dimensional column vector. Then $Av_1 = 0$, which is just another way of saying that the sum of the rows is 0. Suppose v_2 is a nonzero 11-dimensional column vector such that $Av_2 = 0$. Extend v_1 and v_2 to 12-dimensional vectors v'_1, v'_2 by adjoining a 0 at the top of each column vector. Let r_{12} be the bottom row of B . Then

$$Bv_i' = (0, \dots, 0, r_{12} \cdot v_i')^T.$$

This equation follows from the fact that $Av_i = 0$. Note that multiplying a matrix times a vector consists of taking the dot products of the rows of the matrix with the vector.

Since B is invertible and $v_i' \neq 0$, we have $Bv_i' \neq 0$, so $r_{12} \cdot v_i' \neq 0$. Since we are working mod 2, the dot product must equal 1. Therefore,

$$B(v_1' + v_2') = (0, \dots, 0, r_1 \cdot v_1' + r_1 \cdot v_2')^T = (0, \dots, 0, 1+1)^T = 0.$$

Since B is invertible, we must have $v_1' + v_2' = 0$, so $v_1' = v_2'$ (we are working mod 2). Ignoring the top entries in v_1' and v_2' , we obtain $v_2 = (1, \dots, 1)$. Therefore, the only nonzero vector in the null space of A is v_1 . Since the vectors in the null space of a matrix give the linear dependencies among the rows of the matrix, we conclude that the only dependency among the rows of A is that the sum of the rows is 0. This proves the lemma.

Suppose g is a codeword in G_{24} . If g is, for example, the sum of the second, third, and seventh rows, then g will have 1s in the second, third, and seventh positions, because the first 12 columns of G form an identity matrix. In this way, we see that if g is the sum of k rows of G , then $wt(g) \geq k$. Suppose now that $wt(g) = 4$. Then g is the sum of at most four rows of G . Clearly, g cannot be a single row of G , since each row has weight at least 8. If g is the sum of two rows, we proved that $wt(g)$ is 8. If $g = r_1 + r_2 + r_3$ is the sum of three rows of G , then there are two possibilities.

(1) First, suppose that the last row of G is not one of the rows in the sum. Then three 1s are used from the 13th column, so a 1 appears in the 13th position of g . The 1s from the first 12 positions (one for each of the rows r_1, r_2, r_3) contribute three more 1s to g . Since

$wt(g) = 4$, we have accounted for all four 1s in g .

Therefore, the last 11 entries of g are 0. By the preceding lemma, a sum of only three rows of the matrix A cannot be 0. Therefore, this case is impossible.

(2) Second, suppose that the last row of G appears in the sum for g , say $g = r_1 + r_2 + r_3$ with $r_3 =$ the last row of G . Then the last 11 entries of g are formed from the sum of two rows of A (from r_1 and r_2) plus the vector $(1, 1, \dots, 1)$ from r_3 . Recall that the weight of the sum of two distinct rows of G is 8. There is a contribution of 2 to this weight from the first 13 columns. Therefore, looking at the last 11 columns, we see that the sum of two distinct rows of A has weight 6. Adding a vector mod 2 to the vector $(1, 1, \dots, 1)$ changes all the 1s to 0s and all the 0s to 1s. Therefore, the weight of the last 11 entries of g is 5. Since $wt(g) = 4$, this is impossible, so this case also cannot occur.

Finally, if g is the sum of four rows of G , then the first 12 entries of g have four 1s. Therefore, the last 12 entries of g are all 0. By the lemma, a sum of four rows of B cannot be 0, so we have a contradiction. This completes the proof that there is no codeword of weight 4.

Since the weights are multiples of 4, the smallest possibility for the weight is 8. As we pointed out previously, there are codewords of weight 8, so we have proved that the minimum weight of G_{24} is 8. Therefore, G_{24} is a $[24, 12, 8]$ code, as claimed. This completes the proof of the theorem.

The (nonextended) Golay code G_{23} is obtained by deleting the last entry of each codeword in G_{24} .

Theorem

G_{23} is a linear $[23, 12, 7]$ code.

Proof. Clearly each codeword has length 23. Also, the set of vectors in G_{23} is easily seen to be closed under addition (if v_1, v_2 are vectors of length 24, then the first 23 entries of $v_1 + v_2$ are computed from the first 23 entries of v_1 and v_2) and G_{23} forms a binary vector space. The generating matrix G' for G_{23} is obtained by removing the last column of the matrix G for G_{24} . Since G' contains the 12×12 identity matrix, the rows of G' are linearly independent, and hence span a 12-dimensional vector space. If g' is a codeword in G_{23} , then g' can be obtained by removing the last entry of some element g of G_{24} . If $g' \neq 0$, then $g \neq 0$, so $\text{wt}(g) \geq 8$. Since g' has one entry fewer than g , we have $\text{wt}(g') \geq 7$. This completes the proof.

Decoding G_{24}

Suppose a message is encoded using G_{24} and the received message contains at most three errors. In the following, we show a way to correct these errors.

Let G be the 12×24 generating matrix for G_{24} . Write G in the form

$$G = [I, B] = (c_1, \dots, c_{24}),$$

where I is the 12×12 identity matrix, B consists of the last 12 columns of G , and c_1, \dots, c_{24} are column vectors. Note that c_1, \dots, c_{12} are the standard basis elements for 12-dimensional space. Write

$$B^T = (b_1, \dots, b_{12}),$$

where b_1, \dots, b_{12} are column vectors. This means that b_1^T, \dots, b_{12}^T are the rows of B .

Suppose the received message is $r = c + e$, where c is a codeword from G_{24} and

$$e = (e_1, \dots, e_{24})$$

is the error vector. We assume $wt(e) \leq 3$.

The algorithm is as follows. The justification is given below.

1. Let $s = rG^T$ be the syndrome.
2. Compute the row vectors $s, sB, s + c_j^T, 13 \leq j \leq 24$, and $sB + b_j^T, 1 \leq j \leq 12$.
3. If $wt(s) \leq 3$, then the nonzero entries of s correspond to the nonzero entries of e .
4. If $wt(sB) \leq 3$, then there is a nonzero entry in the k th position of sB exactly when the $(k+12)$ th entry of e is nonzero.
5. If $wt(s + c_j^T) \leq 2$ for some j with $13 \leq j \leq 24$, then $e_j = 1$ and the nonzero entries of $s + c_j^T$ are in the positions of the other nonzero entries of the error vector e .
6. If $wt(sB + b_j^T) \leq 2$ for some j with $1 \leq j \leq 12$, then $e_j = 1$. If there is a nonzero entry for this $sB + b_j^T$ in position k (there are at most two such k), then $e_{12+k} = 1$.

Example

The sender starts with the message

$$m = (1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0).$$

The codeword is computed as

$$mG = (1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0)$$

and sent to us. Suppose we receive the message as

$$r = (1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0).$$

A calculation shows that

$$s = (0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0)$$

and

$$sB = (1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0).$$

Neither of these has weight at most 3, so we compute $s + c_j^T$, $13 \leq j \leq 24$ and $sB + b_j^T$, $1 \leq j \leq 12$. We find that

$$sB + b_4^T = (0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0).$$

This means that there is an error in position 4 (corresponding to the choice b_4) and in positions 20 ($= 12 + 8$) and 22 ($= 12 + 10$) (corresponding to the nonzero entries in positions 8 and 10 of $sB + b_4^T$). We therefore compute

$$\begin{aligned} c &= r + (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0) \\ &= (1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0). \end{aligned}$$

Moreover, since G is in systematic form, we recover the original message from the first 12 entries:

$$m = (1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0).$$

We now justify the algorithm and show that if $wt(e) \leq 3$, then at least one of the preceding cases occurs.

Since G_{24} is self-dual, the dot product of a row of G with any codeword c is 0. This means that $cG^T = 0$. In our case, we have $r = c + e$, so

$$s = rG^T = cG^T + eG^T = eG^T = e_1c_1^T + \cdots + e_{24}c_{24}^T.$$

This last equality just expresses the fact that the vector $e = (e_1, \dots, e_{24})$ times the matrix G^T equals e_1 times the first row c_1^T of G^T , plus e_2 times the second row of G^T , etc. Also,

$$sB = eG^T B = e \begin{bmatrix} I \\ B^T \end{bmatrix} B = e \begin{bmatrix} B \\ I \end{bmatrix},$$

since $B^T = B^{-1}$ (proved in the preceding lemma). We have

$$\begin{bmatrix} B \\ I \end{bmatrix} = [B^T, I]^T = (b_1, \dots, b_{12}, c_1, \dots, c_{12}).$$

Therefore,

$$sB = e(b_1, \dots, b_{12}, c_1, \dots, c_{12})^T = e_1 b_1^T + \dots + e_{24} c_{12}^T.$$

If $\text{wt}(e) \leq 3$, then either $\text{wt}((e_1, \dots, e_{12})) \leq 1$ or $\text{wt}((e_{13}, \dots, e_{24})) \leq 1$, since otherwise there would be too many nonzero entries in e . We therefore consider the following four cases.

1. $\text{wt}((e_1, \dots, e_{12})) = 0$. Then

$$sB = e_{13} c_1^T + \dots + e_{24} c_{12}^T = (e_{13}, \dots, e_{24}).$$

Therefore, $\text{wt}(sB) \leq 3$ and we can determine the errors as in step (4) of the algorithm.

2. $\text{wt}((e_1, \dots, e_{12})) = 1$. Then $e_j = 1$ for exactly one j with $1 \leq j \leq 12$, so

$$sB = b_j^T + e_{13} c_1^T + \dots + e_{24} c_{12}^T.$$

Therefore,

$$sB + b_j^T = e_{13} c_1^T + \dots + e_{24} c_{12}^T = (e_{13}, \dots, e_{24}).$$

The vector (e_{13}, \dots, e_{24}) has at most two nonzero entries, so we are in step (6) of the algorithm.

The choice of j is uniquely determined by sB . Suppose $\text{wt}(sB + b_k^T) \leq 2$ for some $k \neq j$. Then

$$\begin{aligned} \text{wt}(b_k^T + b_j^T) &= \text{wt}(sB + b_k^T + sB + b_j^T) \\ &\leq \text{wt}(sB + b_k^T) + \text{wt}(sB + b_j^T) \leq 2 + 2 = 4 \end{aligned}$$

(see [Exercise 6](#)). However, we showed in the proof of the theorem about G_{24} that the weight of the sum of any two distinct rows of G has weight 8, from which it follows that the sum of any two distinct rows of B has weight 6. Therefore, $\text{wt}(b_k^T + b_j^T) = 6$.

This contradiction shows that b_k cannot exist, so b_j is unique.

3. $\text{wt}((e_{13}, \dots, e_{24})) = 0$. In this case,

$$s = e_1 c_1^T + \dots + e_{12} c_{12}^T = (e_1, \dots, e_{12}).$$

We have $\text{wt}(s) \leq 3$, so we are in step (3) of the algorithm.

4. $\text{wt}((e_{13}, \dots, e_{24})) = 1$. In this case, $e_j = 1$ for some j with $13 \leq j \leq 24$. Therefore,

$$s = e_1 c_1^T + \dots + e_{12} c_{12}^T + c_j^T,$$

and we obtain

$$s + c_j^T = e_1 c_1^T + \cdots + e_{12} c_{12}^T = (e_1, \dots, e_{12}).$$

There are at most two nonzero entries in (e_1, \dots, e_{12}) , so we are in step (5) of the algorithm.

As in (2), the choice of c_j is uniquely determined by s .

In each of these cases, we obtain a vector, let's call it e' , with at most three nonzero entries. To correct the errors, we add (or subtract; we are working mod 2) e' to the received vector r to get $c' = r + e'$. How do we know this is the vector that was sent? By the choice of e' , we have

$$e' G^T = s,$$

so

$$c' G^T = r G^T + e' G^T = s + s = 0.$$

Since G_{24} is self-dual, G is a parity check matrix for G_{24} . Since $c' G^T = 0$, we conclude that c' is a codeword. We obtained c' by correcting at most three errors in r . Since we assumed there were at most three errors, and since the minimum weight of G_{24} is 8, this must be the correct decoding. So the algorithm actually corrects the errors, as claimed.

The preceding algorithm requires several steps. We need to compute the weights of 26 vectors. Why not just look at the various possibilities for three errors and see which correction yields a codeword? There are

$$\binom{24}{0} + \binom{24}{1} + \binom{24}{2} + \binom{24}{3} = 2325 \text{ possibilities}$$

for the locations of at most three errors, so this could be done on a computer. However, the preceding decoding algorithm is faster.

24.7 Cyclic Codes

Cyclic codes are a very important class of codes. In the next two sections, we'll meet two of the most useful examples of these codes. In this section, we describe the general framework.

A code C is called **cyclic** if

$$(c_1, c_2, \dots, c_n) \in C \text{ implies } (c_n, c_1, c_2, \dots, c_{n-1}) \in C.$$

For example, if $(1, 1, 0, 1)$ is in a cyclic code, then so is $(1, 1, 1, 0)$. Applying the definition two more times, we see that $(0, 1, 1, 1)$ and $(1, 0, 1, 1)$ are also codewords, so all cyclic permutations of the codeword are codewords. This might seem to be a strange condition for a code to satisfy. After all, it would seem to be rather irrelevant that, for a given codeword, all of its cyclic shifts are still codewords. The point is that cyclic codes have a lot of structure, which makes them easier to study. In the case of BCH codes (see [Section 24.8](#)), this structure yields an efficient decoding algorithm.

Let's start with an example. Consider the binary matrix

$$G = \begin{matrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{matrix} .$$

The rows of G generate a three-dimensional subspace of seven-dimensional binary space. In fact, in this case, the cyclic shifts of the first row give all the nonzero codewords:

$$G = \{(0, 0, 0, 0, 0, 0, 0), (1, 0, 1, 1, 1, 0, 0), (0, 1, 0, 1, 1, 1, 0), (0, 0, 1, 0, 1, 1, 1), (1, 0, 0, 1, 0, 1, 1), (1, 1, 0, 0, 1, 0, 1), (1, 1, 1, 0, 0, 1, 0), (0, 1, 1, 1, 0, 0, 1)\}.$$

Clearly the minimum weight is 4, so we have a cyclic [7, 3, 4] code.

We now show an algebraic way to obtain this code. Let $\mathbf{Z}_2[X]$ denote polynomials in X with coefficients mod 2, and let $\mathbf{Z}_2[X]/(X^7 - 1)$ denote these polynomials mod $(X^7 - 1)$. For a detailed description of what this means, see [Section 3.11](#). For the present, it suffices to say that working mod $X^7 - 1$ means we are working with polynomials of degree less than 7. Whenever we have a polynomial of degree 7 or higher, we divide by $X^7 - 1$ and take the remainder.

Let $g(X) = 1 + X^2 + X^3 + X^4$. Consider all products

$$g(X)f(X) = a_0 + a_1X + \cdots + a_6X^6$$

with $f(X)$ of degree ≤ 2 . Write the coefficients of the product as a vector (a_0, \dots, a_6) . For example, $g(X) \cdot 1$ yields $(1, 0, 1, 1, 1, 0, 0)$, which is the top row of G . Similarly, $g(X)X$ yields the second row of G and $g(X)X^2$ yields the third row of G . Also, $g(X)(1 + X^2)$ yields $(1, 0, 0, 1, 0, 1, 1)$, which is the sum of the first and third rows of G . In this way, we obtain all the codewords of our code.

We obtained this code by considering products $g(X)f(X)$ with $\deg(f) \leq 2$. We could also work with $f(X)$ of arbitrary degree and obtain the same code, as long as we work mod $(X^7 - 1)$. Note that $g(X)(X^3 + X^2 + 1) = X^7 - 1 \pmod{2}$. Divide $X^3 + X^2 + 1$ into $f(X)$:

$$f(X) = (X^3 + X^2 + 1)q(X) + f_1(X),$$

with $\deg(f_1) \leq 2$. Then

$$\begin{aligned} g(X)f(X) &= g(X)(X^3 + X^2 + 1)q(X) + g(X)f_1(X) \\ &= (X^7 - 1)q(X) + g(X)f_1(X) \equiv g(X)f_1(X) \pmod{X^7 - 1}. \end{aligned}$$

Therefore, $g(X)f_1(X)$ gives the same codeword as $g(X)f(X)$, so we may restrict to working with polynomials of degree at most two, as claimed.

Why is the code cyclic? Start with the vector for $g(X)$. The vectors for $g(X)X$ and $g(X)X^2$ are cyclic shifts of the one for $g(X)$ by one place and by two places, respectively. What happens if we multiply by X^3 ? We obtain a polynomial of degree 7, so we divide by $X^7 - 1$ and take the remainder:

$$g(X)X^3 = X^3 + X^5 + X^6 + X^7 = (X^7 - 1)(1) + (1 + X^3 + X^5 + X^6).$$

The remainder yields the vector $(1, 0, 0, 1, 0, 1, 1)$. This is the cyclic shift by three places of the vector for $g(X)$.

A similar calculation for $j = 4, 5, 6$ shows that the vector for $g(X)X^j$ yields the shift by j places of the vector for $g(X)$. In fact, this is a general phenomenon. If $q(X) = a_0 + a_1X + \dots + a_6X^6$ is a polynomial, then

$$\begin{aligned} q(X)X &= a_0X + a_1X^2 + \dots + a_6X^7 \\ &= a_6(X^7 - 1) + a_6 + a_0X + a_1X^2 + \dots + a_5X^6. \end{aligned}$$

The remainder is $a_6 + a_0X + a_1X^2 + \dots + a_5X^6$, which corresponds to the vector (a_6, a_0, \dots, a_5) . Therefore, multiplying by X and reducing mod $X^7 - 1$ corresponds to a cyclic shift by one place of the corresponding vector. Repeating this j times shows that multiplying by X^j corresponds to shifting by j places.

We now describe the general situation. Let \mathbf{F} be a finite field. For a treatment of finite fields, see [Section 3.11](#). For the present purposes, you may think of \mathbf{F} as being the integers mod p , where p is a prime number, since this is an example of a finite field. For example, you could take $\mathbf{F} = \mathbf{Z}_2 = \{0, 1\}$, the integers mod 2. Let $\mathbf{F}[X]$ denote polynomials in X with coefficients in \mathbf{F} . Choose a positive integer n . We'll work in $\mathbf{F}[X]/(X^n - 1)$, which denotes the elements of $\mathbf{F}[X]$ mod $(X^n - 1)$. This means we're working with polynomials of degree less than n . Whenever we encounter a polynomial of degree $\geq n$, we divide by $X^n - 1$ and take the

remainder. Let $g(X)$ be a polynomial in $\mathbf{F}[X]$. Consider the set of polynomials

$$m(X) = g(X)f(X) \bmod (X^n - 1),$$

where $f(X)$ runs through all polynomials in $\mathbf{F}[X]$ (we only need to consider $f(X)$ with degree less than n , since higher-degree polynomials can be reduced mod $X^n - 1$). Write

$$m(X) = a_0 + a_1X + \cdots + a_{n-1}X^{n-1}.$$

The coefficients give us the n -dimensional vector (a_0, \dots, a_{n-1}) . The set of all such coefficients forms a subspace C of n -dimensional space \mathbf{F}^n . Then C is a code.

If $m(X) = g(X)f(X) \bmod (X^n - 1)$ is any such polynomial, and $s(X)$ is another polynomial, then $m(X)s(X) = g(X)f(X)s(X) \bmod (X^n - 1)$ is the multiple of $g(X)$ by the polynomial $f(X)s(X)$. Therefore, it yields an element of the code C . In particular, multiplication by X and reducing mod $X^n - 1$ corresponds to a codeword that is a cyclic shift of the original codeword, as above. Therefore, C is cyclic.

The following theorem gives the general description of cyclic codes.

Theorem

Let C be a cyclic code of length n over a finite field \mathbf{F} . To each codeword $(a_0, \dots, a_{n-1}) \in C$, associate the polynomial $a_0 + a_1X + \cdots + a_{n-1}X^{n-1}$ in $\mathbf{F}[X]$. Among all the nonzero polynomials obtained from C in this way, let $g(X)$ have the smallest degree. By dividing by its highest coefficient, we may assume that the highest nonzero coefficient of $g(X)$ is 1. The polynomial $g(X)$ is called the **generating polynomial** for C . Then

1. $g(X)$ is uniquely determined by C .
2. $g(X)$ is a divisor of $X^n - 1$.
3. C is exactly the set of coefficients of the polynomials of the form $g(X)f(X)$ with $\deg(f) \leq n - 1 - \deg(g)$.
4. Write $X^n - 1 = g(X)h(X)$. Then $m(X) \in \mathbf{F}[X]/(X^n - 1)$ corresponds to an element of C if and only if $h(X)m(X) \equiv 0 \pmod{(X^n - 1)}$.

Proof.

1. If $g_1(X)$ is another such polynomial, then $g(X)$ and $g_1(X)$ have the same degree and have highest nonzero coefficient equal to 1. Therefore, $g(X) - g_1(X)$ has lower degree and still corresponds to a codeword, since C is closed under subtraction. Since $g(X)$ had the smallest degree among nonzero polynomials corresponding to codewords, $g(X) - g_1(X)$ must be 0, which means that $g_1(X) = g(X)$. Therefore, $g(X)$ is unique.

2. Divide $g(X)$ into $X^n - 1$:

$$X^n - 1 = g(X)h(X) + r(X)$$

for some polynomials $h(X)$ and $r(X)$, with $\deg(r) < \deg(g)$. This means that

$$-r(X) \equiv g(X)h(X) \pmod{(X^n - 1)}.$$

As explained previously, multiplying $g(X)$ by powers of X corresponds to cyclic shifts of the codeword associated to $g(X)$. Since C is assumed to be cyclic, the polynomials $g(X)X^j \pmod{(X^n - 1)}$ for $j = 0, 1, 2, \dots$ therefore correspond to codewords; call them c_0, c_1, c_2, \dots . Write $h(X) = b_0 + b_1X + \dots + b_kX^k$. Then $g(X)h(X)$ corresponds to the linear combination

$$b_0c_0 + b_1c_1 + \dots + b_kc_k.$$

Since each b_i is in \mathbf{F} and each c_i is in C , we have a linear combination of elements of C . But C is a vector subspace of n -dimensional space \mathbf{F}^n . Therefore, this linear combination is in C . This means that $r(X)$, which is $g(X)h(X) \pmod{(X^n - 1)}$, corresponds to a codeword. But $\deg(r) < \deg(g)$, which is the minimal degree of a polynomial corresponding to a nonzero codeword in C . Therefore, $r(X) = 0$. Consequently $X^n - 1 = g(X)h(X)$, so $g(X)$ is a divisor of $X^n - 1$.

3. Let $m(X)$ correspond to an element of C . Divide $g(X)$ into $m(X)$:

$$m(X) = g(X)f(X) + r_1(X),$$

with $\deg(r_1(X)) < \deg(g(X))$. As before, $g(X)f(X) \bmod (X^n - 1)$ corresponds to a codeword. Also, $m(X)$ corresponds to a codeword, by assumption. Therefore, $m(X) - g(X)f(X) \bmod (X^n - 1)$ corresponds to the difference of these codewords, which is a codeword. But this polynomial is just $r_1(X) = r_1(X) \bmod (X^n - 1)$. As before, this polynomial has degree less than $\deg(g(X))$, so $r_1(X) = 0$. Therefore, $m(X) = g(X)f(X)$. Since $\deg(m) \leq n - 1$, we must have $\deg((f)) \leq n - 1 - \deg(g)$. Conversely, as explained in the proof of (2), since C is cyclic, any such polynomial of the form $g(X)f(X)$ yields a codeword. Therefore, these polynomials yield exactly the elements of C .

4. Write $X^n - 1 = g(X)h(X)$, which can be done by (2). Suppose $m(X)$ corresponds to an element of C . Then $m(X) = g(X)f(X)$, by (3), so

$$h(X)m(X) = h(X)g(X)f(X) = (X^n - 1)f(X) \equiv 0 \pmod{(X^n - 1)}.$$

Conversely, suppose $m(X)$ is a polynomial such that $h(X)m(X) \equiv 0 \pmod{(X^n - 1)}$. Write $h(X)m(X) = (X^n - 1)q(X) = h(X)g(X)q(X)$, for some polynomial $q(X)$. Dividing by $h(X)$ yields $m(X) = g(X)q(X)$, which is a multiple of $g(X)$, and hence corresponds to a codeword. This completes the proof of the theorem.

Let $g(X) = a_0 + a_1X + \dots + a_{k-1}X^{k-1} + X^k$ be as in the theorem. By part (3) of the theorem, every element of C corresponds to a polynomial of the form $g(X)f(X)$, with $\deg(f(X)) \leq n - 1 - k$. This means that each such $f(X)$ is a linear combination of the monomials $1, X, X^2, \dots, X^{n-1-k}$. It follows that the codewords of C are linear combinations of the codewords corresponding to the polynomials

$$g(X), g(X)X, g(X)X^2, \dots, g(X)X^{n-1-k}.$$

But these are the vectors

$$(a_0, \dots, a_k, 0, 0, \dots), (0, a_0, \dots, a_k, 0, \dots), \dots, (0, \dots, 0, a_0, \dots, a_k).$$

Therefore, a generating matrix for C can be given by

$$G = \begin{matrix} a_0 & a_1 & \cdots & a_k & 0 & 0 & \cdots \\ 0 & a_0 & a_1 & \cdots & a_k & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & a_0 & a_1 & \cdots & a_k \end{matrix}.$$

We can use part (4) of the theorem to obtain a parity check matrix for C . Let

$h(X) = b_0 + b_1X + \cdots + b_lX^l$ be as in the theorem (where $l = n - k$). We'll prove that the $k \times n$ matrix

$$H = \begin{matrix} b_l & b_{l-1} & \cdots & b_0 & 0 & 0 & \cdots \\ 0 & b_l & b_{l-1} & \cdots & b_0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b_l & b_{l-1} & \cdots & b_0 \end{matrix}$$

is a parity check matrix for C . Note that the order of the coefficients of $h(X)$ is reversed. Recall that H is a parity check matrix for C means that $Hc^T = 0$ if and only if $c \in C$.

Proposition

H is a parity check matrix for C .

Proof. First observe that since $g(X)$ has 1 as its highest nonzero coefficient, and since $g(X)h(X) = X^n - 1$, the highest nonzero coefficient b_l of $h(X)$ must also be 1. Therefore, H is in row echelon form and consequently its rows are linearly independent. Since H has k rows, it has rank k . The right null space of H therefore has dimension $n - k$.

Let $m(X) = c_0 + c_1X + \cdots + c_{n-1}X^{n-1}$. We know from part (4) that $(c_0, c_1, \dots, c_{n-1}) \in C$ if and only if $h(X)m(X) \equiv 0 \pmod{X^n - 1}$.

Choose j with $l \leq j \leq n - 1$ and look at the coefficient of X^j in the product $h(X)m(X)$. It equals

$$b_0c_j + b_1c_{j-1} + \cdots + b_{l-1}c_{j-l+1} + b_lc_{j-l}.$$

There is a technical point to mention: Since we are looking at $h(X)m(X) \pmod{X^n - 1}$, we need to worry about a contribution from the term X^{n+j} (since $X^{n+j} \equiv X^n X^j \equiv 1 \cdot X^j$, the monomial X^{n+j} reduces

to X^j). However, the highest-degree term in the product $h(X)m(X)$ before reducing mod $X^n - 1$ is $c_{n-1}X^{l+n-1}$. Since $l \leq j$, we have $l + n - 1 \geq j + n$. Therefore, there is no term with X^{n+j} to worry about.

When we multiply H times $(c_0, c_1, \dots, c_{n-1})^T$, we obtain a vector whose first entry is

$$b_l c_0 + b_{l-1} c_1 + \cdots + b_0 c_l.$$

More generally, the i th entry (where $1 \leq i \leq k$) is

$$b_l c_{i-1} + b_{l-1} c_i + \cdots + b_0 c_{l+i-1}.$$

This is the coefficient of X^{l+i-1} in the product $h(X)m(X) \bmod (X^n - 1)$.

If $(c_0, c_1, \dots, c_{n-1})$ is in C , then $h(X)m(X) \equiv 0 \bmod (X^n - 1)$, so all these coefficients are 0. Therefore, H times $(c_0, c_1, \dots, c_{n-1})^T$ is the 0 vector, so the transposes of the vectors of C are contained in the right null space of H . Since both C and the null space have dimension k , we must have equality. This proves that $c \in C$ if and only if $Hc^T = 0$, which means that H is a parity check matrix for C .

Example

In the example at the beginning of this section, we had $n = 7$ and $g(X) = X^4 + X^3 + X^2 + 1$. We have $g(X)(X^3 + X^2 + 1) = X^7 - 1$, so $h(X) = X^3 + X^2 + 1$. The parity check matrix is

$$H = \begin{matrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{matrix}.$$

The parity check matrix gives a way of detecting errors, but correcting errors for general cyclic codes is generally

quite difficult. In the next section, we describe a class of cyclic codes for which a good decoding algorithm exists.

24.8 BCH Codes

BCH codes are a class of cyclic codes. They were discovered around 1959 by R. C. Bose and D. K. Ray-Chaudhuri and independently by A. Hocquenghem. One reason they are important is that there exist good decoding algorithms that correct multiple errors (see, for example, [Gallager] or [Wicker]). BCH codes are used in satellites. The special BCH codes called Reed-Solomon codes (see [Section 24.9](#)) have numerous applications.

Before describing BCH codes, we need a fact about finite fields. Let \mathbf{F} be a finite field with q elements. From [Section 3.11](#), we know that $q = p^m$ is a power of a prime number p . Let n be a positive integer not divisible by p . Then it can be proved that there exists a finite field \mathbf{F}' containing \mathbf{F} such that \mathbf{F}' contains a primitive n th root of unity α . This means that $\alpha^n = 1$, but $\alpha^k \neq 1$ for $1 \leq k < n$.

For example, if $\mathbf{F} = \mathbf{Z}_2$, the integers mod 2, and $n = 3$, we may take $\mathbf{F}' = GF(4)$. The element ω in the description of $GF(4)$ in [Section 3.11](#) is a primitive third root of unity. More generally, a primitive n th root of unity exists in a finite field \mathbf{F}' with q' elements if and only if $n|q' - 1$.

The reason we need the auxiliary field \mathbf{F}' is that several of the calculations we perform need to be carried out in this larger field. In the following, when we use an n th root of unity α , we'll implicitly assume that we're calculating in some field \mathbf{F}' that contains α . The results of the calculations, however, will give results about codes over the smaller field \mathbf{F} .

The following result, often called the **BCH bound**, gives an estimate for the minimum weight of a cyclic code.

Theorem 24.8

Let C be a cyclic $[n, k, d]$ code over a finite field \mathbf{F} , where \mathbf{F} has $q = p^m$ elements. Assume $p \nmid n$. Let $g(X)$ be the generating polynomial for C . Let α be a primitive n th root of unity and suppose that for some integers ℓ and δ ,

$$g(\alpha^\ell) = g(\alpha^{\ell+1}) = \cdots = g(\alpha^{\ell+\delta}) = 0.$$

Then $d \geq \delta + 2$.

Proof. Suppose $(c_0, c_1, \dots, c_{n-1}) \in C$ has weight w with $1 \leq w < \delta + 2$. We want to obtain a contradiction. Let

$m(X) = c_0 + c_1X + \cdots + c_{n-1}X^{n-1}$. We know that $m(X)$ is a multiple of $g(X)$, so

$$m(\alpha^\ell) = m(\alpha^{\ell+1}) = \cdots = m(\alpha^{\ell+\delta}) = 0.$$

Let $c_{i_1}, c_{i_2}, \dots, c_{i_w}$ be the nonzero coefficients of $m(X)$, so

$$m(X) = c_{i_1}X^{i_1} + c_{i_2}X^{i_2} + \cdots + c_{i_w}X^{i_w}.$$

The fact that $m(\alpha^j) = 0$ for $l \leq j \leq \ell + w - 1$ (note that $w - 1 \leq \delta$) can be rewritten as

$$\begin{pmatrix} \alpha^{\ell i_1} & \cdots & \alpha^{\ell i_w} \\ \alpha^{(\ell+1)i_1} & \cdots & \alpha^{(\ell+1)i_w} \\ \vdots & \ddots & \vdots \\ \alpha^{(\ell+w-1)i_1} & \cdots & \alpha^{(\ell+w-1)i_w} \end{pmatrix} \begin{pmatrix} c_{i_1} \\ c_{i_2} \\ \vdots \\ c_{i_w} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

We claim that the determinant of the matrix is nonzero. We need the following evaluation of the Vandermonde determinant. The proof can be found in most books on linear algebra.

Proposition

$$\det \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{pmatrix} = \prod_{1 \leq i < j \leq n} (x_j - x_i).$$

(The product is over all pairs of integers (i, j) with $1 \leq i < j \leq n$.) In particular, if x_1, \dots, x_n are pairwise distinct, the determinant is nonzero.

In our matrix, we can factor $\alpha^{\ell i_1}$ from the first column, $\alpha^{\ell i_2}$ from the second column, etc., to obtain

$$\begin{aligned} & \det \begin{pmatrix} \alpha^{\ell i_1} & \cdots & \alpha^{\ell i_w} \\ \alpha^{(\ell+1)i_1} & \cdots & \alpha^{(\ell+1)i_w} \\ \vdots & \ddots & \vdots \\ \alpha^{(\ell+w-1)i_1} & \cdots & \alpha^{(\ell+w-1)i_w} \end{pmatrix} \\ &= \alpha^{\ell i_1 + \cdots + \ell i_w} \det \begin{pmatrix} 1 & \cdots & 1 \\ \alpha^{i_1} & \cdots & \alpha^{i_w} \\ \vdots & \ddots & \vdots \\ \alpha^{(w-1)i_1} & \cdots & \alpha^{(w-1)i_w} \end{pmatrix}. \end{aligned}$$

Since $\alpha^{i_1}, \dots, \alpha^{i_w}$ are pairwise distinct, the determinant is nonzero. Why are these numbers distinct? Suppose $\alpha^{i_j} = \alpha^{i_k}$. We may assume $i_j \leq i_k$. We have $0 \leq i_j \leq i_k < n$. Therefore, $0 \leq i_k - i_j < n$. Note that $\alpha^{i_k - i_j} = 1$. Since α is a primitive n th root of unity, $\alpha^i \neq 1$ for $1 \leq i < n$. Therefore, $i_k - i_j = 0$, so $i_j = i_k$. This means that the numbers $\alpha^{i_1}, \dots, \alpha^{i_w}$ are pairwise distinct, as claimed.

Since the determinant is nonzero, the matrix is nonsingular. This implies that $(c_{i_1}, \dots, c_{i_w}) = 0$, contradicting the fact that these were the nonzero c_i 's. Therefore, all nonzero codewords have weight at least $\delta + 2$. This completes the proof of the theorem.

Example

Let $\mathbf{F} = \mathbf{Z}_2$ = the integers mod 2, and let $n = 3$. Let $g(X) = X^2 + X + 1$. Then

$$C = \{(0, 0, 0), (1, 1, 1)\},$$

which is a binary repetition code. Let ω be a primitive third root of unity, as in the description of $GF(4)$ in Section 3.11. Then $g(\omega) = g(\omega^2) = 0$. In the theorem, we can therefore take $\ell = 1$ and $\delta = 1$. We find that the minimal weight of C is at least 3. In this case, the bound is sharp, since the minimal weight of C is exactly 3.

Example

Let \mathbf{F} be any finite field and let n be any positive integer. Let $g(X) = X - 1$. Then $g(1) = 0$, so we may take $\ell = 0$ and $\delta = 0$. We conclude that the minimum weight of the code generated by $g(X)$ is at least 2 (actually, the theorem assumes that $p \nmid n$, but this assumption is not needed for this special case where $\ell = \delta = 0$). We have seen this code before. If (c_0, \dots, c_{n-1}) is a vector, and $m(X) = c_0 + \dots + c_{n-1}X^{n-1}$ is the associated polynomial, then $m(X)$ is a multiple of $X - 1$ exactly when $m(1) = 0$. This means that $c_0 + \dots + c_{n-1} = 0$. So a vector is a codeword if and only if the sum of its entries is 0. When $\mathbf{F} = \mathbf{Z}_2$, this is the parity check code, and for other finite fields it is a generalization of the parity check code. The fact that its minimal weight is 2 is easy to see directly: If a codeword has a nonzero entry, then it must contain another nonzero entry to cancel it and make the sum of the entries be 0. Therefore, each nonzero codeword has at least two nonzero entries, and hence has weight at least 2. The vector $(1, -1, 0, \dots)$ is a codeword and has weight 2, so the minimal weight is exactly 2.

Example

Let's return to the example of a binary cyclic code of length 7 from Section 24.7. We have $\mathbf{F} = \mathbf{Z}_2$, and $g(X) = 1 + X^2 + X^3 + X^4$. We can factor $g(X) = (X - 1)(X^3 + X + 1)$. Let α be a root of $X^3 + X + 1$. Then α is a primitive seventh root of unity (see Exercise 18), and we are working in $GF(8)$. Since $\mathbf{Z}_2 \subset GF(8)$, we have $2 = 1 + 1 = 0$ and $-1 = 1$. Therefore, $\alpha^3 = \alpha + 1$. Squaring yields $\alpha^6 = \alpha^2 + 2\alpha + 1 = \alpha^2 + 1$. Therefore, $(\alpha^2)^3 + (\alpha^2) + 1 = 0$. This means that $g(\alpha^2) = 0$, so

$$g(1) = g(\alpha) = g(\alpha^2) = 0.$$

In the theorem, we can take $\ell = 0$ and $\delta = 2$. Therefore, the minimal weight in the code is at least 4 (in fact, it is exactly 4).

To define the BCH codes, we need some more notation. We are going to construct codes of length n over a finite field \mathbf{F} . Factor $X^n - 1$ into irreducible factors over \mathbf{F} :

$$X^n - 1 = f_1(X)f_2(X) \cdots f_r(X),$$

where each $f_i(X)$ is a polynomial with coefficients in \mathbf{F} , and each $f_i(X)$ cannot be factored into lower-degree polynomials with coefficients in \mathbf{F} . We may assume that the highest nonzero coefficient of each $f_i(X)$ is 1. Let α be a primitive n th root of unity. Then $\alpha^0, \alpha^1, \alpha^2, \dots, \alpha^{n-1}$ are roots of $X^n - 1$. This means that

$$X^n - 1 = (X - 1)(X - \alpha)(X - \alpha^2) \cdots (X - \alpha^{n-1}).$$

Therefore, each $f_i(X)$ is a product of some of these factors $(X - \alpha^j)$, and each α^j is a root of exactly one of the polynomials $f_i(X)$. For each j , let $q_j(X)$ be the polynomial $f_i(X)$ such that $f_i(\alpha^j) = 0$. This gives us polynomials $q_0(X), q_1(X), \dots, q_{n-1}(X)$. Of course, usually these polynomials are not all distinct, since a

polynomial $f_i(X)$ that has two different powers α^j , α^k as roots will serve as both $q_j(X)$ and $q_k(X)$ (see the examples given later in this section).

A **BCH code of designed distance d** is a code with generating polynomial

$g(X) = \text{least common multiple of } q_{k+1}(X), q_{k+2}(X), \dots, q_{k+d-1}(X)$

for some integer k .

Theorem

A BCH code of designed distance d has minimum weight greater than or equal to d .

Proof. Since $q_j(X)$ divides $g(X)$ for $k+1 \leq j \leq k+d-1$, and $q_j(\alpha^j) = 0$, we have

$$g(\alpha^{k+1}) = g(\alpha^{k+2}) = \dots = g(\alpha^{k+d-1}) = 0.$$

The BCH bound (with $\ell = k+1$ and $\delta = d-2$) implies that the code has minimum weight at least $d = \delta + 2$.

Example

Let $\mathbf{F} = \mathbf{Z}_2$, and let $n = 7$. Then

$$X^7 - 1 = (X - 1)(X^3 + X^2 + 1)(X^3 + X + 1).$$

Let α be a root of $X^3 + X + 1$. Then α is a primitive 7th root of unity, as in the previous example. Moreover, in that example, we showed that α^2 is also a root of $X^3 + X + 1$. In fact, we actually showed that the square of a root of $X^3 + X + 1$ is also a root, so we have that $\alpha^4 = (\alpha^2)^2$ is also a root of $X^3 + X + 1$. (We could square this again, but $\alpha^8 = \alpha$, so we are back

to where we started.) Therefore, α , α^2 , α^4 are the roots of $X^3 + X + 1$, so

$$X^3 + X + 1 = (X - \alpha)(X - \alpha^2)(X - \alpha^4).$$

The remaining powers of α must be roots of $X^3 + X^2 + 1$, so

$$X^3 + X^2 + 1 = (X - \alpha^3)(X - \alpha^5)(X - \alpha^6).$$

Therefore,

$$\begin{aligned} q_0(X) &= X - 1, & q_1(X) &= q_2(X) = q_4(X) = X^3 + X + 1, \\ q_3(X) &= q_5(X) = q_6(X) = X^3 + X^2 + 1. \end{aligned}$$

If we take $k = -1$ and $d = 3$, then

$$\begin{aligned} g(X) &= \text{lcm}(q_0(X), q_1(X)) \\ &= (X - 1)(X^3 + X + 1) = X^4 + X^3 + X^2 + 1. \end{aligned}$$

We obtain the cyclic [7, 3, 4] code discussed in [Section 24.7](#). The theorem says that the minimum weight is at least 3. In this case, we can do a little better. If we take $k = -1$ and $d = 4$, then we have a generating polynomial $g_1(X)$ with

$$g_1(X) = \text{lcm}(q_0(X), q_1(X), q_2(X)) = g(X).$$

This is because $q_2(X) = q_1(X)$, so the least common multiple doesn't change when $q_2(X)$ is included. The theorem now tells us that the minimum weight of the code is at least 4. As we have seen before, the minimum weight is exactly 4.

Example (continued)

Let's continue with the previous example, but take $k = 0$ and $d = 7$. Then

$$\begin{aligned} g(X) &= \text{lcm}(q_1(X), \dots, q_6(X)) = (X^3 + X + 1)(X^3 + X^2 + 1) \\ &= X^6 + X^5 + X^4 + X^3 + X^2 + X + 1. \end{aligned}$$

We obtain the repetition code with only two codewords:

$$\{(0, 0, 0, 0, 0, 0, 0), (1, 1, 1, 1, 1, 1, 1)\}.$$

The theorem says that the minimum distance is at least 7. In fact it is exactly 7.

Example

Let $\mathbf{F} = \mathbf{Z}_5 = \{0, 1, 2, 3, 4\}$ = the integers mod 5. Let $n = 4$. Then

$$X^4 - 1 = (X - 1)(X - 2)(X - 3)(X - 4)$$

(this is an equality, or congruence if you prefer, in \mathbf{Z}_5).

Let $\alpha = 2$. We have $2^4 = 1$, but $2^j \neq 1$ for $1 \leq j < 4$.

Therefore, 2 is a primitive 4th root of unity in \mathbf{Z}_5 . We have $2^0 = 1$, $2^2 = 4$, $2^3 = 3$ (these are just congruences mod 5). Therefore,

$$q_0(X) = X - 1, \quad q_1(X) = X - 2, \quad q_2(X) = X - 4, \quad q_3(X) = X - 3.$$

In the theorem, let $k = 0$, $d = 3$. Then

$$\begin{aligned} g(X) &= \text{lcm}(q_1(X), q_2(X)) = (X - 2)(X - 4) \\ &= X^2 - 6X + 8 = X^2 + 4X + 3. \end{aligned}$$

We obtain a cyclic [4, 2] code over \mathbf{Z}_5 with generating matrix

$$\begin{pmatrix} 3 & 4 & 1 & 0 \\ 0 & 3 & 4 & 1 \end{pmatrix}.$$

The theorem says that the minimum weight is at least 3. Since the first row of the matrix is a codeword of weight 3, the minimum weight is exactly 3. This code is an example of a Reed-Solomon code, which will be discussed in the next section.

24.8.1 Decoding BCH Codes

One of the reason BCH codes are useful is that there are good decoding algorithms. One of the best known is due

to Berlekamp and Massey (see [Gallager] or [Wicker]). In the following, we won't give the algorithm, but, in order to give the spirit of some of the ideas that are involved, we show a way to correct one error in a BCH code with designed distance $d \geq 3$.

Let C be a BCH code of designed distance $d \geq 3$. Then C is a cyclic code, say of length n , with generating polynomial $g(X)$. There is a primitive n th root of unity α such that

$$g(\alpha^{k+1}) = g(\alpha^{k+2}) = 0$$

for some integer k .

Let

$$H = \begin{pmatrix} 1 & \alpha^{k+1} & \alpha^{2(k+1)} & \cdots & \alpha^{(n-1)(k+1)} \\ 1 & \alpha^{k+2} & \alpha^{2(k+2)} & \cdots & \alpha^{(n-1)(k+2)} \end{pmatrix}.$$

If $c = (c_0, \dots, c_{n-1})$ is a codeword, then the polynomial $m(X) = c_0 + c_1X + \cdots + c_{n-1}X^{n-1}$ is a multiple of $g(X)$, so

$$m(\alpha^{k+1}) = m(\alpha^{k+2}) = 0.$$

This may be rewritten in terms of H :

$$cH^T = (c_0, c_1, \dots, c_{n-1}) \begin{pmatrix} 1 & 1 \\ \alpha^{k+1} & \alpha^{k+2} \\ \alpha^{2(k+1)} & \alpha^{2(k+2)} \\ \vdots & \vdots \\ \alpha^{(n-1)(k+1)} & \alpha^{(n-1)(k+2)} \end{pmatrix} = 0.$$

H is not necessarily a parity check matrix for C , since there might be noncodewords that are also in the null space of H . However, as we shall see, H can correct an error.

Suppose the vector $r = c + e$ is received, where c is a codeword and $e = (e_0, \dots, e_{n-1})$ is an error vector. We assume that at most one entry of e is nonzero.

Here is the algorithm for correcting one error.

1. Write $rH^T = (s_1, s_2)$.
2. If $s_1 = 0$, there is no error (or there is more than one error), so we're done.
3. If $s_1 \neq 0$, compute s_2/s_1 . This will be a power α^{j-1} of α . The error is in the j th position. If we are working over the finite field \mathbf{Z}_2 , we are done, since then $e_j = 1$. But for other finite fields, there are several choices for the value of e_j .
4. Compute $e_j = s_1/\alpha^{(j-1)(k+1)}$. This is the j th entry of the error vector e . The other entries of e are 0.
5. Subtract the error vector e from the received vector r to obtain the correct codeword c .

Example

Let's look at the BCH code over \mathbf{Z}_2 of length 7 and designed distance 7 considered previously. It is the binary repetition code of length 7 and has two codewords: $(0, 0, 0, 0, 0, 0, 0)$, $(1, 1, 1, 1, 1, 1, 1)$. The algorithm corrects one error. Suppose the received vector is $r = (1, 1, 1, 1, 0, 1, 1)$. As before, let α be a root of $X^3 + X + 1$. Then α is a primitive 7th root of unity.

Before proceeding, we need to deduce a few facts about computing with powers of α . We have $\alpha^3 = \alpha + 1$. Multiplying this relation by powers of α yields

$$\begin{aligned}\alpha^4 &= \alpha^2 + \alpha, \\ \alpha^5 &= \alpha^3 + \alpha^2 = \alpha^2 + \alpha + 1, \\ \alpha^6 &= \alpha^3 + \alpha^2 + \alpha = (\alpha + 1) + \alpha^2 + \alpha = \alpha^2 + 1.\end{aligned}$$

Also, the fact that $\alpha^j = \alpha^{j \pmod 7}$ is useful.

We now can compute

$$\begin{aligned}
rH^T &= (1, 1, 1, 1, 0, 1, 1) \begin{pmatrix} 1 & 1 \\ \alpha & \alpha^2 \\ \alpha^2 & \alpha^4 \\ \vdots & \vdots \\ \alpha^6 & \alpha^{12} \end{pmatrix} \\
&= (1 + \alpha + \alpha^2 + \alpha^3 + \alpha^5 + \alpha^6, \quad 1 + \alpha^2 + \alpha^4 + \alpha^6 + \alpha^{10} + \alpha^{12}) \\
&= (\alpha + \alpha^2, \quad \alpha).
\end{aligned}$$

The sum in the first entry, for example, can be evaluated as follows:

$$1 + \alpha + \alpha^2 + \alpha^3 + \alpha^5 + \alpha^6 = 1 + \alpha + \alpha^2 + (1 + \alpha) + (\alpha^2 + \alpha + 1) + (\alpha^2 + 1) = \alpha + \alpha^2.$$

Therefore, $s_1 = \alpha + \alpha^2$ and $s_2 = \alpha$. We need to calculate s_2/s_1 . Since $s_1 = \alpha + \alpha^2 = \alpha^4$, we have

$$s_2/s_1 = \alpha/\alpha^4 = \alpha^{-3} = \alpha^4.$$

Therefore, $j - 1 = 4$, so the error is in position $j = 5$. The fifth entry of the error vector is $s_1/\alpha^4 = 1$, so the error vector is $(0, 0, 0, 0, 1, 0, 0)$. The corrected message is

$$r - e = (1, 1, 1, 1, 1, 1, 1).$$

Here is why the algorithm works. Since $cH^T = 0$, we have

$$rH^T = cH^T = eH^T = eH^T = (s_1, s_2).$$

If $e = (0, 0, \dots, e_j, 0, \dots)$ with $e_j \neq 0$, then the definition of H gives

$$s_1 = e_j \alpha^{(j-1)(k+1)}, \quad s_2 = e_j \alpha^{(j-1)(k+2)}.$$

Therefore, $s_2/s_1 = \alpha^{j-1}$. Also, $s_1/\alpha^{(j-1)(k+1)} = e_j$, as claimed.

24.9 Reed-Solomon Codes

The Reed-Solomon codes, constructed in 1960, are an example of BCH codes. Because they work well for certain types of errors, they have been used in spacecraft communications and in compact discs.

Let \mathbf{F} be a finite field with q elements and let $n = q - 1$. A basic fact from the theory of finite fields is that \mathbf{F} contains a primitive n th root of unity α . Choose d with $1 \leq d < n$ and let

$$g(X) = (X - \alpha)(X - \alpha^2) \cdots (X - \alpha^{d-1}).$$

This is a polynomial with coefficients in \mathbf{F} . It generates a BCH code C over \mathbf{F} of length n , called a **Reed-Solomon code**.

Since $g(\alpha) = \cdots = g(\alpha^{d-1}) = 0$, the BCH bound implies that the minimum distance for C is at least d . Since $g(X)$ is a polynomial of degree $d - 1$, it has at most d nonzero coefficients. Therefore, the codeword corresponding to the coefficients of $g(X)$ is a codeword of weight at most d . It follows that the minimum weight for C is exactly d . The dimension of C is $n - \deg(g) = n + 1 - d$. Therefore, a Reed-Solomon code is a cyclic $[n, n + 1 - d, d]$ code.

The codewords in C correspond to the polynomials

$$g(X)f(X) \text{ with } \deg(f) \leq n - d.$$

There are q^{n-d+1} such polynomials $f(X)$ since there are q choices for each of the $n - d + 1$ coefficients of $f(X)$, and thus there are q^{n-d+1} codewords in C . Therefore, a Reed-Solomon code is a MDS code, namely, one that makes the Singleton bound (Section 24.3) an equality.

Example

Let $\mathbf{F} = \mathbf{Z}_7 = \{0, 1, 2, \dots, 6\}$, the integers mod 7. Then $q = 7$ and $n = q - 1 = 6$. A primitive sixth root of unity α in \mathbf{F} is the same as a primitive root mod 7 (see Section 3.7). We may take $\alpha = 3$. Choose $d = 4$. Then

$$g(X) = (X - 3)(X - 3^2)(X - 3^3) = X^3 + 3X^2 + X + 6.$$

The code has generating matrix

$$G = \begin{pmatrix} 6 & 1 & 3 & 1 & 0 & 0 \\ 0 & 6 & 1 & 3 & 1 & 0 \\ 0 & 0 & 6 & 1 & 3 & 1 \end{pmatrix}.$$

There are $7^3 = 343$ codewords in the code, obtained by taking all linear combinations mod 7 of the three rows of G . The minimum weight of the code is 4.

Example

Let $\mathbf{F} = GF(4) = \{0, 1, \omega, \omega^2\}$, which was introduced in Section 3.11. Then \mathbf{F} has 4 elements, $n = q - 1 = 3$, and $\alpha = \omega$. Choose $d = 2$, so

$$g(X) = (X - \omega).$$

The matrix

$$G = \begin{pmatrix} \omega & 1 & 0 \\ 0 & \omega & 1 \end{pmatrix}$$

is a generating matrix for the code. The code contains all 16 linear combinations of the two rows of G , for example,

$$\omega \cdot (\omega, 1, 0) + 1 \cdot (0, \omega, 1) = (\omega^2, 0, 1).$$

The minimum weight of the code is 2.

In many applications, errors are not randomly distributed. Instead, they occur in bursts. For example,

in a CD, a scratch introduces errors in many adjacent bits. A burst of solar energy could have a similar effect on communications from a spacecraft. Reed-Solomon codes are useful in such situations.

For example, suppose we take $\mathbf{F} = GF(2^8)$. The elements of \mathbf{F} are represented as bytes of eight bits each, as in [Section 3.11](#). We have $n = 2^8 - 1 = 255$. Let $d = 33$. The codewords are then vectors consisting of 255 bytes. There are 222 information bytes and 33 check bytes. These codewords are sent as strings of $8 \times 255 = 2040$ binary bits. Disturbances in the transmission will corrupt some of these bits. However, in the case of bursts, these bits will often be in a small region of the transmitted string. If, for example, the corrupted bits all lie within a string of 121 ($= 15 \times 8 + 1$) consecutive bits, there can be errors in at most 16 bytes. Therefore, these errors can be corrected (because $16 < d/2$). On the other hand, if there were 121 bit errors randomly distributed through the string of 2040 bits, numerous bytes would be corrupted, and correct decoding would not be possible. Therefore, the choice of code depends on the type of errors that are expected.

24.10 The McEliece Cryptosystem

In this book, we have mostly described cryptographic systems that are based on number theoretic principles. There are many other cryptosystems that are based on other complex problems. Here we present one based on the difficulty of finding the nearest codeword for a linear binary code.

The idea is simple. Suppose you have a binary string of length 1024 that has 50 errors. There are $\binom{1024}{50} \approx 3 \times 10^{85}$ possible locations for these errors, so an exhaustive search that tries all possibilities is infeasible. Suppose, however, that you have an efficient decoding algorithm that is unknown to anyone else. Then only you can correct these errors and find the corrected string. McEliece showed how to use this to obtain a public key cryptosystem.

Bob chooses G to be the generating matrix for an (n, k) linear error correcting code C with $d(C) = d$. He chooses S to be a $k \times k$ matrix that is invertible mod 2 and lets P be an $n \times n$ permutation matrix, which means that P has exactly one 1 in every row and in every column, with all the other entries being 0. Define

$$G_1 = SGP.$$

The matrix G_1 is the public key for the cryptosystem. Bob keeps S, G, P secret.

In order for Alice to send Bob a message x , she generates a random binary string e of length n that has weight t . She forms the ciphertext by computing

$$y \equiv xG_1 + e \pmod{2}.$$

Bob decrypts y as follows:

1. Calculate $y_1 \equiv yP^{-1}$. (Since P is a permutation matrix, $e_1 = eP^{-1}$ is still a binary string of weight t . We have $y_1 \equiv xSG + e_1$.)
2. Apply the error decoder for the code C to y_1 to correct the “error” and obtain the codeword x_1 closest to y_1 .
3. Compute x_0 such that $x_0G \equiv x_1$ (in the examples we have considered, x_0 is simply the first k bits of x_1).
4. Compute $x \equiv x_0S^{-1}$.

The security of the system lies in the difficulty of decoding y_1 to obtain x_1 . There is a little security built into the system by S ; however, once a decoding algorithm is known for the code generated by GP , a chosen plaintext attack allows one to solve for the matrix S (as in the Hill cipher).

To make decoding difficult, $d(C)$ should be chosen to be large. McEliece suggested using a [1024, 512, 101] Goppa code. The **Goppa codes** have parameters of the form $n = 2^m$, $d = 2t + 1$, $k = n - mt$. For example, taking $m = 10$ and $t = 50$ yields the [1024, 524, 101] code just mentioned. It can correct up to 50 errors. For given values of m and t , there are in fact many inequivalent Goppa codes with these parameters. We will not discuss these codes here except to mention that they have an efficient decoding algorithm and therefore can be used to correct errors quickly.

Example

Consider the matrix

$$G = \begin{matrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{matrix},$$

which is the generator matrix for the [7, 4] Hamming code. Suppose Alice wishes to send a message

$$m = (1, 0, 1, 1)$$

to Bob. In order to do so, Bob must create an invertible matrix S and a random permutation matrix P that he will keep secret. If Bob chooses

$$S = \begin{matrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$

and

$$P = \begin{matrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{matrix} .$$

Using these, Bob generates the public encryption matrix

$$G_1 = \begin{matrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{matrix} .$$

In order to encrypt, Alice generates her own random error vector e and calculates the ciphertext $y = xG_1 + e$. In the case of a Hamming code the error vector has weight 1. Suppose Alice chooses

$$e = (0, 1, 0, 0, 0, 0, 0).$$

Then

$$y = (0, 0, 0, 1, 1, 0, 0).$$

Bob decrypts by first calculating

$$y_1 = yP^{-1} = (0, 0, 1, 0, 0, 0, 1).$$

Calculating the syndrome of y_1 by applying the parity check matrix H and changing the corresponding bit yields

$$x_1 = (0, 0, 1, 0, 0, 1, 1).$$

Bob next forms a vector x_0 such that $x_0G = x_1$, which can be done by extracting the first four components of x_1 , that is,

$$x_0 = (0, 0, 1, 0).$$

Bob decrypts by calculating

$$x = x_0S^{-1} = (1, 0, 1, 1),$$

which is the original plaintext message.

The McEliece system seems to be reasonably secure. For a discussion of its security, see [Chabaud]. A disadvantage of the system compared to RSA, for example, is that the size of the public key G_1 is rather large.

24.11 Other Topics

The field of error correcting codes is a vast subject that is explored by both the mathematical community and the engineering community. In this chapter we have only touched upon a select handful of the concepts of this field. There are many other areas of error correcting codes that we have not discussed.

Perhaps most notable of these is the study of convolutional codes. In this chapter we have entirely focused on block codes, where typically the data are segmented into blocks of a fixed length k and mapped into codewords of a fixed length n . However, in many applications, the data are produced in a continuous fashion, and it is better to map the stream of data into a stream of coded symbols. For example, such codes have the advantage of not requiring the delay needed to observe an entire block of symbols before encoding or decoding. A good analogy is that block codes are the coding theory analogue of block ciphers, while convolutional codes are the analogue of stream ciphers.

Another topic that is very important in the study of error correcting codes is that of efficient decoding. In the case of linear codes, we presented syndrome decoding, which is more efficient than performing a search for the nearest codeword. However, for large linear codes, syndrome decoding is still too inefficient to be practical. When BCH and Reed-Solomon codes were introduced, the decoding schemes that were originally presented were impractical for decoding more than a few errors. Later, Berlekamp and Massey provided an efficient approach to decoding BCH and Reed-Solomon codes. There is still a lot of research being done on this topic. We direct the reader to the books [Lin-Costello], [Wicker], [Gallager], and

[Berlekamp] for further discussion on the subject of decoding.

We have also focused entirely on bit or symbol errors. However, in modern computer networks, the types of errors that occur are not simply bit or symbol errors but also the complete loss of segments of data. For example, on the Internet, data are transferred over the network in chunks called packets. Due to congestion at various locations on the network, such as routers and switches, packets might be dropped and never reach their intended recipient. In this case, the recipient might notify the sender, requesting a packet to be resent. Protocols such as the Transmission Control Protocol (TCP) provide mechanisms for retransmitting lost packets.

When performing cryptography, it is critical to use a combination of many different types of error control techniques to assure reliable delivery of encrypted messages; otherwise, the receiver might not be able to decrypt the messages that were sent.

Finally, we mention that coding theory has strong connections with various problems in mathematics such as finding dense packings of high-dimensional spheres. For more on this, see [Thompson].

24.12 Exercises

1. Two codewords were sent using the Hamming [7, 4] code and were received as 0100111 and 0101010. Each one contains at most one error. Correct the errors. Also, determine the 4-bit messages that were multiplied by the matrix G to obtain the codewords.
2. An ISBN number is incorrectly written as 0-13-116093-8. Show that this is not a correct ISBN number. Find two different valid ISBN numbers such that an error in one digit would give this number. This shows that ISBN cannot correct errors.
3. The following is a parity check matrix for a binary $[n, k]$ code C :

$$\begin{matrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{matrix} .$$

1. Find n and k .
2. Find the generator matrix for C .
3. List the codewords in C .
4. What is the code rate for C ?
4. Let $C = \{(0, 0, 0), (1, 1, 1)\}$ be a binary repetition code.
 1. Find a parity check matrix for C .
 2. List the cosets and coset leaders for C .
 3. Find the syndrome for each coset.
 4. Suppose you receive the message $(1, 1, 0)$. Use the syndrome decoding method to decode it.
5. Let C be the binary code $\{(0, 0, 1), (1, 1, 1), (1, 0, 0), (0, 1, 0)\}$.
 1. Show that C is not linear.
 2. What is $d(C)$? (Since C is not linear, this cannot be found by calculating the minimum weight.)
 3. Show that C satisfies the Singleton bound with equality.

6. Show that the weight function (on \mathbf{F}^n) satisfies the triangle inequality: $wt(u + v) \leq wt(u) + wt(v)$.
7. Show that $A_q(n, n) = q$, where $A_q(n, d)$ is the function defined in [Section 24.3](#).
8. Let C be the repetition code of length n . Show that C^\perp is the parity check code of length n . (This is true for arbitrary \mathbf{F} .)
9. Let C be a linear code and let $u + C$ and $v + C$ be cosets of C . Show that $u + C = v + C$ if and only if $u - v \in C$. (Hint: To show $u + C = v + C$, it suffices to show that $u + c \in v + C$ for every $c \in C$, and that $v + c \in u + C$ for every $c \in C$. To show the opposite implication, use the fact that $u \in u + C$.)
10. Show that if C is a self-dual $[n, k, d]$ code, then n must be even.
11. Show that $g(X) = 1 + X + X^2 + \cdots + X^{n-1}$ is the generating polynomial for the $[n, 1]$ repetition code. (This is true for arbitrary \mathbf{F} .)
12. Let $g(X) = 1 + X + X^3$ be a polynomial with coefficients in \mathbf{Z}_2 .

1. Show that $g(X)$ is a factor of $X^7 - 1$ in $\mathbf{Z}_2[X]$.
2. The polynomial $g(X)$ is the generating polynomial for a cyclic code $[7, 4]$ code C . Find the generating matrix for C .
3. Find a parity check matrix H for C .
4. Show that $G'H^T = 0$, where

$$G' = \begin{matrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{matrix}.$$

5. Show that the rows of G' generate C .
6. Show that a permutation of the columns of G' gives the generating matrix for the Hamming $[7, 4]$ code, and therefore these two codes are equivalent.
13. Let C be the cyclic binary code of length 4 with generating polynomial $g(X) = X^2 + 1$. Which of the following polynomials correspond to elements of C ?
$$f_1(X) = 1 + X + X^3, \quad f_2(X) = 1 + X + X^2 + X^3, \quad f_3(X) = X^2 + X^3$$
14. Let $g(X)$ be the generating polynomial for a cyclic code C of length n , and let $g(X)h(X) = X^n - 1$. Write $h(X) = b_0 + b_1X + \cdots + X^\ell$. Show that the dual code C^\perp is

cyclic with generating polynomial

$\tilde{h}_r(X) = (1/b_0)(1 + b_{\ell-1}X + \dots + b_1X^{\ell-1} + b_0X^\ell)$. (The factor $1/b_0$ is included to make the highest nonzero coefficient be 1.)

15.
 1. Let C be a binary repetition code of odd length n (that is, C contains two vectors, one with all 0s and one with all 1s). Show that C is perfect. (Hint: Show that every vector lies in exactly one of the two spheres of radius $(n - 1)/2$.)
 2. Use (a) to show that if n is odd then $\sum_{j=0}^{(n-1)/2} \binom{n}{j} = 2^{n-1}$. (This can also be proved by applying the binomial theorem to $(1 + 1)^n$, and then observing that we're using half of the terms.)
16. Let $2 \leq d \leq n$ and let $V_q(n, d - 1)$ denote the number of points in a Hamming sphere of radius $d - 1$. The proof of the Gilbert-Varshamov bound constructs an (n, M, d) code with $M \geq q^n/V_q(n, d - 1)$. However, this code is probably not linear. This exercise will construct a linear $[n, k, d]$ code, where k is the smallest integer satisfying $q^k \geq q^{n-1}/V_q(n, d - 1)$.
 1. Show that there exists an $[n, 1, d]$ code C_1 .
 2. Suppose $q^{j-1} < q^n/V_q(n, d - 1)$ and that we have constructed an $[n, j - 1, d]$ code C_{j-1} in \mathbf{F}^n (where \mathbf{F} is the finite field with q elements). Show that there is a vector v with $d(v, c) \geq d$ for all $c \in C_{j-1}$.
 3. Let C_j be the subspace spanned by v and C_{j-1} . Show that C_j has dimension j and that every element of C_j can be written in the form $av + c$ with $a \in \mathbf{F}$ and $c \in C_{j-1}$.
 4. Let $av + c$, with $a \neq 0$, be an element of C_j , as in (c). Show that $wt(av + c) = wt(v + a^{-1}c) = d(v, -a^{-1}c) \geq d$.
 5. Show that C_j is an $[n, j, d]$ code. Continuing by induction, we obtain the desired code C_k .
 6. Here is a technical point. We have actually constructed an $[n, k, e]$ code with $e \geq d$. Show that by possibly modifying v in step (b), we may arrange that $d(v, c) = d$ for some $c \in C_{j-1}$, so we obtain an $[n, k, d]$ code.
17. Show that the Golay code G_{23} is perfect.
18. Let α be a root of the polynomial $X^3 + X + 1 \in \mathbf{Z}_2[X]$.

1. Using the fact that $X^3 + X + 1$ divides $X^7 - 1$, show that $\alpha^7 = 1$.
 2. Show that $\alpha \neq 1$.
 3. Suppose that $\alpha^j = 1$ with $1 \leq j < 7$. Then $\gcd(j, 7) = 1$, so there exist integers a, b with $ja + 7b = 1$. Use this to show that $\alpha^1 = 1$, which is a contradiction. This shows that α is a primitive seventh root of unity.
19. Let C be the binary code of length 7 generated by the polynomial $g(X) = 1 + X^2 + X^3 + X^4$. As in [Section 24.8](#), $g(1) = g(\alpha) = 0$, where α is a root of $X^3 + X + 1$. Suppose the message $(1, 0, 1, 1, 0, 1, 1)$ is received. It has one error. Use the procedure from [Section 24.8](#) to correct the error.
20. Let $C \subset \mathbf{F}^n$ be a cyclic code of length n with generating polynomial $g(X)$. Assume $0 \neq C \neq \mathbf{F}^n$ and $p \nmid n$ (as in the theorem on p. 472).
1. Show that $\deg(g) \geq 1$.
 2. Write $X^n - 1 = g(X)h(X)$. Let α be a primitive n th root of unity. Show that at least one of $1, \alpha, \alpha^2, \dots, \alpha^{n-1}$ is a root of $g(X)$. (You may use the fact that $h(X)$ cannot have more than $\deg(h)$ roots.)
 3. Show that $d(C) \geq 2$.

24.13 Computer Problems

1. Three codewords from the Golay code G_{24} are sent and you receive the vectors

(0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1),
(0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0),
(1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1).

Correct the errors. (The Golay matrix is stored as *golay* and the matrix B is stored in the downloadable computer files (bit.ly/2JbcS6p) as *golaybt*.)

2. An 11-bit message is multiplied by the generating matrix for the Hamming [15, 11] code and the resulting codeword is sent. The vector

(0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0)

is received. Assuming there is at most one error, correct it and determine the original 11-bit message. (The parity check matrix for the Hamming [15, 11] code is stored in the downloadable computer files (bit.ly/2JbcS6p) as *hammingpc*.)

Chapter 25 Quantum Techniques in Cryptography

Quantum computing is a new area of research that has only recently started to blossom. Quantum computing and quantum cryptography were born out of the study of how quantum mechanical principles might be used in performing computations. The Nobel Laureate Richard Feynman observed in 1982 that certain quantum mechanical phenomena could not be simulated efficiently on a classical computer. He suggested that the situation could perhaps be reversed by using quantum mechanics to do computations that are impossible on classical computers. Feynman didn't present any examples of such devices, and only recently has there been progress in constructing even small versions.

In 1994 the field of quantum computing had a significant breakthrough when Peter Shor of AT&T Research Labs introduced a quantum algorithm that can factor integers in (probabilistic) polynomial time (if a suitable quantum computer is ever built). This was a dramatic breakthrough as it presented one of the first examples of a scenario in which quantum techniques might significantly outperform classical computing techniques.

In this chapter we introduce a couple of examples from the area of quantum computing and quantum cryptography. By no means is this chapter a thorough treatment of this young field, for even as we write this chapter significant breakthroughs are being made at NIST and other places, and the field likely will continue to advance rapidly.

There are many books and expository articles being written on quantum computing. One readable account is

[Rieffel-Polak].

25.1 A Quantum Experiment

Quantum mechanics is a difficult subject to explain to nonphysicists since it deals with concepts where our everyday experiences aren't applicable. In particular, the scale at which quantum mechanical phenomena take place is on the atomic level, which is something that can't be observed without special equipment. There are a few examples, however, that are accessible to us, and we now present one such example and use it to develop the mathematical formulation needed to describe some quantum computing protocols.

Since quantum mechanics is a particle-level physics, we need particles that we are able to observe. Photons are the particles that make up light and are therefore observable (similar demonstrations using other particles, such as electrons, can be performed but require more sophisticated equipment).

In order to understand this experiment better, we recommend that you try it at home. Start with a light source and three Polaroid® filters from a camera supply store or three lenses from Polaroid sunglasses.

Label the three filters *A*, *B*, and *C*. Rotate them so that they have the following polarizations: horizontal, 45° , and vertically, respectively (we will explain polarization in more detail after the experiment). Shine the light at the wall and insert filter *A* between the light source and the wall as in [Figure 25.1](#). The photons coming out of the filter will have horizontal polarization. Now insert filter *C* as in [Figure 25.2](#). Since filter *C* has vertical polarization, it filters out all of the horizontally polarized photons from filter *A*. Notice that no light arrives at the wall after this step, the two filters have removed all of the

light components. Now for the final (and most bizarre) step, insert filter *B* in between filter *A* and *C*. You should observe that there is now light arriving at the wall, as depicted in Figure 25.3. This is puzzling, since filter *A* and *C* were enough to remove all of the light, yet the addition of a third filter allows for light to reach the wall.

Figure 25.1 The Photon Experiment with Only Filter A Inserted

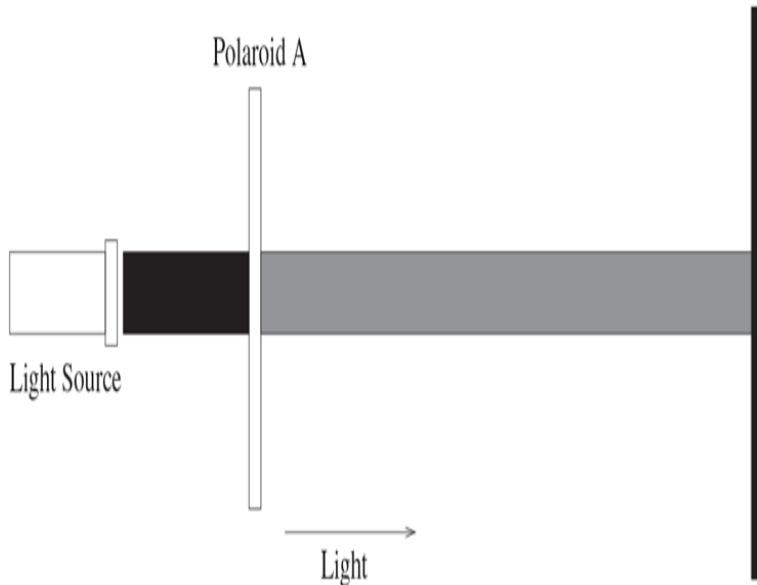


Figure 25.1 Full Alternative Text

Figure 25.2 The Photon Experiment with Filters A and C Inserted

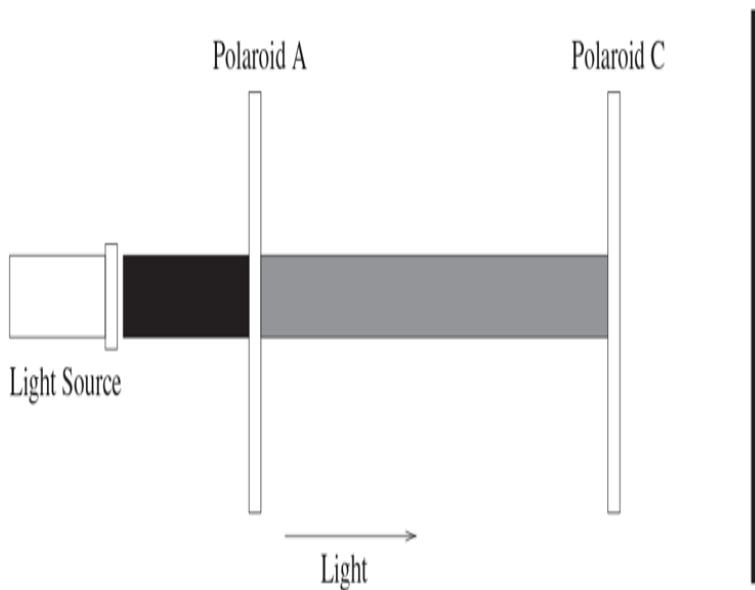


Figure 25.2 Full Alternative Text

Figure 25.3 The Photon Experiment after All Filters Have Been Inserted

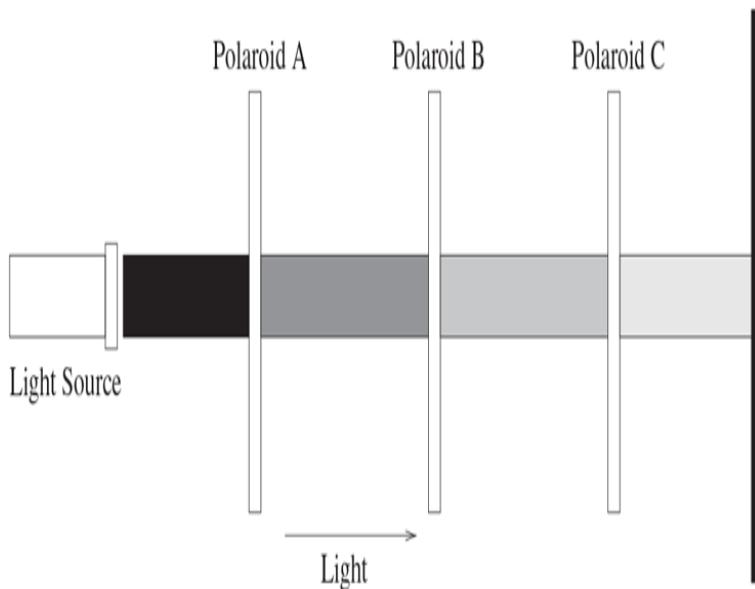


Figure 25.3 Full Alternative Text

In order to explain this demonstration, we need to discuss the concept of polarization of light.

Light is an example of an electromagnetic wave, meaning that it consists of an electric field that travels orthogonally to a corresponding magnetic field. In order to visualize this, consider the light traveling along the x -axis. Now imagine, for example, that the electric field is a wavelike function that lies in the xz -plane. Then the corresponding magnetic field would be a wavelike function in the xy -plane. For such a scenario, the light is referred to as vertically polarized. In general, polarization refers to the direction in which the electric field lies. There is no constraint on this direction.

We will represent a photon's polarization by a unit vector in the two-dimensional complex vector space (however, for our present purposes, real numbers suffice). This vector space has a dot product given by

$(a, b) \cdot (c, d) = a\bar{c} + b\bar{d}$, where \bar{c} and \bar{d} denote the complex conjugates of c and d . The square of the length of a vector (a, b) is then $(a, b) \cdot (a, b) = |a|^2 + |b|^2$. Choose a basis, which we shall denote $|\uparrow\rangle$ and $|\rightarrow\rangle$, for this vector space. We are choosing to use the ket (the second half of “bracket”) notation from physics to represent vectors. We can think of $|\uparrow\rangle$ as being the vertical direction and $|\rightarrow\rangle$ as being horizontal.

Therefore, an arbitrary polarization may be represented as $a|\uparrow\rangle + b|\rightarrow\rangle$, where a and b are complex numbers. Since we are working with unit vectors, the following property holds: $|a|^2 + |b|^2 = 1$. We could just have well chosen a different orthogonal basis, for example, one corresponding to a 45° rotation: $|\nwarrow\rangle$ and $|\nearrow\rangle$.

The Polaroid filters perform a measurement of the polarity of the photon. There are two possible outcomes: Either the photon is aligned with the filter, or it is perpendicular to the direction of the filter. If the vector $a|\uparrow\rangle + b|\rightarrow\rangle$ is measured by a vertical filter, then the probability that the photon has vertical polarity after passing through the filter is $|a|^2$. The probability that it

is measured as having horizontal polarity, and therefore not pass through, is $|b|^2$.

Similarly, suppose we measure a vertically aligned photon with respect to a 45° filter. Since

$$|\uparrow\rangle = \frac{1}{\sqrt{2}}|\nwarrow\rangle + \frac{1}{\sqrt{2}}|\nearrow\rangle,$$

the probability that the photon passes through the filter (which means that it is measured as being aligned at 45°) is $(1/\sqrt{2})^2 = 1/2$. Similarly, the probability that it doesn't pass through the filter (which means that it is measured at -45°) is also $1/2$.

One of the basic principles of quantum mechanics is that such a measurement forces the photon into a definite state. After being measured, the state of the photon will be changed to the result of the measurement. Therefore, if we measured the state of $a|\uparrow\rangle + b|\rightarrow\rangle$ as $|\rightarrow\rangle$, then, from that moment on, the photon will have the state $|\rightarrow\rangle$. If we then measure this photon with a $|\rightarrow\rangle$ filter, we will always observe that the photon is in the $|\rightarrow\rangle$ state; however, if we measure with a $|\uparrow\rangle$ filter, we will never observe that the photon is in the $|\uparrow\rangle$ state.

Let's now explain the interpretation of the experiment. The original light was emitted with random polarization, so only half of the photons being emitted will pass through the $|\rightarrow\rangle$ filter, and these photons will have their state changed to $|\rightarrow\rangle$. The remaining half will be absorbed or reflected and will be changed to $|\uparrow\rangle$. When we place the vertical filter after the horizontal filter, the photons that hit it, which are in state $|\rightarrow\rangle$, will be stopped.

When we insert filter B in the middle, it corresponds to measuring with respect to $|\nearrow\rangle$, and hence those photons that had $|\rightarrow\rangle$ polarity will come out having $|\nearrow\rangle$ polarity with probability $1/2$. Therefore, there has been a $4 : 1$

reduction in the amount of photons passing through up to filter B . Now the $|\nearrow\rangle$ photons pass through the $|\uparrow\rangle$ filter with probability $1/2$ also, and so the total intensity of light arriving at the wall is $1/8$ th the original intensity.

25.2 Quantum Key Distribution

Now that we have set up some of the ideas behind quantum mechanics, we can use them to describe a technique for distributing bits through a quantum channel. These bits can be used to establish a key that can be used for communicating across a classical channel, or any other shared secret.

We begin by describing a quantum bit. Start with a two-dimensional complex vector space. Choose a pair of orthogonal vectors of length 1; call them $|0\rangle$ and $|1\rangle$. For example, these two vectors could be either of the two pairs of orthogonal vectors used in the previous section. A **quantum bit**, also known as a **qubit**, is a unit vector in this vector space. For the purposes of the present discussion, we can think of a qubit as a polarized photon. We have chosen $|0\rangle$ and $|1\rangle$ as notation to conveniently represent the 0 and 1 bits, respectively. The other qubits are linear combinations of these two bits.

Since a qubit is a unit vector, it can be represented as $a|0\rangle + b|1\rangle$, where a and b are complex numbers such that $|a|^2 + |b|^2 = 1$. Just as in the case for photons from the preceding section, we can measure this qubit with respect to the basis $|0\rangle$, $|1\rangle$. The probability that we observe it in the $|0\rangle$ state is $|a|^2$.

Let us now examine how Alice and Bob can communicate with each other in order to establish a message. They will need two things: a quantum channel and a classical channel. A quantum channel is one through which they can exchange polarized photons that are isolated from interactions with the environment (that is, the environment doesn't alter the photons). The classical

channel will be used to send ordinary messages to each other. We assume that the evil observer Eve can observe what is being sent on the classical channel and that she can observe and resend photons on the quantum channel.

Alice starts the establishment of a message by sending a sequence of bits to Bob. They are encoded using a randomly chosen basis for each bit as follows. There are two bases: $B_1 = \{|\uparrow\rangle, |\rightarrow\rangle\}$ and $B_2 = \{|\nwarrow\rangle, |\nearrow\rangle\}$. If Alice chooses B_1 , then she encodes 0 as $|\uparrow\rangle$ and 1 as $|\rightarrow\rangle$, while if she chooses B_2 then she encodes 0 and 1 using the two elements of B_2 .

Each time Alice sends a photon, Bob randomly chooses to measure with respect to either basis B_1 or B_2 . Therefore, for each photon, he obtains an element of that choice of basis as the result of his measurement. Bob records the measurements he has made and keeps them secret. He then tells Alice the basis with which he measured each photon. Alice responds to Bob by telling him which bases were the correct bases for the polarity of the photons that she sent. They keep the bits that used the same bases and discard the other bits. Since two bases were used, Alice and Bob will agree on roughly half of the amount of bits that Alice sent. They can then use these bits as the key for a conventional cryptographic system.

Example

Suppose Alice wants to send the bits 0, 1, 1, 1, 0, 0, 1, 0. She randomly chooses the bases $B_1, B_2, B_1, B_1, B_2, B_2, B_1, B_2$. Therefore, she sends the qubits (photons)

$$|\uparrow\rangle, |\nearrow\rangle, |\rightarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle, |\nwarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle$$

to Bob. He chooses the bases

$B_2, B_2, B_2, B_1, B_2, B_1, B_1, B_2$. He measures the qubits that Alice sent and also tells Alice which bases he used. Alice tells him that the second, fourth, fifth, seventh, and eighth match her choices. These yielded measurements

$$|\nearrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle$$

for Bob, and they correspond to the bits 1, 1, 0, 1, 0. Therefore, both Alice and Bob have the same string 1, 1, 0, 1, 0. They use 11010 as a key for future communication (for example, if they obtained a longer string, they could use the first 128 characters for an AES key).

The security behind quantum key distribution is based upon the laws of quantum mechanics and the fundamental principle that following a measurement of a particle, that particle's state will be altered. Since an eavesdropper Eve must perform measurements in order to observe the photon transmissions between Alice and Bob, Eve will introduce errors in the data that Alice and Bob agreed upon.

Let's see how this happens. Suppose Eve measures the states of the photons transmitted by Alice and allows these measured photons to proceed onto Bob. Since these photons were measured by Eve, they will have the state that Eve observed. Eve will use the wrong basis half of the time when performing the measurement. When Bob performs his measurements, if he uses the correct basis there will be a 25% chance that he will have measured the wrong value.

Let's examine this last statement in more detail. Suppose that Alice sends a photon corresponding to $|\rightarrow\rangle$ and that Bob uses the same basis B_1 as Alice. If Eve uses B_1 , then the photon is passed through correctly and then Bob measures the photon correctly. However, if Eve used B_2 ,

then she will measure $|\nearrow\rangle$ and $|\nwarrow\rangle$ equally likely. The photons that pass to Bob will have one of these orientations and he will therefore half the time measure them correctly as $|\rightarrow\rangle$ and half the time incorrectly. Combining the two possible choices of basis that Eve has causes Bob to have a 25 chance of measuring the incorrect value.

Thus, any eavesdropping introduces a higher error rate in the communication between Alice and Bob. If Alice and Bob test their data for discrepancies over the conventional channel (for example, they could send parity bits), they will detect any eavesdropping.

Actual implementations of this technique have been used to establish keys over distances of more than 100 km using conventional fiber optical cables.

25.3 Shor's Algorithm

Quantum computers are not yet a reality. The current versions can only handle a few qubits. But, if the great technical problems can be overcome and large quantum computers are built, the effect on cryptography will be enormous. In this section we give a brief glimpse at how a quantum computer could factor large integers, using an algorithm developed by Peter Shor. We avoid discussing quantum mechanics and ask the reader to believe that a quantum computer should be able to do all the operations we describe, and do them quickly. For more details, see, for example, [Ekert-Josza] or [Rieffel-Polak].

What is a quantum computer and what does it do? First, let's look at what a classical computer does. It takes a binary input, for example, 100010, and gives a binary output, perhaps 0101. If it has several inputs, it has to work on them individually. A quantum computer takes as input a certain number of qubits and outputs some qubits. The main difference is that the input and output qubits can be linear combinations of certain basic states. The quantum computer operates on all basic states in this linear combination simultaneously. In effect, a quantum computer is a massively parallel machine.

For example, think of the basic state $|100\rangle$ as representing three particles, the first in orientation 1 and the last two in orientation 0 (with respect to some basis that will implicitly be fixed throughout the discussion). The quantum computer can take $|100\rangle$ and produce some output. However, it can also take as input a normalized (that is, of length 1) linear combination of basic quantum states such as

$$\frac{1}{\sqrt{3}}(|100\rangle + |011\rangle + |110\rangle)$$

and produce an output just as quickly as it did when working with a basic state. After all, the computer could not know whether a quantum state is one of the basic states, or a linear combination of them, without making a measurement. But such a measurement would alter the input. It is this ability to work with a linear combination of states simultaneously that makes a quantum computer potentially very powerful.

Suppose we have a function $f(x)$ that can be evaluated for an input x by a classical computer. The classical computer asks for an input and produces an output. A quantum computer, on the other hand, can accept as input a sum

$$\frac{1}{C} \sum_x |x\rangle$$

(C is a normalization factor) of all possible input states and produce the output

$$\frac{1}{C} \sum_x |x, f(x)\rangle,$$

where $|x, f(x)\rangle$ is a longer sequence of qubits, representing both x and the value of $f(x)$. (*Technical point:* It might be notationally better to input $(1/C) \sum |x, 00\cdots\rangle$ in order to have some particles to change to $f(x)$. For simplicity, we will not do this.) So we can obtain a list of all the values of $f(x)$. This looks great, but there is a problem. If you make a measurement, you force the quantum state into the result of the measurement. You get $|x_0, f(x_0)\rangle$ for some randomly chosen x_0 , and the other states in the output are destroyed. So, if you are going to look at the list of values of $f(x)$, you'd better do it carefully, since you get only one chance. In particular, you probably want to apply some transformation to the output in order to put it into a more desirable form. The skill in programming a quantum computer is in designing the computation so that the outputs you want to examine appear with much

higher probability than the others. This is what is done in Shor's factorization algorithm.

25.3.1 Factoring

We want to factor n . The strategy is as follows. Recall that if we can find (nontrivial) a and r with $a^r \equiv 1 \pmod{n}$, then we have a good chance of factoring n (see the factorization method in Subsection 9.4.1). Choose a random a and consider the sequence $1, a, a^2, a^3, \dots \pmod{n}$. If $a^r \equiv 1 \pmod{n}$, then this sequence will repeat every r terms since $a^{j+r} \equiv a^j a^r \equiv a^j \pmod{n}$. If we can measure the period of this sequence (or a multiple of the period), we will have an r such that $a^r \equiv 1 \pmod{n}$. We therefore want to design our quantum computer so that when we make a measurement on the output, we'll have a high chance of obtaining the period.

25.3.2 The Discrete Fourier Transform

We need a technique for finding the period of a periodic sequence. Classically, Fourier transforms can be used for this purpose, and they can be used in the present situation, too. Suppose we have a sequence

$$a_0, a_1, \dots, a_{2^m-1}$$

of length 2^m , for some integer m . Define the Fourier transform to be

$$F(x) = \frac{1}{\sqrt{2^m}} \sum_{c=0}^{2^m-1} e^{\frac{2\pi i c x}{2^m}} a_c,$$

where $0 \leq x < 2^m$.

For example, consider the sequence

1, 3, 7, 2, 1, 3, 7, 2

of length 8 and period 4. The length divided by the period is the frequency, namely 2, which is how many times the sequence repeats. The Fourier transform takes the values

$$\begin{aligned} F(0) &= 26/\sqrt{8}, & F(2) &= (-12 + 2i)/\sqrt{8}, \\ F(4) &= 6/\sqrt{8}, & F(6) &= (-12 - 2i)/\sqrt{8}, \\ F(1) &= F(3) = F(5) = F(7) = 0. \end{aligned}$$

For example, letting $\zeta = e^{2\pi i/8}$, we find that

$$\sqrt{8}F(1) = 1 + 3\zeta + 7\zeta^2 + 2\zeta^3 + \zeta^4 + 3\zeta^5 + 7\zeta^6 + 2\zeta^7.$$

Since $\zeta^4 = -1$, the terms cancel and we obtain $F(1) = 0$. The nonzero values of F occur at multiples of 2, which is the frequency.

Let's consider another example: 2, 1, 2, 1, 2, 1, 2, 1.

The Fourier transform is

$$\begin{aligned} F(0) &= 12/\sqrt{8}, & F(4) &= 4/\sqrt{8}, \\ F(1) &= F(2) = F(3) = F(5) = F(6) = F(7) = 0. \end{aligned}$$

Here the nonzero values of F are again at the multiples of the frequency.

In general, if the period is a divisor of 2^m , then all the nonzero values of F will occur at multiples of the frequency (however, a multiple of the frequency could still yield 0). See [Exercise 2](#).

Suppose now that the period isn't a divisor of 2^m . Let's look at an example. Consider the sequence

1, 0, 0, 1, 0, 0, 1, 0. It has length 8 and almost has period 3 and frequency 3, but we stopped the sequence before it had a chance to complete the last period. In [Figure 25.4](#), we graph the absolute value of its Fourier transform (these are real numbers, hence easier to graph than the complex values of the Fourier transform). Note that there are peaks at 0, 3, and 5. If we continued $F(x)$

to larger values of x we would get peaks at 8, 11, 13, 16, The peaks are spaced at an average distance of $8/3$. Dividing the length of the sequence by the average distance yields a period of $8/(8/3) = 3$, which agrees with our intuition.

Figure 25.4 The Absolute Value of a Discrete Fourier Transform

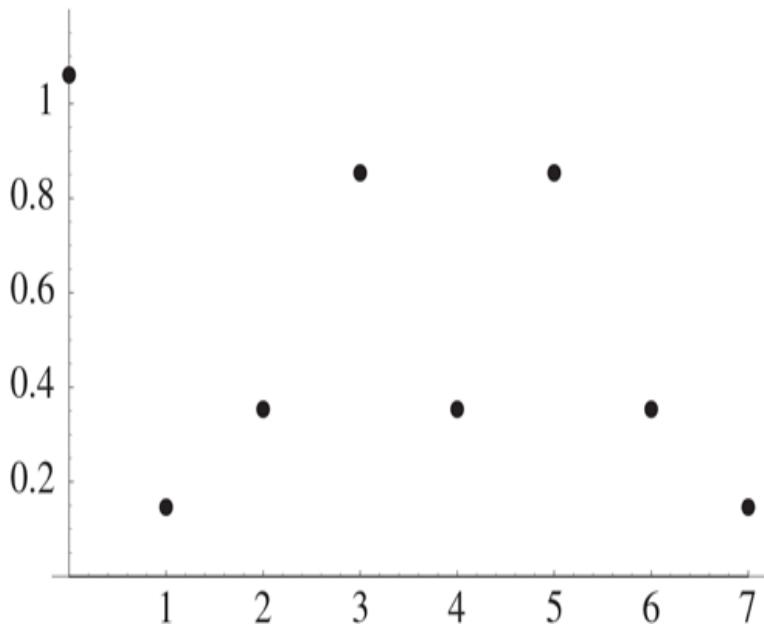


Figure 25.4 Full Alternative Text

The fact that there is a peak at 0 is not very surprising. The formula for the Fourier transform shows that the value at 0 is simply the sum of the elements in the sequence divided by the square root of the length of the sequence.

Let's look at one more example: 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1. This sequence has 16 terms. Our intuition might say that the period is around 5 and the frequency is slightly more than 3. [Figure 25.5](#) shows the graph of the absolute value of its Fourier transform. Again, the

peaks are spaced around 3 apart, so we can say that the frequency is around 3. The period of the original sequence is therefore around 5, which agrees with our intuition.

Figure 25.5 The Absolute Value of a Discrete Fourier Transform

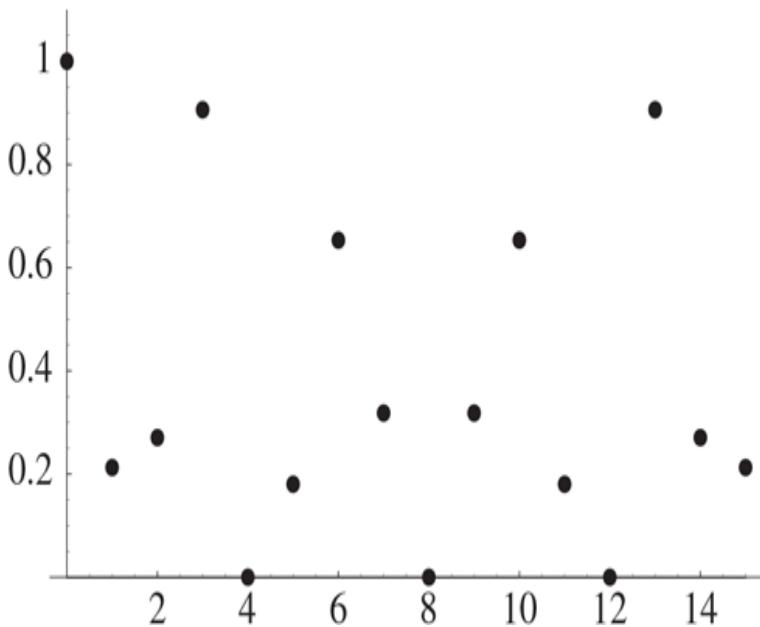


Figure 25.5 Full Alternative Text

In the first two examples, the period was a divisor of the length (namely, 8) of the sequence. We obtained nonzero values of the Fourier transform only at multiples of the frequency. In these last two examples, the period was not a divisor of the length (8 or 16) of the sequence. This introduced some “noise” into the situation. We had peaks at approximate multiples of the frequency and values close to 0 away from these peaks.

The conclusion is that the peaks of the Fourier transform occur approximately at multiples of the frequency, and

the period is approximately the number of peaks. This will be useful in Shor's algorithm.

25.3.3 Shor's Algorithm

Choose m so that $n^2 \leq 2^m < 2n^2$. We start with m qubits, all in state 0:

$$|000000000\rangle.$$

As in the previous section, by changing axes, we can transform the first bit to a linear combination of $|0\rangle$ and $|1\rangle$, which gives us

$$\frac{1}{\sqrt{2}}(|000000000\rangle + |100000000\rangle).$$

We then successively do a similar transformation to the second bit, the third bit, up through the m th bit, to obtain the quantum state

$$\frac{1}{\sqrt{2^m}}(|000000000\rangle + |000000001\rangle + |000000010\rangle + \cdots + |111111111\rangle).$$

Thus all possible states of the m qubits are superimposed in this sum. For simplicity of notation, we replace each string of 0s and 1s with its decimal equivalent, so we write

$$\frac{1}{\sqrt{2^m}}(|0\rangle + |1\rangle + |2\rangle + \cdots + |2^m - 1\rangle).$$

Choose a random number a with $1 < a < n$. We may assume $\gcd(a, n) = 1$; otherwise, we have a factor of n . The quantum computer computes the function $f(x) = a^x \pmod{n}$ for this quantum state to obtain

$$\frac{1}{\sqrt{2^m}}(|0, a^0\rangle + |1, a^1\rangle + |2, a^2\rangle + \cdots + |2^m - 1, a^{2^m-1}\rangle)$$

(for ease of notation, a^x is used to denote $a^x \pmod{n}$). This gives a list of all the values of a^x . However, so far we are not any better off than with a classical computer.

If we measure the state of the system, we obtain a basic state $|x_0, a^{x_0}\rangle$ for some randomly chosen x_0 . We cannot even specify which x_0 we want to use. Moreover, the system is forced into this state, obliterating all the other values of a^x that have been computed. Therefore, we do not want to measure the whole system. Instead, we measure the value of the second half. Each basic piece of the system is of the form $|x, a^x\rangle$, where x represents m bits and a^x is represented by $m/2$ bits (since $a^x \pmod{n} < n < 2^{m/2}$). If we measure these last $m/2$ bits, we obtain some number $u \pmod{n}$, and the whole system is forced into a combination of those states of the form $|x, u\rangle$ with $a^x \equiv u \pmod{n}$:

$$\frac{1}{C} \sum_{\substack{0 \leq x \leq 2^m \\ a^x \equiv u \pmod{n}}} |x, u\rangle,$$

where C is whatever factor is needed to make the vector have length 1 (in fact, C is the square root of the number of terms in the sum).

Example

At this point, it is probably worthwhile to have an example. Let $n = 21$. (This example might seem simple, but it is the largest that quantum computers using Shor's algorithm can currently handle. Other algorithms are being developed that can go somewhat farther.) Since $21^2 < 2^9 < 2 \cdot 21^2$, we have $m = 9$. Let's choose $a = 11$, so we compute the values of $11^x \pmod{21}$ to obtain

$$\begin{aligned} \frac{1}{\sqrt{512}} (&|0, 1\rangle + |1, 11\rangle + |2, 16\rangle + |3, 8\rangle + |4, 4\rangle + |5, 2\rangle + |6, 1\rangle + |7, 11\rangle + \\ &|8, 16\rangle + |9, 8\rangle + |10, 4\rangle + |11, 2\rangle + |12, 1\rangle + |13, 11\rangle + |14, 16\rangle + \\ &|15, 8\rangle + |16, 4\rangle + |17, 2\rangle + |18, 1\rangle + |19, 11\rangle + |20, 16\rangle + \dots \\ &+ |508, 4\rangle + |509, 2\rangle + |510, 1\rangle + |511, 11\rangle). \end{aligned}$$

Suppose we measure the second part and obtain 2. This means we have extracted all the terms of the form $|x, 2\rangle$

to obtain

$$\frac{1}{\sqrt{85}}(|5, 2\rangle + |11, 2\rangle + |17, 2\rangle + |23, 2\rangle + \dots + |497, 2\rangle + |503, 2\rangle + |509, 2\rangle).$$

For notational convenience, and since it will no longer be needed, we drop the second part to obtain

$$\frac{1}{\sqrt{85}}(|5\rangle + |11\rangle + |17\rangle + |23\rangle + \dots + |497\rangle + |503\rangle + |509\rangle).$$

If we now measured this system, we would simply obtain a number x such that $11^x \equiv 2 \pmod{21}$. This would not be useful.

Suppose we could take two measurements. Then we would have two numbers x and y with $11^x \equiv 11^y \pmod{21}$. This would yield $11^{x-y} \equiv 1 \pmod{21}$. By the factorization method of Subsection 9.4.1, this would give us a good chance of being able to factor 21. However, we cannot take two independent measurements. The first measurement puts the system into the output state, so the second measurement would simply give the same answer as the first.

Not all is lost. Note that in our example, the numbers in our state are periodic with period 6. In general, the values of $a^x \pmod{n}$ are periodic with period r , with $a^r \equiv 1 \pmod{n}$. So suppose we are able to make a measurement that yields the period. We then have a situation where $a^r \equiv 1 \pmod{n}$, so we can hope to factor n by the method from Subsection 9.4.1 mentioned above.

The **quantum Fourier transform** is exactly the tool we need. It measures frequencies, which can be used to find the period. If r happens to be a divisor of 2^m , then the frequencies we obtain are multiples of a fundamental frequency f_0 , and $rf_0 = 2^m$. In general, r is not a divisor of 2^m , so there will be some dominant

frequencies, and they will be approximate multiples of a fundamental frequency f_0 with $rf_0 \approx 2^m$. This will be seen in the analysis of our example and in Figure 25.6.

Figure 25.6 The Absolute Value of $g(c)$

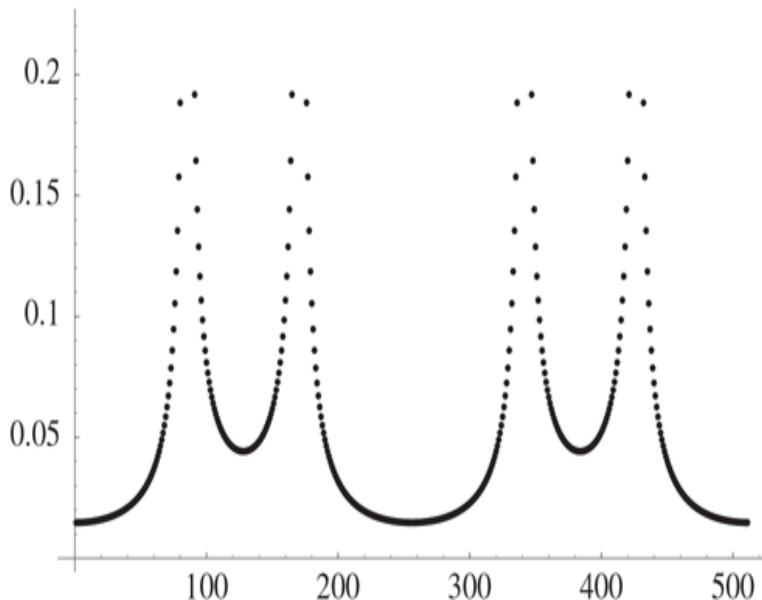


Figure 25.6 Full Alternative Text

The quantum Fourier transform is defined on a basic state $|x\rangle$ (with $0 \leq x < 2^m$) by

$$QFT(|x\rangle) = \frac{1}{\sqrt{2^m}} \sum_{c=0}^{2^m-1} e^{\frac{2\pi i cx}{2^m}} |c\rangle.$$

It extends to a linear combination of states by linearity:

$$QFT(a_1|x_1\rangle + \cdots + a_t|x_t\rangle) = a_1QFT(|x_1\rangle) + \cdots + a_tQFT(|x_t\rangle).$$

We can therefore apply $QF <$ to our quantum state.

In our example, we compute

$$QFT < \left(\frac{1}{\sqrt{85}} (|5\rangle + |11\rangle + |17\rangle + |23\rangle + \dots + |497\rangle + |503\rangle + |509\rangle) \right)$$

and obtain a sum

$$\frac{1}{\sqrt{85}} \sum_{c=0}^{511} g(c) |c\rangle$$

for some numbers $g(c)$.

The number $g(c)$ is given by

$$g(c) = \frac{1}{\sqrt{512}} \sum_{\substack{0 \leq x < 512 \\ x \equiv 5 \pmod{6}}} e^{\frac{2\pi i cx}{512}},$$

which is the discrete Fourier transform of the sequence

$$0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, \dots, 0, 0, 0, 0, 0, 1, 0, 0.$$

Therefore, the peaks of the graph of the absolute value of g should correspond to the frequency of the sequence, which should be around $512/6 \approx 85$. The graph in Figure 25.6 is a plot of $|g|$.

There are sharp peaks at $c = 0, 85, 171, 256, 341, 427$ (the ones at 0 and 256 do not show up on the graph since they are centered at one value; see below). These are the dominant frequencies mentioned previously. The values of g near the peak at $c = 341$ are

338	339	340	341	342	343	344	345
0.305	0.439	0.773	3.111	1.567	0.631	0.398	0.291

The behavior near $c = 85, 171$, and 427 is similar. At $c = 0$ and 256 , we have $g(0) = 3.756$, while all the nearby values of c have $g(c) \approx 0.015$.

The peaks are approximately at multiples of the fundamental frequency $f_0 = 85$. Of course, we don't really know this yet, since we haven't made any measurements.

Now we measure the quantum state of this Fourier transform. Recall that if we start with a linear combination of states $a_1|x_1\rangle + \dots + a_{<}|x_{<}\rangle$ normalized such that $\sum |a_j|^2 = 1$, then the probability of obtaining $|x_k\rangle$ is $|a_k|^2$. More generally, if we don't assume $\sum |a_j|^2 = 1$, the probability is

$$|a_k|^2 / \sum |a_j|^2.$$

In our example,

$$3.111^2 / \sum |a_j|^2 \approx .114,$$

so if we sample the Fourier transform, the probability is around $4 \times .114 = .456$ that we obtain one of $c = 85, 171, 341, 427$. Let's suppose this is the case; say we get $c = 427$. We know, or at least expect, that 427 is approximately a multiple of the frequency f_0 that we're looking for:

$$427 \approx j f_0$$

for some j . Since $r f_0 \approx 2^m = 512$, we divide to obtain

$$\frac{427}{512} \approx \frac{j}{r}.$$

Note that $427/512 \approx .834 \approx 5/6$. Since we must have $r \leq \phi(21) < 21$, a reasonable guess is that $r = 6$ (see the following discussion of continued fractions).

In general, Shor showed that there is a high chance of obtaining a value of $c/2^m$ with

$$\left| \frac{c}{2^m} - \frac{j}{r} \right| < \frac{1}{2^{m+1}} < \frac{1}{2n^2},$$

for some j . The method of continued fractions will find the unique (see [Exercise 3](#)) value of j/r with $r < n$ satisfying this inequality.

In our example, we take $r = 6$ and check that $a^r = 11^6 \equiv 1 \pmod{21}$.

We want to use the factorization method of Subsection 9.4.1 to factor 21. Recall that this method writes

$r = 2^k m$ with m odd, and then computes

$b_0 \equiv a^m \pmod{n}$. We then successively square b_0 to get b_1, b_2, \dots , until we reach $1 \pmod{n}$. If b_u is the last $b_i \not\equiv 1 \pmod{n}$, we compute $\gcd(b_u - 1, n)$ to get a factor (possibly trivial) of n .

In our example, we write $6 = 2 \cdot 3$ (a power of 2 times an odd number) and compute (in the notation of Subsection 9.4.1)

$$\begin{aligned} b_0 &\equiv 11^3 \equiv 8 \pmod{21} \\ b_1 &\equiv 11^6 \equiv 1 \pmod{21} \end{aligned}$$

$$\gcd(b_0 - 1, 21) = \gcd(7, 21) = 7,$$

so we obtain $21 = 7 \cdot 3$.

In general, once we have a candidate for r , we check that $a^r \equiv 1 \pmod{n}$. If not, we were unlucky, so we start over with a new a and form a new sequence of quantum states. If $a^r \equiv 1 \pmod{n}$, then we use the factorization method from Subsection 9.4.1. If this fails to factor n , start over with a new a . It is very likely that, in a few attempts, a factorization of n will be found.

We now say more about continued fractions. In Chapter 3, we outlined the method of continued fractions for finding rational numbers with small denominator that approximate real numbers. Let's apply the procedure to the real number $427/512$. We have

$$\frac{427}{512} = 0 + \cfrac{1}{1 + \cfrac{1}{5 + \cfrac{1}{42 + \cfrac{1}{2}}}}.$$

This yields the approximating rational numbers

$$0, \quad 1, \quad \frac{5}{6}, \quad \frac{211}{253}, \quad \frac{427}{512}.$$

Since we know the period in our example is less than $n = 21$, the best guess is the last denominator less than n , namely $r = 6$.

In general, we compute the continued fraction expansion of $c/2^m$, where c is the result of the measurement. Then we compute the approximations, as before. The last denominator less than n is the candidate for r .

25.3.4 Final Words

The capabilities of quantum computers and quantum algorithms are of significant importance to economic and government institutions. Many secrets are protected by cryptographic protocols. Quantum cryptography's potential for breaking these secrets as well as its potential for protecting future secrets has caused this new research field to grow rapidly over the past few years.

Although the first full-scale quantum computer is probably many years off, and there are still many who are skeptical of its possibility, quantum cryptography has already succeeded in transmitting secure messages over distances of more than 100 km, and quantum computers have been built that can handle a (very) small number of qubits. Quantum computation and cryptography have already changed the manner in which computer scientists and engineers perceive the capabilities and limits of the computer. Quantum computing has rapidly become a popular interdisciplinary research area and promises to offer many exciting new results in the future.

25.4 Exercises

1. Consider the sequence $2^0, 2^1, 2^2, \dots \pmod{15}$.
 1. What is the period of this sequence?
 2. Suppose you want to use Shor's algorithm to factor $n = 15$. What value of m would you take?
 3. Suppose the measurement in Shor's algorithm yields $c = 192$. What value do you obtain for r ? Does this agree with part (a)?
 4. Use the value of r from part (c) to factor 15.
2. Let $0 < s \leq m$. Fix an integer c_0 with $0 \leq c_0 < 2^s$. Show that

$$\sum_{\substack{0 \leq c < 2^m \\ c \equiv c_0 \pmod{2^s}}} e^{\frac{2\pi i cx}{2^m}} = 0$$

if $x \not\equiv 0 \pmod{2^{m-s}}$ and $= 2^{m-s} e^{2\pi i xc_0/2^m}$ if $x \equiv 0 \pmod{2^{m-s}}$. (Hint: Write $c = c_0 + j2^s$ with $0 \leq j < 2^{m-s}$, factor $e^{2\pi i xc_0/2^m}$ off the sum, and recognize what's left as a geometric sum.)

2. Suppose $a_0, a_1, \dots, a_{2^m-1}$ is a sequence of length 2^m such that $a_k = a_{k+j2^s}$ for all j, k . Show that the Fourier transform $F(x)$ of this sequence is 0 whenever $x \not\equiv 0 \pmod{2^{m-s}}$.

This shows that if the period of a sequence is a divisor of 2^m then all the nonzero values of F occur at multiples of the frequency (namely, 2^{m-s}).

3. 1. Suppose j/r and j_1/r_1 are two distinct rational numbers, with $0 < r < n$ and $0 < r_1 < n$. Show that

$$\left| \frac{j_1}{r_1} - \frac{j}{r} \right| g > \frac{1}{n^2}.$$

2. Suppose, as in Shor's algorithm, that we have

$$\left| \frac{c}{2^m} - \frac{j}{r} \right| < \frac{1}{2n^2} \text{ and } \left| \frac{c}{2^m} - \frac{j_1}{r_1} \right| < \frac{1}{2n^2}.$$

Show that $j/r = j_1/r_1$.

Appendix A Mathematica® Examples

These computer examples are written in Mathematica. If you have Mathematica available, you should try some of them on your computer. If Mathematica is not available, it is still possible to read the examples. They provide examples for several of the concepts of this book. For information on getting started with Mathematica, see [Section A.1](#). To download a Mathematica notebook that contains these commands, go to

bit.ly/2u5R7dW

A.1 Getting Started with Mathematica

1. Download the Mathematica notebook crypto.nb that you find using the links starting at bit.ly/2u5R7dW
2. Open Mathematica, and then open crypto.nb using the menu options under File on the command bar at the top of the Mathematica window. (Perhaps this is done automatically when you download it; it depends on your computer settings.)
3. With crypto.nb in the foreground, click (left button) on Evaluation on the command bar. A menu will appear. Move the arrow down to the line Evaluate Notebook and click (left button). This evaluates the notebook and loads the necessary functions. Ignore any warning messages about spelling. They occur because a few functions have similar names.
4. Go to the command bar at the top and click on File. Move the arrow down to New and left click. Then left click on Notebook. A new notebook will appear on top of crypto.nb. However, all the commands of crypto.nb will still be working.
5. If you want to give the new notebook a name, use the File command and scroll down to Save As.... Then save under some name with a .nb at the end.
6. You are now ready to use Mathematica. If you want to try something easy, type $1 + 2^*3 + 4^5$ and then press the Shift and Enter keys simultaneously. Or, if your keyboard has a number pad with Enter, probably on the right side of the keyboard, you can press that (without the Shift). The result 1031 should appear (it's $1 + 2 \cdot 3 + 4^5$).
7. Turn to the Computer Examples Section A.3. Try typing in some of the commands there. The outputs should be the same as that in the examples. Remember to press Shift Enter (or the numeric Enter) to make Mathematica evaluate an expression.
8. If you want to delete part of your notebook, simply move the arrow to the line at the right edge of the window and click the left button. The highlighted part can be deleted by clicking on Edit on the top command bar, then clicking on Cut on the menu that appears.
9. Save your notebook by clicking on File on the command bar, then clicking on Save on the menu that appears.

10. Print your notebook by clicking on File on the command bar, then clicking on Print on the menu that appears. (You will see the advantage of opening a new notebook in Step 4; if you didn't open one, then all the commands in crypto.nb will also be printed.)
11. If you make a mistake in typing in a command and get an error message, you can edit the command and hit Shift Enter to try again. You don't need to retype everything.
12. Look at the commands available through the command bar at the top. For example, Format then Style allows you to change the type font on any cell that has been highlighted (by clicking on its bar on the right side).
13. If you are looking for help or a command to do something, try the Help command. Note that the commands that are built into Mathematica always start with capital letters. The commands that are coming from crypto.nb start with small letters and will not be found via Help.

A.2 Some Commands

The following are some Mathematica commands that are used in the Computer Examples. The commands that start with capital letters, such as `EulerPhi`, are built into Mathematica. The ones that start with small letters, such as `addell`, have been written specially for this text and are in the Mathematica notebook available at

bit.ly/2u5R7dW

`addell[{x,y}, {u,v}, b, c, n]` finds the sum of the points $\{x, y\}$ and $\{u, v\}$ on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod{n}$, where n is odd.

`affinecrypt[txt, m, n]` affine encryption of `txt` using $mx + n$.

`allshifts[txt]` gives all 26 shifts of `txt`.

`ChineseRemainder[{a, b, ...}, {m, n, ...}]` gives a solution to the simultaneous congruences $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$, ...

`choose[txt, m, n]` lists the characters in `txt` in positions congruent to $n \pmod{m}$.

`coinc[txt, n]` the number of matches between `txt` and `txt` displaced by n .

`corr[v]` the dot product of the vector v with the 26 shifts of the alphabet frequency vector.

`EulerPhi[n]` computes $\phi(n)$ (don't try very large values of n).

`ExtendedGCD[m, n]` computes the gcd of m and n along with a solution of $mx + ny = \text{gcd}$.

`FactorInteger[n]` factors n .

`frequency[txt]` lists the number of occurrences of each letter a through z in `txt`.

`GCD[m, n]` is the gcd of m and n .

`Inverse[M]` finds the inverse of the matrix M .

`lfsr[c, k, n]` gives the sequence of n bits produced by the recurrence that has coefficients given by the vector c . The initial values of the bits are given by the vector k .

`lfsrlength[v, n]` tests the vector v of bits to see if it is generated by a recurrence of length at most n .

`lfsrsolve[v, n]` given a guess n for the length of the recurrence that generates the binary vector v , it computes the coefficients of the recurrence.

`Max[v]` is the largest element of the vector v .

`Mod[a, n]` is the value of $a \pmod{n}$.

`multell[{x, y}, m, b, c, n]` computes m times the point $\{x, y\}$ on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod{n}$.

`multsell[{x, y}, m, b, c, n]` lists the first m multiples of the point $\{x, y\}$ on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod{n}$.

`NextPrime[x]` gives the next prime $> x$.

`num2text0[n]` changes a number n to letters. The successive pairs of digits must each be at most 25; a is oo, z is 25.

`num2text[n]` changes a number n to letters. The successive pairs of digits must each be at most 26; *space* is oo, a is 01, z is 26.

`PowerMod[a,b,n]` computes $a^b \pmod{n}$.

`PrimitiveRoot[p]` finds a primitive root for the prime p .

`shift[txt,n]` shifts `txt` by n .

`txt2num0[txt]` changes `txt` to numbers, with $a = 00, \dots, z = 25$.

`txt2num[txt]` changes `txt` to numbers, with *space* = oo, $a = 01, \dots, z = 26$.

`vigenere[txt,v]` gives the Vigenère encryption of `txt` using the vector v .

`vigvec[txt,m,n]` gives the frequencies of the letters a through z in positions congruent to $n \pmod{m}$.

A.3 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext kddmu.
Decrypt it by trying all possibilities.

```
In[1]:= allshifts["kddkmu"]
```

```
kddkmu
leelnv
mffmow
nggnpx
ohhoqy
piiprz
qjjqsa
rkkrtb
sllsuc
tmmtvd
unnuwe
voovxf
wppwyg
xqqxzh
yrryai
zsszbj
attack
buubdl
cvvcem
dwwdfn
exxego
fyyfhp
gzzgiq
haahjr
ibbiks
jccjlt
```

As you can see, *attack* is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message *cleopatra* using the affine function $7x + 8$:

```
In[2]:=affinecrypt["cleopatra", 7, 8]
```

```
Out[2]=whkcjilxi
```

Example 3

The ciphertext *mzdvezc* was encrypted using the affine function $5x + 12$. Decrypt it.

SOLUTION

First, solve $y \equiv 5x + 12 \pmod{26}$ for x to obtain $x \equiv 5^{-1}(y - 12)$. We need to find the inverse of 5 (mod 26):

```
In[3]:=PowerMod[5, -1, 26]
```

```
Out[3]=21
```

Therefore, $x \equiv 21(y - 12) \equiv 21y - 12 \cdot 21$. To change $-12 \cdot 21$ to standard form:

```
In[4]:=Mod[-12*21, 26]
```

```
Out[4]=8
```

Therefore, the decryption function is $x \equiv 21y + 8$. To decrypt the message:

```
In[5]:=affinecrypt["mzdvezc", 21, 8]
```

```
Out[5]=anthony
```

In case you were wondering, the plaintext was encrypted as follows:

```
In[6]:= affinecrypt["anthony", 5, 12]
```

```
Out[6]= mzdvezc
```

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in [Section 2.3](#) to decrypt it. For convenience, we've already stored the ciphertext under the name *vvhq*.

```
In[7]:= vvhq
```

```
Out[7]=
```

```
vvhqvvvrhmustgjgthkihtssejchlsfcbgvwcrlyqtfsvgahw  
kcuuhwauglqhnslrljs  
hbltspisprdxljsveeghlqwasskuwepwqtwspsgoelkcqyfn  
svwljsniqkgnrgybw1  
wgovioikhkazkqkxzgyhcecmeliujojqkwfwefqhkijrclrlkbi  
enqfrjljsdhgrhlsfq  
twlauqrhwdmwlgusgikkflryvcwvsgpmlkassjvoqxeggvey  
ggzmljcxxljspaivw  
ikvrdrygfrjlslveggveyggeiapuuuisfpbtgnwwmuczrvtwg  
lrwugumnczvile
```

Find the frequencies of the letters in the ciphertext:

```
In[8]:= frequency[vvhq]
```

```
Out[8]=
```

```
 {{a, 8}, {b, 5}, {c, 12}, {d, 4}, {e, 15}, {f, 10}, {g, 27}, {h, 16}, {i, 13}, {j, 14}, {k, 17}, {l, 25}, {m, 7}, {n, 7}, {o, 5}, {p, 9}, {q, 14}, {r, 17}, {s, 24}, {t, 8}, {u, 12}, {v, 22}, {w, 22}, {x, 5}, {y, 8}, {z, 5}}
```

Let's compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

In[9]:= `coinc[vvhq, 1]`

Out[9]= 14

In[10]:= `coinc[vvhq, 2]`

Out[10]= 14

In[11]:= `coinc[vvhq, 3]`

Out[11]= 16

In[12]:= `coinc[vvhq, 4]`

Out[12]= 14

In[13]:= `coinc[vvhq, 5]`

Out[13]= 24

In[14]:= `coinc[vvhq, 6]`

Out[14]= 12

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to 1 mod 5):

In[15]:= `choose[vvhq, 5, 1]`

Out[15]=

vvutccqgcunjtpjgkuqpknjkygkkgcjfqrkqjrqu
dukvpkvggjjivgjggpfncwuce

In[16]:= `frequency[%]`

```
Out[16]= {{a, 0}, {b, 0}, {c, 7}, {d, 1},  
{e, 1}, {f, 2}, {g, 9}, {h, 0}, {i, 1},  
{j, 8}, {k, 8}, {l, 0}, {m, 0}, {n, 3},  
{o, 0}, {p, 4}, {q, 5}, {r, 2}, {s, 0},  
{t, 3}, {u, 6}, {v, 5}, {w, 1}, {x, 0},  
{y, 1}, {z, 0}}
```

To express this as a vector of frequencies:

```
In[17]:= vigvec[vvhq, 5, 1]
```

```
Out[17]= {0, 0, 0.104478, 0.0149254,  
0.0149254, 0.0298507, 0.134328, 0,  
0.0149254, 0.119403, 0.119403, 0, 0,  
0.0447761, 0, 0.0597015, 0.0746269,  
0.0298507, 0, 0.0447761, 0.0895522,  
0.0746269, 0.0149254, 0, 0.0149254, 0}
```

The dot products of this vector with the displacements of the alphabet frequency vector are computed as follows:

```
In[18]:= corr[%]
```

```
Out[18]=
```

```
{0.0250149, 0.0391045, 0.0713284, 0.0388209,  
0.0274925, 0.0380149, 0.051209, 0.0301493,  
0.0324776, 0.0430299, 0.0337761, 0.0298507,  
0.0342687, 0.0445672, 0.0355522, 0.0402239,  
0.0434328, 0.0501791, 0.0391791, 0.0295821,  
0.0326269, 0.0391791, 0.0365522, 0.0316119,  
0.0488358, 0.0349403}
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

```
In[19]:= Max[%]
```

```
Out[19]= 0.0713284
```

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter *c*. A procedure similar to the one just used (using *vigvec[vvhq, 5,2],..., vigvec[vvhq,5,5]*) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

In[20]:= *vigenere[vvhq, -{2, 14, 3, 4, 18}]*

Out[20]=

```
themethodusedfortheprparationandreadingofcodemes
sagesissimpleinthe
extremeandalthesametimeimpossibleoftranslationunl
essthekeyisknownth
eeasewithwhichthekeymaybechangedisanotherpointinf
avoroftheadoptiono
fthiscodebythosedesiringtotransmitimportantmessag
eswithouttheslight
estdangeroftheirmessagesbeingreadbypoliticalorbus
inessrivalsetc
```

For the record, the plaintext was originally encrypted by the command

In[21]:= *vigenere[% , {2, 14, 3, 4, 18}]*

Out[21]=

```
vvhqvvrhmustgjgthkihtssejchlsfcbgvwcrlyqtfsvgahw
kcuwauglqhnsrljs
hbltspisprdxljsveeghlqwkasskuwepwqtvwspgoelkcqyfn
svwljsniqkgnrgybw1
wgoviokhkazkqkxzgyhcecmeyiujoqkwfwefqhkijrcrlkbi
enqfrjljsdhgrhlsfq
twlauqrhwdmwlgusgikkflryvcwvsgpmlkassjvoqxeggvey
ggzmljcxxljspaivw
ikvrdrygfrjljslveggveyggeiapuuuisfpbtgnwwwmuczrvtwg
lrwugumnczvile
```

A.4 Examples for Chapter 3

Example 5

Find gcd (23456, 987654).

```
In[1]:= GCD[23456, 987654]
```

```
Out[1]= 2
```

Example 6

Solve $23456x + 987654y = 2$ in integers x, y .

```
In[2]:= ExtendedGCD[23456, 987654]
```

```
Out[2]= {2, {-3158, 75}}
```

This means that 2 is the gcd and
 $23456 \cdot (-3158) + 987654 \cdot 75 = 2$.

Example 7

Compute $234 \cdot 456 \pmod{789}$.

```
In[3]:= Mod[234*456, 789]
```

```
Out[3]= 189
```

Example 8

Compute $234567^{876543} \pmod{565656565}$.

```
In[4]:= PowerMod[234567, 876543, 565656565]
```

```
Out[4]= 473011223
```

Example 9

Find the multiplicative inverse of
 $87878787 \pmod{9191919191}$.

```
In[5]:= PowerMod[87878787, -1, 9191919191]
```

```
Out[5]= 7079995354
```

Example 10

Solve $7654x \equiv 2389 \pmod{65537}$.

SOLUTION

Here is one way. It corresponds to the method in [Section 3.3](#). We calculate 7654^{-1} and then multiply it by 2389:

```
In[6]:= PowerMod[7654, -1, 65537]
```

```
Out[6]= 54637
```

```
In[7]:= Mod[%*2389, 65537]
```

```
Out[7]= 43626
```

Example 11

Find x with

$$x \equiv 2 \pmod{78}, x \equiv 5 \pmod{97}, x \equiv 1 \pmod{119}.$$

SOLUTION

```
In[8]:= ChineseRemainder[{2, 5, 1}, {78, 97, 119}]
```

```
Out[8]= 647480
```

We can check the answer:

```
In[9]:= Mod[647480, {78, 97, 119}]
```

```
Out[9]= {2, 5, 1}
```

Example 12

Factor 123450 into primes.

```
In[10]:= FactorInteger[123450]
Out[10]= {{2, 1}, {3, 1}, {5, 2}, {823, 1}}
```

This means that $123450 = 2^1 3^1 5^2 823^1$.

Example 13

Evaluate $\phi(12345)$.

```
In[11]:= EulerPhi[12345]
```

```
Out[11]= 6576
```

Example 14

Find a primitive root for the prime 65537.

```
In[12]:= PrimitiveRoot[65537]
```

```
Out[12]= 3
```

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix

$$\begin{pmatrix} 13 & 12 & 35 \\ 41 & 53 & 62 \\ 71 & 68 & 10 \end{pmatrix} \pmod{999}.$$

SOLUTION

First, invert the matrix without the mod:

```
In[13]:= Inverse[{{13, 12, 35}, {41, 53, 62}, {71, 68, 10}}]
```

$$\text{Out[13]}= \left\{ \left\{ \frac{3686}{34139}, -\frac{2260}{34139}, \frac{1111}{34139} \right\}, \left\{ -\frac{3992}{34139}, \frac{2355}{34139}, -\frac{629}{34139} \right\}, \left\{ \frac{975}{34139}, \frac{32}{34139}, -\frac{197}{34139} \right\} \right\}$$

We need to clear the 34139 out of the denominator, so we evaluate $1/34139 \pmod{999}$:

```
In[14]:= PowerMod[34139, -1, 999]
```

```
Out[14]= 410
```

Since $410 \cdot 34139 \equiv 1 \pmod{999}$, we multiply the inverse matrix by $410 \cdot 34139$ and reduce mod 999 in order to remove the denominators without changing anything mod 999:

```
In[15]:= Mod[410*34139*%%, 999]
```

```
Out[15]= {{772, 472, 965}, {641, 516, 851}, {150, 133, 149}}
```

Therefore, the inverse matrix mod 999 is

$$\begin{pmatrix} 772 & 472 & 965 \\ 641 & 516 & 851 \\ 150 & 133 & 149 \end{pmatrix}.$$

In many cases, it is possible to determine by inspection the common denominator that must be removed. When this is not the case, note that the determinant of the original matrix will always work as a common denominator.

Example 16

Find a square root of $26951623672 \pmod{98573007539}$.

SOLUTION

Since $p \equiv 3 \pmod{4}$, we can use the Proposition of Section 3.9:

```
In[16]:= PowerMod[26951623672, (98573007539 + 1)/4, 98573007539]
```

```
Out[16]= 98338017685
```

The other square root is minus this one:

```
In[17]:= Mod[-%, 98573007539]
```

```
Out[17]= 234989854
```

Example 17

Let $n = 34222273 = 9803 \cdot 3491$. Find all four solutions of $x^2 \equiv 19101358 \pmod{34222273}$.

SOLUTION

First, find a square root mod each of the two prime factors, both of which are congruent to 3 (mod 4):

```
In[18]:= PowerMod[19101358, (9803 + 1)/4,  
9803]
```

```
Out[18]= 3998
```

```
In[19]:= PowerMod[19101358, (3491 + 1)/4,  
3491]
```

```
Out[19]= 1318
```

Therefore, the square roots are congruent to $\pm 3998 \pmod{9803}$ and are congruent to $\pm 1318 \pmod{3491}$. There are four ways to combine these using the Chinese remainder theorem:

```
In[20]:= ChineseRemainder[ {3998, 1318},  
{9803, 3491}]
```

```
Out[20]= 43210
```

```
In[21]:= ChineseRemainder[ {-3998, 1318},  
{9803, 3491}]
```

```
Out[21]= 8397173
```

```
In[22]:= ChineseRemainder[ {3998, -1318},  
{9803, 3491}]
```

```
Out[22]= 25825100
```

```
In[23]:= ChineseRemainder[ {-3998, -1318},  
{9803, 3491}]
```

```
Out[23]= 34179063
```

These are the four desired square roots.

A.5 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod{2}.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is {1, 0, 1, 0, 0} and the initial values are given by the vector {0, 1, 0, 0, 0}.

Type

```
In[1]:= lfsr[{1, 0, 1, 0, 0}, {0, 1, 0, 0, 0}, 50]
```

```
Out[1]= {0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1,
0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0,
0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1}
```

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recurrence that generates this sequence.

SOLUTION

First, we find the length of the recurrence. The command *lfsrlength[v, n]* calculates the determinants mod 2 of the

first n matrices that appear in the procedure in Section 5.2:

In[2]:=

```
lfsrlength[{1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1,  
1, 1, 0, 1, 0, 1, 0, 1}, 10]  
{1, 1}  
{2, 1}  
{3, 0}  
{4, 1}  
{5, 0}  
{6, 1}  
{7, 0}  
{8, 0}  
{9, 0}  
{10, 0}
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

```
In[3]:= lfsrsolve[{1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1}, 6]
```

Out[3]= {1, 0, 1, 1, 1, 0}

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod{2}.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning of the LFSR output:

```
In[4]:= Mod[{1, 1, 1, 1, 1, 1, 0, 0, 0, 0,  
0, 0, 1, 1, 1, 0, 0} + {0, 1, 1, 0, 1, 0,  
1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1}, 2]
```

```
Out[4]= {1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,  
1, 0, 1, 1, 0, 1}
```

This is the beginning of the LFSR output. Now let's find the length of the recurrence:

```
In[5]:= lfsrlength[% , 8]
```

```
{1, 1}  
{2, 0}  
{3, 1}  
{4, 0}  
{5, 1}  
{6, 0}  
{7, 0}  
{8, 0}
```

We guess the length is 5. To find the coefficients of the recurrence:

```
In[6]:= lfsrsolve[%%, 5]
```

```
Out[6]= {1, 1, 0, 0, 1}
```

Now we can generate the full output of the LFSR using the coefficients we just found plus the first five terms of the LFSR output:

```
In[7]:= lfsr[{1, 1, 0, 0, 1}, {1, 0, 0, 1,  
0}, 40]
```

```
Out[7]={1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,  
0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0,  
0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0}
```

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

```
In[8]:= Mod[% + {0, 1, 1, 0, 1, 0, 1, 0, 1,  
0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1,  
0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,  
1, 1, 0}, 2]
```

```
Out[8]={1, 1, 1, 1, 1, 1, 0, 0, 0, 0,  
0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0,  
0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,  
0}
```

This is the plaintext.

A.6 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}.$$

Decrypt it.

SOLUTION

A matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is entered as $\{\{a, b\}, \{c, d\}\}$. Type

$M.N$ to multiply matrices M and N . Type $v.M$ to multiply a vector v on the right by a matrix M .

First, we need to invert the matrix mod 26:

```
In[1]:= Inverse[\{\{ 1,2,3\},\{ 4,5,6\},  
\{7,8,10\}\}]
```

```
Out[1]= \{\{-\frac{2}{3}, -\frac{4}{3}, 1\}, \{\frac{2}{3}, \frac{11}{3}, -2\}, \{1,  
-2, 1\}\}
```

Since we are working mod 26, we can't stop with numbers like $2/3$. We need to get rid of the denominators and reduce mod 26. To do so, we multiply by 3 to extract the numerators of the fractions, then

multiply by the inverse of 3 mod 26 to put the “denominators” back in (see [Section 3.3](#)):

```
In[2]:= %*3
```

```
Out[2]= {{-2, -4, 3}, {-2, 11, -6}, {3, -6, 3}}
```

```
In[3]:= Mod[PowerMod[3, -1, 26]*%, 26]
```

```
Out[3]= {{8, 16, 1}, {8, 21, 24}, {1, 24, 1}}
```

This is the inverse of the matrix mod 26. We can check this as follows:

```
In[4]:= Mod[%.{ {1, 2, 3}, {4, 5, 6}, {7, 8, 10}}, 26]
```

```
Out[4]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

```
In[5]:= Mod[{22, 09, 00}.%, 26]
```

```
Out[5]= {14, 21, 4}
```

```
In[6]:= Mod[{12, 03, 01}.%%, 26]
```

```
Out[6]= {17, 19, 7}
```

```
In[7]:= Mod[{10, 03, 04}.%%% , 26]
```

```
Out[7]= {4, 7, 8}
```

```
In[8]:= Mod[{08, 01, 17}.%%%%, 26]
```

```
Out[8]= {11, 11, 23}
```

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. This can be changed back to letters:

```
In[9]:= num2txt0[14210417190704070811123]
```

```
Out[9]= overthehillx
```

Note that the final x was appended to the plaintext in order to complete a block of three letters.

A.7 Examples for Chapter 9

Example 22

Suppose you need to find a large random prime of 50 digits. Here is one way. The function *NextPrime*[*x*] finds the next prime greater than *x*. The function *Random*[*Integer*,{*a,b*}] gives a random integer between *a* and *b*. Combining these, we can find a prime:

```
In[1]:= NextPrime[Random[Integer, {10^49, 10^50 }]]
```

```
Out[1]=  
730505700316671091752153033404883134567089  
13284291
```

If we repeat this procedure, we should get another prime:

```
In[2]:= NextPrime[Random[Integer, {10^49, 10^50 }]]
```

```
Out[2]=  
974764076949313032557243260405861441453410  
54568331
```

Example 23

Suppose you want to change the text *hellohowareyou* to numbers:

```
In[3]:= txt2num1["hellohowareyou"]
```

```
Out[3]= 805121215081523011805251521
```

Note that we are now using $a = 1, b = 2, \dots, z = 26$, since otherwise a 's at the beginnings of messages would disappear. (A more efficient procedure would be to work in base 27, so the numerical form of the message would be

$$8 + 5 \cdot 27 + 12 \cdot 27^2 + \dots + 21 \cdot 27^{13} = 87495221502384554951$$

. Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

```
In[4]:= num2txt1[805121215081523011805251521]
```

```
Out[4]= hellohowareyou
```

Example 24

Encrypt the message *hi* using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the message to numbers:

```
In[5]:= txt2num1["hi"]
```

```
Out[5]= 809
```

Now, raise it to the e th power mod n :

```
In[6]:= PowerMod[%, 17, 823091]
```

```
Out[6]= 596912
```

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d . To do this, we need to find $\phi(823091)$. One way is as follows:

```
In[7]:= EulerPhi[823091]
```

```
Out[7]= 821184
```

Another way is to factor n as $p \cdot q$ and then compute $(p - 1)(q - 1)$:

```
In[8]:= FactorInteger[823091]
```

```
Out[8]= { {659, 1}, {1249, 1} }
```

```
In[9]:= 658*1248
```

```
Out[9]= 821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute the following (note that we are finding the inverse of e mod $\phi(n)$, not mod n):

```
In[10]:= PowerMod[17, -1, 821184]
```

```
Out[10]= 48305
```

Therefore, $d = 48305$. To decrypt, raise the ciphertext to the d th power mod n :

```
In[11]:= PowerMod[596912, 48305, 823091]
```

```
Out[11]= 809
```

Finally, change back to letters:

```
In[12]:= num2txt1[809]
```

```
Out[12]= hi
```

Example 26

Encrypt *hellohowareyou* using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the plaintext to numbers:

```
In[13]:= txt2num1["hellohowareyou"]
```

```
Out[13]= 805121215081523011805251521
```

Suppose we simply raised this to the e th power mod n
:

```
In[14]:= PowerMod[%, 17, 823091]
```

```
Out[14]= 447613
```

If we decrypt (we know d from Example 25), we obtain

```
In[15]:= PowerMod[%, 48305, 823091]
```

```
Out[15]= 628883
```

This is not the original plaintext. The reason is that the plaintext is larger than n , so we have obtained the plaintext mod n :

```
In[16]:= Mod[805121215081523011805251521,  
823091]
```

```
Out[16]= 628883
```

We need to break the plaintext into blocks, each less than n . In our case, we use three letters at a time:

80512 121508 152301 180525 1521

```
In[17]:= PowerMod[80512, 17, 823091]
```

```
Out[17]= 757396
```

```
In[18]:= PowerMod[121508, 17, 823091]
```

```
Out[18]= 164513
```

```
In[19]:= PowerMod[152301, 17, 823091]
```

```
Out[19]= 121217
```

```
In[20]:= PowerMod[180525, 17, 823091]
```

```
Out[20]= 594220
```

```
In[21]:= PowerMod[1521, 17, 823091]
```

```
Out[21]= 442163
```

The ciphertext is therefore

757396164513121217594220442163. Note that there is no reason to change this back to letters. In fact, it doesn't correspond to any text with letters.

Decrypt each block individually:

```
In[22]:= PowerMod[757396, 48305, 823091]
```

```
Out[22]= 80512
```

```
In[23]:= PowerMod[164513, 48305, 823091]
```

```
Out[23]= 121508
```

Etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in [Section 9.5](#). These are stored under the names *rsan*, *rsae*, *rsap*, *rsaq*:

```
In[24]:= rsan
```

Out[24]=

```
1143816257578888676692357799761466120102182967212  
42362562561842935  
7069352457338978305971235639587050589890751475992  
90026879543541
```

In[25]:= rsae

Out[25]= 9007

Example 27

Encrypt each of the messages *b*, *ba*, *bar*, *bard* using *rsan* and *rsaе*.

In[26]:= PowerMod[num1["b"], rsaе, rsan]

Out[26]=

```
7094675846761266859837016499155078618287633106068  
52354105647041144  
8678226171649720012215533234846201405328798758089  
9263765142534
```

In[27]:= PowerMod[txt2num1["ba"], rsaе,
rsan]

Out[27]=

```
3504513060897510032501170944987195427378820475394  
85930603136976982  
2762175980602796227053803156556477335203367178226  
1305796158951
```

In[28]:= PowerMod[txt2num1["bar"], rsaе,
rsan]

Out[28]=

```
4481451286385510107600453085949210934242953160660  
74090703605434080  
0084364598688040595310281831282258636258029878444  
1151922606424
```

In[29]:= PowerMod[txt2num1["bard"], rsae,
rsan]

Out[29]=

```
2423807778511166642320286251209031739348521295905  
62707831349916142  
5605432329717980492895807344575266302644987398687  
7989329909498
```

Observe that the ciphertexts are all the same length.
There seems to be no easy way to determine the length of
the corresponding plaintext.

Example 28

Using the factorization $rsan = rsap \cdot rsaq$, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see [Section 9.5](#)).

SOLUTION

First we find the decryption exponent:

```
In[30]:=rsad=PowerMod[rsaе,-1,(rsap-1)*  
(rsaq-1)];
```

Note that we use the final semicolon to avoid printing out the value. If you want to see the value of $rsad$, see [Section 9.5](#), or don't use the semicolon. To decrypt the ciphertext, which is stored as $rsaci$, and change to letters:

```
In[31]:=num2txt1[PowerMod[rsaci, rsad,  
rsan]]
```

```
Out[31]= the magic words are squeamish  
ossifrage
```

Example 29

Encrypt the message *rsaencryptsmessageswell* using *rsan* and *rsae*.

```
In[32]:=  
PowerMod[txt2num1["rsaencryptsmessageswell  
"], rsae, rsan]
```

```
Out[32]=
```

```
9463942034900225931630582353924949641464096993400  
17097214043524182  
7195065425436558490601396632881775353928311265319  
7553130781884
```

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption exponent *rsad*. Therefore, we compute

```
In[33]:= PowerMod[%, rsad, rsan]
```

```
Out[33]=  
181901051403182516201913051919010705192305  
1212
```

```
In[34]:= num2txt1[%]
```

```
Out[34]= rsaencryptsmessageswell
```

Suppose we lose the final 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4):

```
In[35]:= PowerMod[(%% - 4)/10, rsad, rsan]
```

```
Out[35]=
```

```
4795299917319598866490235262952548640911363389437  
56298468549079705  
8841230037348796965779425411715895692126791262846  
1494475682806
```

If we try to change this to letters, we get a long error message. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that

$n = 11313771275590312567$ is the product of two primes and that $\phi(n) = 11313771187608744400$.

Factor n .

SOLUTION

We know (see [Section 9.1](#)) that p and q are the roots of $X^2 - (n - \phi(n) + 1)X + n$. Therefore, we compute

```
In[36]:= Roots[X^2 - (11313771275590312567  
- 11313771187608744400 + 1)*X +  
11313771275590312567 == 0, X]
```

```
Out[36]:= X == 128781017 || X == 87852787151
```

Therefore, $n = 128781017 \cdot 87852787151$. We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know $rsae$ and $rsad$. Use these to factor $rsan$.

SOLUTION

We use the $a^r \equiv 1$ factorization method from [Section 9.4](#). First write $rsae \cdot rsad - 1 = 2^s m$ with m odd. One way to do this is first to compute $rsae \cdot rsad - 1$, then keep dividing by 2 until we get an odd number:

```
In[37]:= rsae*rsad - 1
```

```
Out[37]=
```

```
9610344196177822661569190233595838341098541290518  
78330250644604041  
1559855750873526591561748985573429951315946804310  
86921245830097664
```

```
In[38]:= %/2
```

```
Out[38]=
```

```
4805172098088911330784595116797919170549270645259  
39165125322302020  
5779927875436763295780874492786714975657973402155  
43460622915048832
```

```
In[39]:= %/2
```

```
Out[39]=
```

```
2402586049044455665392297558398959585274635322629
69582562661151010
2889963937718381647890437246393357487828986701077
71730311457524416
```

We continue this way for six more steps until we get

Out[45]=

```
3754040701631961977175464934998374351991617691608
89972754158048453
5765568652684971324828808197489621074732791720433
933286116523819
```

This number is m . Now choose a random integer a . Hoping to be lucky, we choose 13. As in the $a^r \equiv 1$ factorization method, we compute

In[46]:= PowerMod[13, %, rsan]

Out[46]=

```
2757436850700653059224349486884716119842309570730
78056905698396470
3018310983986237080052933809298479549019264358796
0859870551239
```

Since this is not $\pm 1 \pmod{rsan}$, we successively square it until we get ± 1 :

In[47]:= PowerMod[%, 2, rsan]

Out[47]=

```
4831896032192851558013847641872303455410409906994
08462254947027766
5499641258295563603526615610868643119429857407585
4037512277292
```

```
In[48]:= PowerMod[% , 2 , rsan]
```

```
Out[48]=
```

```
7817281415487735657914192805875400002194878705648  
38209179306251152  
1518183974205601327552191348756094473207351648772  
2273875579363
```

```
In[49]:= PowerMod[% , 2 , rsan]
```

```
Out[49]=
```

```
4283619120250872874219929904058290020297622291601  
77671675518702165  
0944451823946218637947056944205510139299229308225  
9601738228702
```

```
In[50]:= PowerMod[% , 2 , rsan]
```

```
Out[50]= 1
```

Since the last number before the 1 was not $\pm 1 \pmod{rsan}$, we have an example of $x \not\equiv \pm 1 \pmod{rsan}$ with $x^2 \equiv 1$. Therefore, $\gcd(x - 1, rsan)$ is a nontrivial factor of $rsan$:

```
In[51]:= GCD[%% - 1 , rsan]
```

```
Out[51]=
```

```
3276913299326670954996198819083446141317764296799  
2942539798288533
```

This is $rsaq$. The other factor is obtained by computing $rsan/rsaq$:

```
In[52]:= rsan/%
```

Out[52]=

```
3490529510847650949147849619903898133417764638493  
387843990820577
```

This is *rsap*.

Example 33

Suppose you know that

$$150883475569451^2 \equiv 16887570532858^2 \pmod{205611444308117}.$$

Factor 205611444308117.

SOLUTION

We use the Basic Principle of Section 9.4.

```
In[53]:= GCD[150883475569451 -  
16887570532858, 205611444308117]
```

Out[53]= 23495881

This gives one factor. The other is

```
In[54]:= 205611444308117/%
```

Out[54]= 8750957

We can check that these factors are actually primes, so we can't factor any further:

```
In[55]:= PrimeQ[%]
```

Out[55]= True

```
In[56]:= PrimeQ[%]
```

```
Out[56]= True
```

Example 34

Factor

$$n = 376875575426394855599989992897873239$$

by the $p - 1$ method.

SOLUTION

Let's choose our bound as $B = 100$, and let's take $a = 2$, so we compute $2^{100!} \pmod{n}$:

```
In[57]:=
```

```
PowerMod[2, Factorial[100], 3768755754263948  
55599989992897873239]
```

```
Out[57]=
```

```
369676678301956331939422106251199512
```

Then we compute the gcd of $2^{100!} - 1$ and n :

```
In[58]:= GCD[% - 1,
```

```
376875575426394855599989992897873239]
```

```
Out[58]= 430553161739796481
```

This is a factor p . The other factor q is

```
In[59]:=
```

```
376875575426394855599989992897873239/%
```

```
Out[59]= 875328783798732119
```

Let's see why this worked. The factorizations of $p - 1$ and $q - 1$ are

```
In[60]:= FactorInteger[430553161739796481 -  
1]
```

```
Out[60]= {{2, 18}, {3, 7}, {5, 1}, {7, 4}, {11, 3}, {47, 1}}
```

```
In[61]:= FactorInteger[875328783798732119 - 1]
```

```
Out[61]= {{2, 1}, {61, 1}, {20357, 1}, {39301, 1}, {8967967, 1}}
```

We see that $100!$ is a multiple of $p - 1$, so

$2^{100!} \equiv 1 \pmod{p}$. However, $100!$ is not a multiple of $q - 1$, so it is likely that $2^{100!} \not\equiv 1 \pmod{q}$.

Therefore, both $2^{100!} - 1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p .

A.8 Examples for Chapter 10

Example 35

Let's solve the discrete log problem $2^x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take $N = 12$ since $N^2 > p - 1 = 130$ and we form two lists. The first is $2^j \pmod{131}$ for $0 \leq j \leq 11$:

```
In[1]:=Do[Print[n, " ", PowerMod[2, n, 131]], {n, 0, 11}]
```

```
Out[1]= 0 1
```

```
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 125
9 119
10 107
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \leq j \leq 11$:

```
In[2]:=Do[Print[n, " ", Mod[71*PowerMod[2, -12*n, 131], 131]], {n, 0, 11}]
```

```
Out[2]= > 0 71
```

```
1 17
2 124
3 26
4 128
```

5	86
6	111
7	93
8	85
9	96
10	130
11	116

The number 128 is on both lists, so we see that
 $2^7 \equiv 71 \cdot 2^{-12*4} \pmod{131}$. Therefore,

$$71 \equiv 2^{7+4*12} \equiv 2^{55} \pmod{131}.$$

A.9 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod_{i=1}^{22} (1 - i/365)$ (note that the product stops at $i = 22$, not $i = 23$). Subtracting from 1 gives the probability that at least two have the same birthday:

```
In[1]:= 1 - Product[1. - i/365, {i, 22}]
```

```
Out[1]= 0.507297
```

Note that we used 1. in the product instead of 1 without the decimal point. If we had omitted the decimal point, the product would have been evaluated as a rational number (try it, you'll see).

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
In[2]:= 1 - Product[1. - i/10^7, {i, 9999}]
```

```
Out[2]= 0.99327
```

Note that the number of phones is about three times the square root of the number of possibilities. This means that we expect the probability to be high, which it is.

From [Section 12.1](#), we have the estimate that if there are around $\sqrt{2(\ln 2)10^7} \approx 3723$ phones, there should be a 50% chance of a match. Let's see how accurate this is:

```
In[3]:= 1 - Product[1. - i/10^7, i, 3722]
```

```
Out[3]= 0.499895
```

A.10 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme.
Everything is mod the prime $p = 987541$. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

One way: First, find the Lagrange interpolating polynomial through the five points:

```
In[1]:= InterpolatingPolynomial[ { {9853,  
853 }, {4421, 4387 }, {6543, 1234 },  
{93293, 78428 }, {12398, 7563 } }, x]
```

```
Out[1]=  
853 + (- $\frac{1767}{2716}$  + (+ $\frac{2406987}{9538347560}$  + (- $\frac{8464915920541}{3130587195363428640000}$ )  
-  $\frac{49590037201346405337547(-93293+x)}{133788641510994876594882226797600000}$ )(-6543+x))(-4421+x))  
(-9853+x)
```

Now evaluate at $x = 0$ to find the constant term (use
 $/.\ .x -> 0$ to evaluate at $x = 0$):

```
In[2]:= %/. x -> 0
```

```
Out[2]=  

$$\frac{204484326154044983230114592433944282591}{22298106918499146099147037799600000}$$

```

We need to change this to an integer mod 987541, so we find the multiplicative inverse of the denominator:

```
In[3]:= PowerMod[Denominator[%], -1, 987541]
```

```
Out[3]= 509495
```

Now, multiply times the numerator to get the desired integer:

```
In[4]:= Mod[Numerator[%]*%, 987541]
```

```
Out[4]= 678987
```

Therefore, 678987 is the secret.

A.11 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from [Section 18.2](#). There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 24691313099. You are supposed to guess which one is the ace. To start, pick a random exponent. We use the semicolon after *khide* so that we cannot cheat and see what value of *k* is being used.

```
In[1]:= k = khide;
```

Now, shuffle the disguised cards (their numbers are raised to the *k*th power mod *p* and then randomly permuted):

```
In[2]:= shuffle
```

```
Out[2]= {14001090567, 16098641856,  
23340023892, 20919427041, 7768690848}
```

These are the five cards (yours will differ from these because the *k* and the random shuffle will be different). None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

```
In[3]:= reveal[%]
```

```
Out[3]= {ten, ace, queen, jack, king}
```

Let's play again:

```
In[4]:= k = khide;
```

```
In[5]:= shuffle
```

```
Out[5]= {13015921305, 14788966861,  
23855418969, 22566749952, 8361552666}
```

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

```
In[6]:= reveal[%]
```

```
Out[6]= {ten, queen, ace, king, jack}
```

Perhaps you need some help. Let's play one more time:

```
In[7]:= k = khide;
```

```
In[8]:= shuffle
```

```
Out[8]= {13471751030, 20108480083,  
8636729758, 14735216549, 11884022059}
```

We now ask for advice:

```
In[9]:= advise[%]
```

```
Out[9]= 3
```

We are advised that the third card is the ace. Let's see (note that %% is used to refer to the next to last output):

```
In[10]:= reveal[%]
```

```
Out[10]= {jack, ten, ace, queen, king}
```

How does this work? Read the part on “How to Cheat” in [Section 18.2](#). Note that if we raise the numbers for the

cards to the $(p - 1)/2$ power mod p , we get

```
In[11]:= PowerMod[{200514, 10010311,  
1721050514, 11091407, 10305}, (24691313099  
- 1)/ 2, 24691313099]
```

```
Out[11]= {1, 1, 1, 1, 24691313098}
```

Therefore, only the ace is a quadratic nonresidue mod p .

A.12 Examples for Chapter 21

Example 40

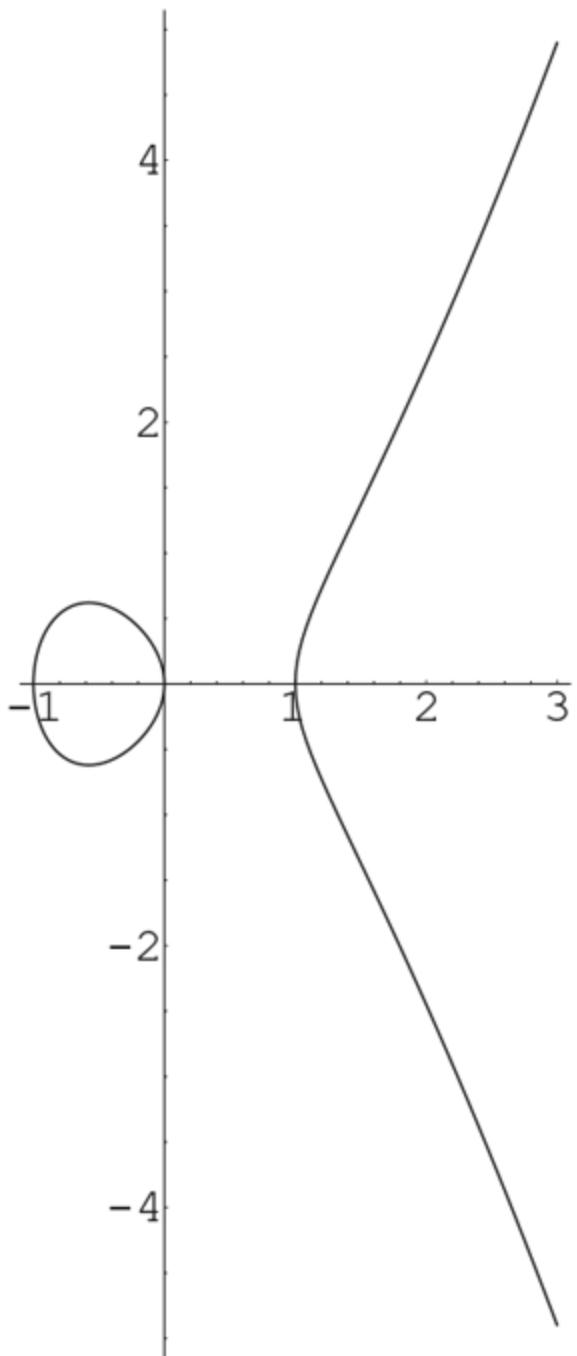
All of the elliptic curves we work with in this chapter are elliptic curves mod n . However, it is helpful to use the graphs of elliptic curves with real numbers in order to visualize what is happening with the addition law, for example, even though such pictures do not exist mod n .

Therefore, let's graph the elliptic curve

$y^2 = x(x - 1)(x + 1)$. We'll specify that $-1 \leq x \leq 3$ and $-y \leq y \leq 5$:

```
In[1]:= ContourPlot[y^2 == x*(x - 1)*(x + 1), {x, -1, 3}, {y, -5, 5}]
```

Graphics



[Full Alternative Text](#)

Example 41

Add the points $(1, 3)$ and $(3, 5)$ on the elliptic curve
 $y^2 \equiv x^3 + 24x + 13 \pmod{29}$.

```
In[2]:= addell[ {1, 3}, {3, 5}, 24, 13,  
29]
```

```
Out[2]= {26, 1}
```

You can check that the point $(26, 1)$ is on the curve:
 $26^3 + 24 \cdot 26 + 13 \equiv 1^2 \pmod{29}$.

Example 42

Add $(1, 3)$ to the point at infinity on the curve of the previous example.

```
In[3]:= addell[ {1, 3}, {"infinity",  
"infinity"}, 24, 13, 29]
```

```
Out[3]= {1, 3}
```

As expected, adding the point at infinity to a point P returns the point P .

Example 43

Let $P = (1, 3)$ be a point on the elliptic curve
 $y^2 \equiv x^3 + 24x + 13 \pmod{29}$. Find $7P$.

```
In[4]:= multell[ {1, 3}, 7, 24, 13, 29]
```

```
Out[4]= {15, 6}
```

Example 44

Find $k(1, 3)$ for $k = 1, 2, 3, \dots, 40$ on the curve of the previous example.

```
In[5]:= multsell[ {1, 3}, 40, 24, 13, 29]
```

```

Out[5]= {1,{1,3},2,{11,10},3,{23,28},4,
{0,10},5,{19,7},6,{18,19},7, {15,6},8,
{20,24},9,{4,12},10,{4,17},11,{20,5},12,
{15,23},13,{18,10}, 14,{19,22},15,{0,19},
16,{23,1},17,{11,19},18,{1,26},19,
{infinity,infinity},20,{1,3},21,{11,10},
22,{23,28},23,{0,10}, 24,{19,7}, 25,
{18,19},26,{15,6},27,{20,24},28,{4,12},29,
{4,17}, 30,{20,5},31,{15,23},32,
{18,10},33,{19,22}, 34,{0,19},35,
{23,1},36, {11,19},37,{1,26}, 38,
{infinity,infinity},39,{1,3},40,{11,10}}

```

Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both factors or we could be at infinity mod one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute $12P$, where $P = (1, 3)$ is on the elliptic curve $y^2 \equiv x^3 - 5x + 13 \pmod{209}$:

```
In[6]:= multell[ {1, 3 }, 12, -5, 13, 11*19]
```

```
Out[6]= {factor=, 19 }
```

Now let's compute the successive multiples to see what happened along the way:

```
In[7]:= multsell[ {1, 3 }, 12, -5, 13,
11*19]
```

```
Out[7]= 1, {{1,3}, 2, {91,27}, 3, {118,133}, 4,  
{148,182}, 5, {20,35}, 6, {factor=, 19}}
```

When we computed $6P$, we ended up at infinity mod 19.
Let's see what is happening mod the two prime factors of
209, namely 19 and 11:

```
In[8]:= multsell[{1,3}, 12, -5, 13, 19]
```

```
Out[8]= 1, {{1,3}, 2, {15,8}, 3, {4,0}, 4,  
{15,11}, 5, {1,16}, 6, {infinity,infinity},  
7, {1,3}, 8, {15,8}, 9, {4,0}, 10, {15,11}, 11,  
{1,16}, 12, {infinity,infinity}}
```

```
In[9]:= multsell[ {1, 3 }, 20, -5, 13, 11]
```

```
Out[9]= 1, {{1,3}, 2, {3,5}, 3, {8,1}, 4, {5,6}, 5,  
{9,2}, 6, {6,10}, 7, {2,0}, 8, {6,1}, 9,  
{9,9}, 10, {5,5}, 11, {8,10}, 12, {3,6}, 13,  
{1,8}, 14, {infinity,infinity}, 15, {1,3},  
16, {3,5}, 17, {8,1}, 18, {5,6}, 19, {9,2}, 20,  
{6,10}}
```

After six steps, we were at infinity mod 19, but it takes 14
steps to reach infinity mod 11. To find $6P$, we needed to
invert a number that was 0 mod 19 and nonzero mod 11.
This couldn't be done, but it yielded the factor 19. This is
the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and
a point on each curve. For example, let's take

$P = (2, 4)$ and the elliptic curve

$$y^2 \equiv x^3 - 10x + b \pmod{193279}.$$

For P to lie on the curve, we take $b = 28$. We'll also take

$$\begin{aligned} y^2 &\equiv x^3 + 11x - 11, & P &= (1, 1) \\ y^2 &\equiv x^3 + 17x - 14, & P &= (1, 2). \end{aligned}$$

Now we compute multiples of the point P . We do the analog of the $p - 1$ method, so we choose a bound B , say $B = 12$, and compute $B!P$.

```
In[10]:= multell[{2,4}, Factorial[12], -10,
28, 193279]
```

```
Out[10]= {factor=, 347}
```

```
In[11]:= multell[{1,1}, Factorial[12], 11,
-11, 193279]
```

```
Out[11]= {13862, 35249}
```

```
In[12]:= multell[{1, 2}, Factorial[12], 17,
-14, 193279]
```

```
Out[12]= {factor=, 557}
```

Let's analyze in more detail what happened in these examples.

On the first curve, $266P$ ends up at infinity mod 557 and $35P$ is infinity mod 347. Since $266 = 2 \cdot 7 \cdot 19$, it has a prime factor larger than $B = 12$, so $B!P$ is not infinity mod 557. But 35 divides $B!$, so $B!P$ is infinity mod 347.

On the second curve, $356P = \text{infinity mod } 347$ and $561P = \text{infinity mod } 557$. Since $356 = 4 \cdot 89$ and $561 = 3 \cdot 11 \cdot 17$, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have $331P = \text{infinity mod } 347$ and $272P = \text{infinity mod } 557$. Since 331 is prime and $272 = 16 \cdot 17$, we don't expect to find the

factorization with this curve. However, by chance, an intermediate step in the calculation of $B!P$ yielded the factorization. Here's what happened. At one step, the program required adding the points $(184993, 13462)$ and $(20678, 150484)$. These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line through these two points is defined mod 347 but is 0/0 mod 557. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from [Section 21.5](#). For more details, see the text. The elliptic curve is $y^2 \equiv x^3 + 3x + 45 \pmod{8831}$ and the point is $G = (4, 11)$. Alice's message is the point $P_m = (5, 1743)$.

Bob has chosen his secret random number $a_B = 3$ and has computed $a_B G$:

```
In[13]:= multell[{4, 11}, 3, 3, 45, 8831]
```

```
Out[13]= {413, 1808}
```

Bob publishes this point. Alice chooses the random number $k = 8$ and computes kG and $P_m + k(a_B G)$:

```
In[14]:= multell[{4, 11}, 8, 3, 45, 8831]
```

```
Out[14]= {5415, 6321}
```

```
In[15]:= addell[{5, 1743}, multell[{413, 1808}, 8, 3, 45, 8831], 3, 45, 8831]
```

```
Out[15]= {6626, 3576}
```

Alice sends $(5415, 6321)$ and $(6626, 3576)$ to Bob, who multiplies the first of these points by a_B :

```
In[16]:= multell[{5415, 6321}, 3, 3, 45,  
8831]
```

```
Out[16]= {673, 146}
```

Bob then subtracts the result from the last point Alice sends him. Note that he subtracts by adding the point with the second coordinate negated:

```
In[17]:= addell[{6626, 3576}, {673, -146},  
3, 45, 8831]
```

```
Out[17]= {5, 1743}
```

Bob has therefore received Alice's message.

Example 48

Let's reproduce the numbers in the example of a Diffie-Hellman key exchange from [Section 21.5](#): The elliptic curve is $y^2 \equiv x^3 + x + 7206 \pmod{7211}$ and the point is $G = (3, 5)$. Alice chooses her secret $N_A = 12$ and Bob chooses his secret $N_B = 23$. Alice calculates

```
In[18]:= multell[{3, 5}, 12, 1, 7206, 7211]
```

```
Out[18]= {1794, 6375}
```

She sends $(1794, 6375)$ to Bob. Meanwhile, Bob calculates

```
In[19]:= multell[{3, 5}, 23, 1, 7206, 7211]
```

```
Out[19]= {3861, 1242}
```

and sends $(3861, 1242)$ to Alice. Alice multiplies what she receives by N_A and Bob multiplies what he receives by

N_B :

```
In[20]:= multell[{3861, 1242}, 12, 1, 7206,  
7211]
```

```
Out[20]= {1472, 2098}
```

```
In[21]:= multell[{1794, 6375}, 23, 1, 7206,  
7211]
```

```
Out[21]= {1472, 2098}
```

Therefore, Alice and Bob have produced the same key.

Appendix B Maple® Examples

These computer examples are written in Maple. If you have Maple available, you should try some of them on your computer. If Maple is not available, it is still possible to read the examples. They provide examples for several of the concepts of this book. For information on getting started with Maple, see [Section B.1](#). To download a Maple notebook that contains the necessary commands, go to

bit.ly/2TzKFec

B.1 Getting Started with Maple

1. Download the Maple notebook math.mws that you find using the links starting at bit.ly/2TzKFec
2. Open Maple (on a Linux machine, use the command `xmaple`; on most other systems, click on the Maple icon)), then open math.mws using the menu options under File on the command bar at the top of the Maple window. (Perhaps this is done automatically when you download it; it depends on your computer settings.)
3. With math.mws in the foreground, press the Enter or Return key on your keyboard. This will load the functions and packages needed for the following examples.
4. You are now ready to use Maple. If you want to try something easy, type $1 + 2^3 + 4^5$; and then press the Return/Enter key. The result 1031 should appear (it's $1 + 2 \cdot 3 + 4^5$).
5. Go to the Computer Examples in [Section B.3](#). Try typing in some of the commands there. The outputs should be the same as those in the examples. Press the Return or Enter key to make Maple evaluate an expression.
6. If you make a mistake in typing in a command and get an error message, you can edit the command and hit Return or Enter to try again. You don't need to retype everything.
7. If you are looking for help or a command to do something, try the Help menu on the command bar at the top. If you can guess the name of a function, there is another way. For example, to obtain information on `gcd`, type `?gcd` and Return or Enter.

B.2 Some Commands

The following are some Maple commands that are used in the examples. Some, such as `phi`, are built into Maple. Others, such as `addell`, are in the Maple notebook available at

bit.ly/2TzKFec

If you want to suppress the output, use a colon instead.

The argument of a function is enclosed in round parentheses. Vectors are enclosed in square brackets. Entering `matrix(m,n,[a,b,c,...,z])` gives the $m \times n$ matrix with first row `a,b,...` and last row `...z`. To multiply two matrices A and B , type `evalm(A*B)`.

If you want to refer to the previous output, use `%`. The next-to-last output is `%%`, etc. Note that `%` refers to the most recent output, not to the last displayed line. If you will be referring to an output frequently, it might be better to name it. For example, `g:=phi(12345)` defines `g` to be the value of $\phi(12345)$. Note that when you are assigning a value to a variable in this way, you should use a colon before the equality sign. Leaving out the colon is a common cause of hard-to-find errors.

Exponentiation is written as `a ^ b`. However, we will need to use modular exponentiation with very large exponents. In that case, use `a &^ b mod n`. For modular exponentiation, you might need to use a `\` between `&` and `^`. Use the right arrow to escape from the exponent.

Some of the following commands require certain Maple packages to be loaded via the commands

```
with(numtheory), with(linalg), with(plots),
with(combinat)
```

These are loaded when the math.mws notebook is loaded. However, if you want to use a command such as `nextprime` without loading the notebook, first type `with(numtheory):` to load the package (once for the whole session). Then you can use functions such as `nextprime`, `isprime`, etc. If you type `with(numtheory)` without the colon, you'll get a list of the functions in the package, too.

The following are some of the commands used in the examples. We list them here for easy reference. To see how to use them, look at the examples. We have used `txt` to refer to a string of letters. Such strings should be enclosed in quotes ("string").

`addell([x,y], [u,v], b, c, n)` finds the sum of the points (x, y) and (u, v) on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod{n}$. The integer n should be odd.

`affinecrypt(txt, m, n)` is the affine encryption of `txt` using $mx + n$.

`allshifts(txt)` gives all 26 shifts of `txt`.

`chrem([a,b,...], [m,n,...])` gives a solution to the simultaneous congruences $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$,

`choose(txt, m, n)` lists the characters in `txt` in positions that are congruent to $n \pmod{m}$.

`coinc(txt, n)` is the number of matches between `txt` and `txt` displaced by n .

`corr(v)` is the dot product of the vector v with the 26 shifts of the alphabet frequency vector.

`phi(n)` computes $\phi(n)$ (don't try very large values of n).

`igcdex(m,n,'x','y')` computes the gcd of m and n along with a solution of $mx + ny = \text{gcd}$. To get x and y , type $x;y$ on this or a subsequent command line.

`ifactor(n)` factors n .

`frequency(txt)` lists the number of occurrences of each letter a through z in `txt`.

`gcd(m,n)` is the gcd of m and n .

`inverse(M)` finds the inverse of the matrix M .

`lfsr(c,k,n)` gives the sequence of n bits produced by the recurrence that has coefficients given by the vector c . The initial values of the bits are given by the vector k .

`lfsrlength(v,n)` tests the vector v of bits to see if it is generated by a recurrence of length at most n .

`lfsrsolve(v,n)` computes the coefficients of a recurrence, given a guess n for the length of the recurrence that generates the binary vector v .

`max(v)` is the largest element of the list v .

$a \bmod n$ is the value of $a \pmod n$.

`multell([x,y], m, b, c, n)` computes m times the point (x, y) on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod n$.

`multsell([x,y], m, b, c, n)` lists the first m multiples of the point (x, y) on the elliptic curve $y^2 \equiv x^3 + bx + c \pmod n$.

`nextprime(x)` gives the next prime $> x$.

`num2text(n)` changes a number n to letters. The successive pairs of digits must each be at most 26 space is oo, a is 01, z is 26.

`primroot(p)` finds a primitive root for the prime p .

`shift(txt,n)` shifts `txt` by n .

`text2num(txt)` changes `txt` to numbers, with space=00, a=01, ..., z=25.

`vigenere(txt,v)` gives the Vigenère encryption of `txt` using the vector v as the key.

`vigvec(txt,m,n)` gives the frequencies of the letters a through z in positions congruent to $n \pmod m$.

B.3 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext kddkmu.
Decrypt it by trying all possibilities.

```
> allshifts("kddkmu")
  "kddkmu"
  "leelnv"
  "mffmow"
  "nggnpx"
  "ohhoqy"
  "piiprz"
  "qjjqsa"
  "rkkrtb"
  "sllsuc"
  "tmmtvd"
  "unnuwe"
  "voovxf"
  "wppwyg"
  "xqqxzh"
  "yrryai"
  "zsszbj"
  "attack"
  "buubdl"
  "cvvcem"
  "dwwdfn"
  "exxego"
  "fyffhp"
  "gzzgiq"
  "haahjr"
  "ibbiks"
  "jccjlt"
```

As you can see, **attack** is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message `cleopatra` using the affine function $7x + 8$:

```
> affinecrypt("cleopatra", 7, 8)  
"whkcjilxi"
```

Example 3

The ciphertext `mzdvezc` was encrypted using the affine function $5x + 12$. Decrypt it.

SOLUTION

First, solve $y \equiv 5x + 12 \pmod{26}$ for x to obtain $x \equiv 5^{-1}(y - 12)$. We need to find the inverse of 5 (mod 26):

```
> 5& ^ (-1) mod 26
```

21

(On some computers, the \wedge doesn't work. Instead, type a backslash \ and then \wedge . Use the right arrow key to escape from the exponent before typing mod. For some reason, a space is needed before a parenthesis in an exponent.)

Therefore, $x \equiv 21(y - 12) \equiv 21y - 12 \cdot 21$. To change $-12 \cdot 21$ to standard form:

```
> -12*21 mod 26
```

Therefore, the decryption function is $x \equiv 21y + 8$. To decrypt the message:

```
> affinecrypt("mzdvezc", 21, 8)  
"anthony"
```

In case you were wondering, the plaintext was encrypted as follows:

```
> affinecrypt("anthony", 5, 12)  
"mzdvezc"
```

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in [Section 2.3](#) to decrypt it. For convenience, we've already stored the ciphertext under the name vvhq.

```
> vvhq  
vvhqvvvrhmustgjgthkihtssejchlsfcbgvwcrlyqtfsvgahw  
kcuuwauglqhnsrljs  
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcqyfn  
svwljsniqkgngrybw  
wgoviokhkazkqkxzgyhcecmeiujoqkfwvefqhkijrclrlkbi  
enqfrjljsdhgrhlsfq  
twlauqrhwdmwlgusgikkflryvcwvsgpmlkassjvoqxeggvey  
ggzmljcxjljsvpaiw  
ikvrdrygfrjljslveggveyggeiapuuuisfpbtgnwwmuczrvtwg  
lrwugumnczvile
```

Find the frequencies of the letters in the ciphertext:

```
> frequency(vvhq)
```

```
[ 8, 5, 12, 4, 15, 10, 27, 16, 13, 14, 17, 25,
 7, 7, 5, 9, 14, 17,
 24, 8, 12, 22, 22, 5, 8, 5]
```

Let's compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

```
> coinc(vvhq,1)
               14
> coinc(vvhq,2)
               14
> coinc(vvhq,3)
               16
> coinc(vvhq,4)
               14
> coinc(vvhq,5)
               24
> coinc(vvhq,6)
               12
```

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to $1 \bmod 5$):

```
> choose(vvhq, 5, 1)
"vvutccqgcunjtpjgkuqpknjkygkkgcjfqrkqjrqudukvpk
vggjjivgjggpfncwuce"
> frequency(%)

[0, 0, 7, 1, 1, 2, 9, 0, 1, 8, 8, 0, 0, 3, 0, 4,
 5, 2, 0, 3, 6, 5, 1, 0, 1, 0]
```

To express this as a vector of frequencies:

```
> vigvec(vvhq, 5, 1)
[0., 0., .1044776119, .01492537313, .01492537313,
 .02985074627, .1343283582, 0., .01492537313,
 .1194029851,
 .1194029851, 0., 0., .04477611940, 0.,
 .05970149254,
```

```
.07462686567, .02985074627, 0., .04477611940,  
.08955223881,  
.07462686567, .01492537313, 0., .01492537313, 0.]
```

The dot products of this vector with the displacements of the alphabet frequency vector are computed as follows:

```
> corr(%)

.02501492539, .03910447762, .07132835821,
.03882089552,
.02749253732, .03801492538, .05120895523,
.03014925374,
.03247761194, .04302985074, .03377611940,
.02985074628,
.03426865672, .04456716420, .03555223882,
.04022388058,
.04343283582, .05017910450, .03917910447,
.02958208957,
.03262686569, .03917910448, .03655223881,
.03161194031,
.04883582088, .03494029848
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

```
> max(%)

.07132835821
```

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter *c*. A procedure similar to the one just used (using `vigvec(vvhq, 5, 2), ..., vigvec(vvhq, 5, 5)`) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

```
> vigenere(vvhq, -[2, 14, 3, 4, 18])
```

themethodusedfortheprparationandreadingofcodemes
sagesissimpleinthe
extremeandatthesametimeimpossibleoftranslationunl
essthekeyisknownth
eeasewithwhichthekeymaybechangedisanotherpointinf
avoroftheadoptiono
fthiscodebythosedesiringtotransmitimportantmessag
eswithouttheslight
estdangeroftheirmessagessbeingreadbypoliticalorbus
inessrivalsetc

For the record, the plaintext was originally encrypted by
the command

```
> vigenere(%, [2, 14, 3, 4, 18])  
  
vvhqvvvrhmustgjgthkihtssejchlsfcbgvwcrlyqtfsvgahw  
kcuuhwauglqhnsrljs  
hbltspisprdxljsveeghlqwkasskuwepwqtwwspgoelkcqyfn  
svwljsniqkgnrgybw1  
wgoviokhkazkqkxzgyhcecmieujoqkfwvefqhkijrclrlkbi  
enqfrjljsdhgrhlsfq  
twlauqrhwdmwlgusgikkflryvcwvsgpmlkassjvoqxeggvey  
ggzmljcxjljsvpaivw  
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrvtwg  
lrwugumnczvile
```

B.4 Examples for Chapter 3

Example 5

Find $\gcd(23456, 987654)$.

```
> gcd(23456, 987654)
```

```
2
```

Example 6

Solve $23456x + 987654y = 2$ in integers x, y .

```
igcdex(23456, 987654,'x','y')
```

```
2
```

```
> x;y
```

```
-3158  
75
```

This means that 2 is the gcd and $23456 \cdot (-3158) + 987654 \cdot 75 = 2$. (The command `igcdex` is for *integer gcd extended*. Maple also calculates gcd's for polynomials.) Variable names other than '`x`' and '`y`' can be used if these letters are going to be used elsewhere, for example, in a polynomial. We can also clear the value of `x` as follows:

```
> x:='x'
```

```
x:=x
```

Example 7

Compute $234 \cdot 456 \pmod{789}$.

```
> 234*456 mod 789
```

```
189
```

Example 8

Compute $234567^{876543} \pmod{565656565}$.

```
> 234567&^876543 mod 565656565
```

```
473011223
```

You might need a `\` before the `^`. Use the right arrow to escape from the exponent mode.

Example 9

Find the multiplicative inverse of
 $87878787 \pmod{9191919191}$.

```
> 87878787&^(-1) mod 9191919191
```

```
7079995354
```

You might need a space before the exponent `(-1)`. (The command `1/87878787 mod 9191919191` also works).

Example 10

Solve $7654x \equiv 2389 \pmod{65537}$.

SOLUTION

Here is one way.

```
> solve(7654*x=2389,x) mod 65537  
43626
```

Here is another way.

```
> 2389/7654 mod 65537  
43626
```

The fraction $2389/7654$ will appear as a vertically set fraction $\frac{2389}{7654}$. Use the right arrow key to escape from the fraction mode.

Example 11

Find x with

$$x \equiv 2 \pmod{78}, \quad x \equiv 5 \pmod{97}, \quad x \equiv 1 \pmod{119}.$$

```
> chrem([2, 5, 1],[78, 97, 119])  
647480
```

We can check the answer:

```
> 647480 mod 78; 647480 mod 97; 647480 mod 119
```

2
5
1

Example 12

Factor 123450 into primes.

```
> ifactor(123450)  
(2) (3)(5)2 (823)
```

This means that $123450 = 2^1 3^1 5^2 823^1$.

Example 13

Evaluate $\phi(12345)$.

```
> phi(12345)  
6576
```

Example 14

Find a primitive root for the prime 65537.

```
> primroot(65537)  
3
```

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix

$$\begin{matrix} 13 & 12 & 35 \\ 41 & 53 & 62 \end{matrix} \pmod{999}.$$
$$71 \quad 68 \quad 10$$

SOLUTION

First, invert the matrix without the mod, and then reduce the matrix mod 999:

```
> inverse(matrix(3,3,[13, 12, 35, 41, 53, 62, 71,
68, 10]))
```

$$\begin{matrix} \frac{3686}{34139} & -\frac{2260}{34139} & \frac{1111}{34139} \\ -\frac{3992}{34139} & \frac{2355}{34139} & -\frac{629}{34139} \\ \frac{975}{34139} & \frac{32}{34139} & -\frac{197}{34139} \end{matrix}$$

```
> map(x->x mod 999, %)
```

$$\begin{matrix} 772 & 472 & 965 \\ 641 & 516 & 851 \\ 150 & 133 & 149 \end{matrix}$$

This is the inverse matrix mod 999.

Example 16

Find a square root of 26951623672 mod the prime $p=98573007539$.

SOLUTION

Since $p \equiv 3 \pmod{4}$, we can use the proposition of Section 3.9:

```
> 26951623672&^((98573007539 + 1)/4) mod  
98573007539  
  
98338017685
```

(You need two right arrows to escape from the fraction mode and then the exponent mode.) The other square root is minus the preceding one:

```
> -% mod 98573007539  
  
234989854
```

Example 17

Let $n = 34222273 = 9803 \cdot 3491$. Find all four solutions of $x^2 \equiv 19101358 \pmod{34222273}$.

SOLUTION

First, find a square root mod each of the two prime factors, both of which are congruent to $3 \pmod{4}$:

```
> 19101358&^((9803 + 1)/4) mod 9803  
  
3998  
  
> 19101358&^((3491 + 1)/4) mod 3491  
  
1318
```

Therefore, the square roots are congruent to $\pm 3998 \pmod{9803}$ and are congruent to ± 1318

$(\text{mod } 3491)$. There are four ways to combine these using the Chinese remainder theorem:

```
> chrem([3998, 1318],[9803, 3491])  
43210  
  
> chrem([-3998, 1318],[9803, 3491])  
8397173  
  
> chrem([3998, -1318],[9803, 3491])  
25825100  
  
> chrem([-3998, -1318],[9803, 3491])  
34179063
```

These are the four desired square roots.

B.5 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod{2}.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is [1, 0, 1, 0, 0] and the initial values are given by the vector [0, 1, 0, 0, 0]. Type

```
> lfsr([1, 0, 1, 0, 0], [0, 1, 0, 0, 0], 50)
[0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1,
 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0,
 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1,
 1, 1]
```

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recurrence that generates this sequence.

SOLUTION

First, we need to find the length of the recurrence. The command `lfsrlength(v, n)` calculates the determinants mod 2 of the first n matrices that appear in the procedure in Section 5.2:

```
> lfsrlength([1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1,
1, 1, 0, 1, 0, 1, 0, 1], 10)

[1, 1]
[2, 1]
[3, 0]
[4, 1]
[5, 0]
[6, 1]
[7, 0]
[8, 0]
[9, 0]
[10, 0]
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

```
> lfsrsolve([1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1,
1, 1, 0, 1, 0, 1, 0, 1], 6)

[1, 0, 1, 1, 1, 0]
```

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod{2}.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning of the LFSR output:

```
> [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,  
0, 0]  
+ [0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0,  
0, 1] mod 2  
  
[1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0,  
1, 1, 0, 1]
```

This is the beginning of the LFSR output. Now let's find the length of the recurrence.

```
> lfsrlength(% , 8)  
  
[1, 1]  
[2, 0]  
[3, 1]  
[4, 0]  
[5, 1]  
[6, 0]  
[7, 0]  
[8, 0]
```

We guess the length is 5. To find the coefficients of the recurrence:

```
> lfsrsolve(%%, 5)  
  
[1, 1, 0, 0, 1]
```

Now we can generate the full output of the LFSR using the coefficients we just found plus the first five terms of the LFSR output:

```
> lfsr([1, 1, 0, 0, 1], [1, 0, 0, 1, 0], 40)  
[1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,
```

```
1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,  
1, 0, 0, 1, 0, 1, 1, 0]
```

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

```
> % + [0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0,  
0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,  
0, 1, 0, 0, 0, 1, 0, 1, 1, 0] mod 2  
  
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,  
0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1,  
1, 0, 0, 0, 0, 0, 0]
```

This is the plaintext.

B.6 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}.$$

Decrypt it.

SOLUTION

There are several ways to input a matrix. One way is the following. A 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ can be entered as `matrix(2,2,[a,b,c,d])`. Type `evalm(M*N)` to multiply matrices M and N . Type `evalm(v*M)` to multiply a vector v on the right by a matrix M .

Here is the encryption matrix.

```
> M:=matrix(3,3,[1,2,3,4,5,6,7,8,10])
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix}$$

We need to invert the matrix mod 26:

```
> invM:=map(x->x mod 26, inverse(M))
```

$$\begin{bmatrix} 8 & 16 & 1 \\ 8 & 21 & 24 \\ 1 & 24 & 1 \end{bmatrix}$$

The command `map(x->x mod 26, E)` takes each number in an expression E and reduces it mod 26.

This is the inverse of the matrix mod 26. We can check this as follows:

```
> evalm(M&*invM)
```

$$\begin{bmatrix} 27 & 130 & 52 \\ 78 & 313 & 130 \\ 130 & 520 & 209 \end{bmatrix}$$

```
> map(x->x mod 26, %)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

```
> map(x->x mod 26, evalm([22,09,00]&*invM))
```

```
[14, 21, 4]
```

```
> map(x->x mod 26, evalm([12,03,01]&*invM))
```

```
[17, 19, 7]
```

```
> map(x->x mod 26, evalm([10,03,04]&*invM))
```

```
[4, 7, 8]
```

```
> map(x->x mod 26, evalm([08,01,17]&*invM))
```

```
[11, 11, 23]
```

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. Changing this back to letters, we obtain overthehillx. Note that the final x was appended to the plaintext in order to complete a block of three letters.

B.7 Examples for Chapter 9

Example 22

Suppose you need to find a large random prime of 50 digits. Here is one way. The function `nextprime` finds the next prime greater than x . The function `rand(a..b)()` gives a random integer between a and b . Combining these, we can find a prime:

```
> nextprime(rand(10 ^ 49..10 ^ 50)())  
  
7305057003166710917521530334048831345670891328429  
1
```

If we repeat this procedure, we should get another prime:

```
> nextprime(rand(10 ^ 49..10 ^ 50)())  
  
9747640769493130325572432604058614414534105456833  
1
```

Example 23

Suppose you want to change the text *hellohowareyou* to numbers:

```
> text2num("hellohowareyou")  
  
805121215081523011805251521
```

Note that we are now using $a = 1, b = 2, \dots, z = 26$, since otherwise a 's at the beginnings of messages would disappear. (A more efficient procedure would be to work in base 27, so the numerical form of the message would be

$$8 + 5 \cdot 27 + 12 \cdot 27^2 + \dots + 21 \cdot 27^{13} = 87495221502384554951.$$

Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

```
> num2text(805121215081523011805251521)  
"hellohowareyou"
```

Example 24

Encrypt the message `hi` using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the message to numbers:

```
> text2num("hi")  
809
```

Now, raise it to the e th power mod n :

```
> %&^17 mod 823091  
596912
```

You might need a `\` before the `^`. Use the right arrow to escape from the exponent mode.

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d . To do this, we need to find $\phi(823091)$. One way is

```
> phi(823091)  
821184
```

Another way is to factor n as $p \cdot q$ and then compute $(p - 1)(q - 1)$:

```
> ifactor(823091)  
(659)(1249)  
> 658*1248  
821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute the following (note that we are finding the inverse of e mod $\phi(n)$, not mod n):

```
> 17&^ (-1) mod 821184  
48305
```

Therefore, $d = 48305$. To decrypt, raise the ciphertext to the d th power mod n :

```
> 596912&^48305 mod 823091
```

Finally, change back to letters:

```
> num2text(809)  
"hi"
```

Example 26

Encrypt `hellohowareyou` using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the plaintext to numbers:

```
> text2num("hellohowareyou")  
805121215081523011805251521
```

Suppose we simply raised this to the e th power mod n :

```
> %%^17 mod 823091  
447613
```

If we decrypt (we know d from [Example 25](#)), we obtain

```
> %%^48305 mod 823091  
628883
```

This is not the original plaintext. The reason is that the plaintext is larger than n , so we have obtained the plaintext mod n :

```
> 805121215081523011805251521 mod 823091
```

```
628883
```

We need to break the plaintext into blocks, each less than n . In our case, we use three letters at a time:

```
80512 121508 152301 180525 1521
```

```
> 80512&^17 mod 823091
```

```
757396
```

```
> 121508&^17 mod 823091
```

```
164513
```

```
> 152301&^17 mod 823091
```

```
121217
```

```
> 180525&^17 mod 823091
```

```
594220
```

```
> 1521&^17 mod 823091
```

```
442163
```

The ciphertext is therefore
757396164513121217594220442163. Note that there
is no reason to change this back to letters. In fact, it
doesn't correspond to any text with letters.

Decrypt each block individually:

```
> 757396&^48305 mod 823091
```

```
80512
```

```
> 164513&^48305 mod 823091
```

```
121508
```

etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in [Section 9.5](#). These are stored under the names *rsan*, *rsae*, *rsap*, *rsaq*:

```
> rsan  
114381625757888676692357799761466120102182967212  
42362562561842935  
7069352457338978305971235639587050589890751475992  
90026879543541  
  
> rsae  
9007
```

Example 27

Encrypt each of the messages *b* , *ba* , *bar* , *bard* using *rsan* and *rsae*.

```
> text2num("b")&^rsa mod rsan  
7094675846761266859837016499155078618287633106068  
52354105647041144  
8678226171649720012215533234846201405328798758089  
9263765142534  
  
> text2num("ba")&^rsa mod rsan  
3504513060897510032501170944987195427378820475394  
85930603136976982  
2762175980602796227053803156556477335203367178226  
1305796158951  
  
> text2num("bar")&^rsa mod rsan  
4481451286385510107600453085949210934242953160660  
74090703605434080  
0084364598688040595310281831282258636258029878444  
1151922606424
```

```
> text2num("bard")^rsa mod rsan  
2423807778511166642320286251209031739348521295905  
62707831349916142  
5605432329717980492895807344575266302644987398687  
7989329909498
```

Observe that the ciphertexts are all the same length.
There seems to be no easy way to determine the length of
the corresponding plaintext.

Example 28

Using the factorization $rsap = rsap \cdot rsap$, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see [Section 9.5](#)).

First we find the decryption exponent:

```
> rsad:=rsa^(-1) mod (rsap-1)*(rsaq-1):
```

Note that we use the final colon to avoid printing out the value. If you want to see the value of $rsad$, see [Section 9.5](#), or don't use the colon. To decrypt the ciphertext, which is stored as $rsaci$, and change to letters:

```
> num2text(rsaci^rsad mod rsan)  
"the magic words are squeamish  
ossifrage"
```

Example 29

Encrypt the message `rsa encrypts messages well` using $rsan$ and rsa .

```
> text2num("rsaencryptsmessageswell")&^rsae mod  
rsan  
  
9463942034900225931630582353924949641464096993400  
17097214043524182  
7195065425436558490601396632881775353928311265319  
7553130781884
```

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption exponent *rsad*. Therefore, we compute

```
> %% ^ rsad mod rsan  
  
1819010514031825162019130519190107051923051212  
> num2text(%)  
"rsaencryptsmessageswell"
```

Suppose we lose the final digit 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4):

```
> (%% - 4)/10&^rsad mod rsan  
  
4795299917319598866490235262952548640911363389437  
56298468549079705  
8841230037348796965779425411715895692126791262846  
1494475682806
```

If we try to change this to letters, we do not get anything resembling the message. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that

$n = 11313771275590312567$ is the product of two

primes and that $\phi(n) = 11313771187608744400$.

Factor n .

SOLUTION

We know (see [Section 9.1](#)) that p and q are the roots of $X^2 - (n - \phi(n) + 1)X + n$. Therefore, we compute

```
> solve(x^2 -  
(11313771275590312567 - 11313771187608744400 +  
1)*x +  
11313771275590312567, x)  
  
87852787151, 128781017
```

Therefore, $n = 128781017 \cdot 87852787151$. We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know $rsae$ and $rsad$. Use these to factor $rsan$.

SOLUTION

We use the $a^r \equiv 1$ factorization method from [Section 9.4](#). First write $rsae \cdot rsad - 1 = 2^s m$ with m odd.

One way to do this is first to compute $rsae \cdot rsad - 1$, and then keep dividing by 2 until we get an odd number:

```
> rsae*rsad - 1

9610344196177822661569190233595838341098541290518
78330250644604041
1559855750873526591561748985573429951315946804310
86921245830097664

> %/2

4805172098088911330784595116797919170549270645259
39165125322302020
5779927875436763295780874492786714975657973402155
43460622915048832

> %/2
2402586049044455665392297558398959585274635322629
69582562661151010
2889963937718381647890437246393357487828986701077
71730311457524416
```

We continue this way for six more steps until we get

```
3754040701631961977175464934998374351991617691608
89972754158048453
5765568652684971324828808197489621074732791720433
933286116523819
```

This number is m . Now choose a random integer a . Hoping to be lucky, we choose 13. As in the $a^r \equiv 1$ factorization method, we compute

```
> 13&^% mod rsan

2757436850700653059224349486884716119842309570730
78056905698396470
3018310983986237080052933809298479549019264358796
0859870551239
```

Since this is not $\pm 1 \pmod{rsan}$, we successively square it until we get ± 1 :

```
> %&^2 mod rsan  
  
4831896032192851558013847641872303455410409906994  
08462254947027766  
5499641258295563603526615610868643119429857407585  
4037512277292  
  
> %&^2 mod rsan  
  
7817281415487735657914192805875400002194878705648  
38209179306251152  
1518183974205601327552191348756094473207351648772  
2273875579363  
  
> %&^2 mod rsan  
  
4283619120250872874219929904058290020297622291601  
77671675518702165  
0944451823946218637947056944205510139299229308225  
9601738228702  
  
> %&^2 mod rsan
```

1

Since the last number before the 1 was not $\pm 1 \pmod{rsan}$, we have an example of $x \not\equiv \pm 1 \pmod{rsan}$ with $x^2 \equiv 1$. Therefore, $\gcd(x - 1, rsan)$ is a nontrivial factor of $rsan$:

```
> gcd(%% - 1, rsan)  
  
3276913299326670954996198819083446141317764296799  
2942539798288533
```

This is $rsaq$. The other factor is obtained by computing $rsan/rsaq$:

rsan/%

```
3490529510847650949147849619903898133417764638493  
387843990820577
```

This is *rsap*.

Example 33

Suppose you know that

$$150883475569451^2 \equiv 16887570532858^2 \pmod{205611444308117}.$$

Factor 205611444308117.

SOLUTION

We use the Basic Principle of Section 9.4:

```
> gcd(150883475569451 -  
16887570532858, 205611444308117)
```

```
23495881
```

This gives one factor. The other is

```
> 205611444308117 / %  
8750957
```

We can check that these factors are actually primes, so we can't factor any further:

```
> isprime(%%)  
true  
> isprime(%%)  
true
```

Example 34

Factor

$$n = 376875575426394855599989992897873239$$

by the $p - 1$ method.

SOLUTION

Let's choose our bound as $B = 100$, and let's take $a = 2$, so we compute $2^{100!} \pmod{n}$:

```
> 2&^factorial(100)
mod 376875575426394855599989992897873239
```

```
369676678301956331939422106251199512
```

Then we compute the gcd of $2^{100!} - 1$ and n :

```
> gcd(% - 1,
376875575426394855599989992897873239)
```

```
430553161739796481
```

This is a factor p . The other factor q is

```
> 376875575426394855599989992897873239/%
```

```
875328783798732119
```

Let's see why this worked. The factorizations of $p - 1$ and $q - 1$ are

```
> ifactor(430553161739796481 - 1)
(2)^18(3)^7(5)(7)^4(11)^3(47)

> ifactor(875328783798732119 - 1)
```

(2)(61)(8967967)(20357)(39301)

We see that $100!$ is a multiple of $p - 1$, so $2^{100!} \equiv 1 \pmod{p}$. However, $100!$ is not a multiple of $q - 1$, so it is likely that $2^{100!} \not\equiv 1 \pmod{q}$. Therefore, both $2^{100!} - 1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p .

B.8 Examples for Chapter 10

Example 35

Let's solve the discrete log problem $2^x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take $N = 12$ since $N^2 > p - 1 = 130$ and we form two lists. The first is $2^j \pmod{131}$ for $0 \leq j \leq 11$:

```
>for j from 0 while j <= 11 do; (j, 2&j mod 131);  
end do;
```

```
0, 1  
1, 2  
2, 4  
3, 8  
4, 16  
5, 32  
6, 64  
7, 128  
8, 125  
9, 119  
10, 107  
11, 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \leq j \leq 11$:

```
> for j from 0 while j <= 11 do; (j, 71*2&\^\\:(  
-12*j) mod 131); end do;
```

```
0, 71  
1, 17  
2, 124  
3, 26  
4, 128  
5, 86  
6, 111  
7, 93
```

8, 85
9, 96
10, 130
11, 116

The number 128 is on both lists, so we see that $2^7 \equiv 71 \cdot 2^{-12*4} \pmod{131}$. Therefore,

$$71 \equiv 2^{7+4*12} \equiv 2^{55} \pmod{131}.$$

B.9 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod_{i=1}^{22} (1 - i/365)$ (note that the product stops at $i = 22$, not $i = 23$). Subtracting from 1 gives the probability that at least two have the same birthday:

```
> 1-mul(1.-i/365, i=1..22)
.5072972344
```

Note that we used 1. in the product instead of 1 without the decimal point. If we had omitted the decimal point, the product would have been evaluated as a rational number (try it, you'll see).

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
> 1-mul(1.-i/10^7, i=1..9999)
.9932699133
```

Note that the number of phones is about three times the square root of the number of possibilities. This means that we expect the probability to be high, which it is.

From [Section 12.1](#), we have the estimate that if there are around $\sqrt{2(\ln 2)10^7} \approx 3723$ phones, there should be a 50% chance of a match. Let's see how accurate this is:

```
> 1-mul(1.-i/10^7, i=1..3722)  
.4998945441
```

B.10 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme.
Everything is mod the prime $p = 987541$. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

One way: First, find the Lagrange interpolating polynomial through the five points:

```
> interp([9853,4421,6543,93293,12398],  
[853,4387,1234,78428,7563],x)
```

$$\begin{aligned}& -\frac{49590037201346405337547}{133788641510994876594882226797600000}x^4 \\& + \frac{353130857169192557779073307}{8919242767399658439658815119840000}x^3 \\& - \frac{8829628978321139771076837361481}{19112663072999268084983175256800000}x^2 \\& + \frac{9749049230474450716950803519811081}{44596213836998292198294075599200000}x \\& + \frac{204484326154044983230114592433944282591}{22298106918499146099147037799600000}\end{aligned}$$

Now evaluate at $x = 0$ to find the constant term:

```
> eval(%,x=0)
```

$$\frac{204484326154044983230114592433944282591}{22298106918499146099147037799600000}$$

We need to change this to an integer mod 987541:

```
> % mod 987541  
678987
```

Therefore, 678987 is the secret.

Here is another way. Set up the matrix equations as in the text and then solve for the coefficients of the polynomial mod 987541:

```
> map(x->x mod 987541, evalm(inverse(matrix(5,5,  
[1,9853,9853^2,9853^3,9853^4,  
1,4421,4421^2,4421^3,4421^4,  
1,6543,6543^2,6543^3, 6543^4,  
1, 93293, 93293^2,93293^3, 93293^4,  
1, 12398, 12398^2,12398^3,12398^4])))  
&*matrix(5,1,[853,4387,1234,78428,7563])))
```

```
678987  
14728  
1651  
574413  
456741
```

The constant term is 678987, which is the secret.

B.11 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from [Section 18.2](#). There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 24691313099. You are supposed to guess which one is the ace.

To start, pick a random exponent. We use the colon after `khide()` so that we cannot cheat and see what value of k is being used.

```
> k := khide():
```

Now, shuffle the disguised cards (their numbers are raised to the k th power mod p and then randomly permuted):

```
> shuffle(k)  
[14001090567, 16098641856, 23340023892,  
20919427041, 7768690848]
```

These are the five cards. None looks like the ace that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

```
> reveal(%)

["ten", "ace", "queen", "jack",
"king"]
```

Let's play again:

```
> k:= khide():

> shuffle(k)

[13015921305, 14788966861, 23855418969,
22566749952, 8361552666]
```

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

```
> reveal(%)

["ten", "queen", "ace", "king",
"jack"]
```

Perhaps you need some help. Let's play one more time:

```
> k:= khide():

> shuffle(k)

[13471751030, 20108480083, 8636729758,
14735216549, 11884022059]
```

We now ask for advice:

```
> advise(%)
```

We are advised that the third card is the ace. Let's see (recall that $\% \%$ is used to refer to the next to last output):

```
> reveal(%%)  
["jack", "ten", "ace", "queen",  
"king"]
```

How does this work? Read the part on “How to Cheat” in [Section 18.2](#). Note that if we raise the numbers for the cards to the $(p - 1)/2$ power mod p , we get

```
> map(x->x&^((24691313099-1)/2) mod 24691313099,  
[200514, 10010311, 1721050514, 11091407, 10305])  
[1, 1, 1, 1, 24691313098]
```

Therefore, only the ace is a quadratic nonresidue mod p .

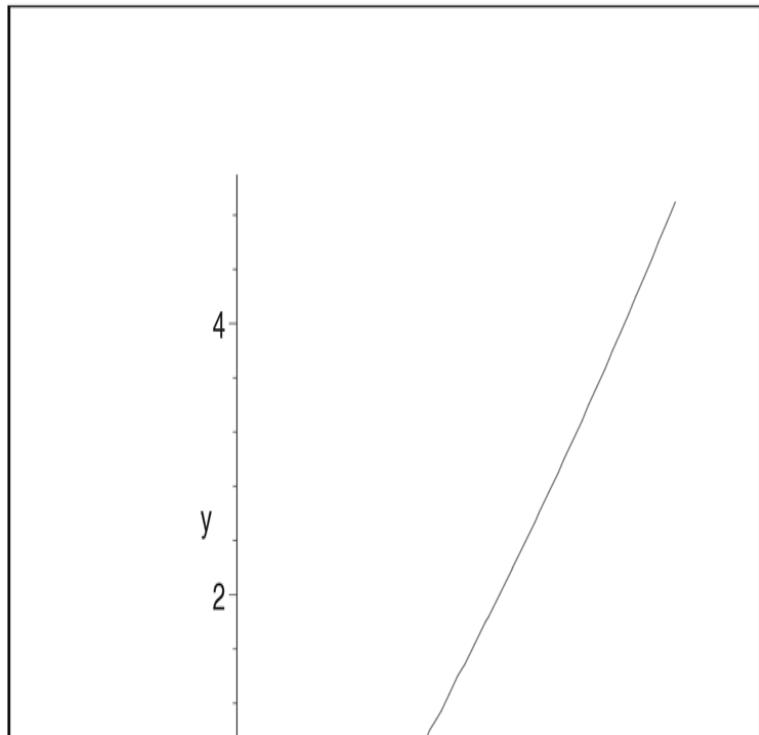
B.12 Examples for Chapter 21

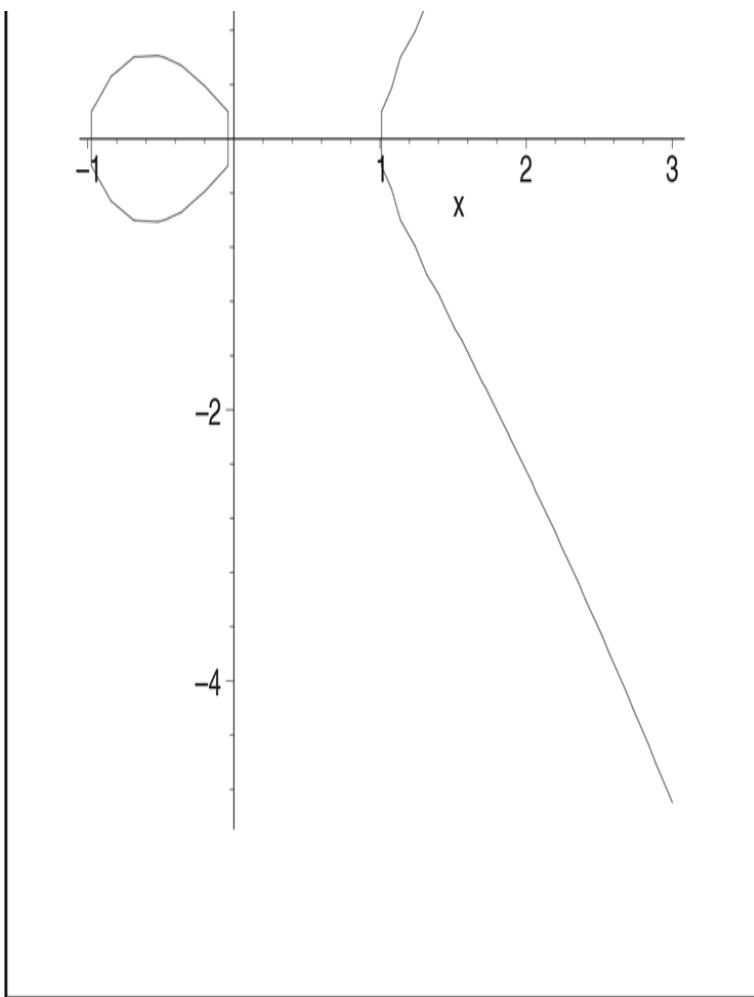
Example 40

All of the elliptic curves we work with in this chapter are elliptic curves mod n . However, it is helpful use the graphs of elliptic curves with real numbers in order to visualize what is happening with the addition law, for example, even though such pictures do not exist mod n .

Let's graph the elliptic curve $y^2 = x(x - 1)(x + 1)$. We'll specify that $-1 \leq x \leq 3$ and $-5 \leq y \leq 5$, and make sure that x and y are cleared of previous values.

```
> x:='x';y:='y';implicitplot(y^2=x*(x-1)*(x+1),  
x=-1..3,y=-5..5)
```





B.12-1 Full Alternative Text

Example 41

Add the points $(1, 3)$ and $(3, 5)$ on the elliptic curve
 $y^2 \equiv x^3 + 24x + 13 \pmod{29}$.

```
> addell([1,3], [3,5], 24, 13, 29)
[26,1]
```

You can check that the point $(26, 1)$ is on the curve:
 $26^3 + 24 \cdot 26 + 13 \equiv 1^2 \pmod{29}$.

Example 42

Add $(1, 3)$ to the point at infinity on the curve of the previous example.

```
> addell([1,3], ["infinity","infinity" ], 24, 13,  
29)
```

```
[1,3]
```

As expected, adding the point at infinity to a point P returns the point P .

Example 43

Let $P = (1, 3)$ be a point on the elliptic curve $y^2 \equiv x^3 + 24x + 13 \pmod{29}$. Find $7P$.

```
> multell([1,3], 7, 24, 13, 29)
```

```
[15,6]
```

Example 44

Find $k(1, 3)$ for $k = 1, 2, 3, \dots, 40$ on the curve of the previous example.

```
> multsell([1,3], 40, 24, 13, 29)  
[[1,[1,3]],[2,[11,10]],[3,[23,28]],[4,[0,10]],[5,  
[19,7]],[6,[18,19]],  
[7,[15,6]],[8,[20,24]],[9,[4,12]],[10,[4,17]],  
[11,[20,5]],  
[12,[15,23]],[13,[18,10]],[14,[19,22]],[15,  
[0,19]],[16,[23,1]],  
[17,[11,19]],[18,[1,26]],[19,  
["infinity","infinity"]],[20,[1,3]],
```

```
[21,[11,10]],[22,[23,28]],[23,[0,10]], [24,
[19,7]],[25,[18,19]],
[26,[15,6]],[27,[20,24]],[28,[4,12]],[29,[4,17]],
[30,[20,5]],
[31,[15,23]],[32,[18,10]],[33,[19,22]],[34,
[0,19]],[35,[23,1]],
[36,[11,19]],[37,[1,26]],[38,
["infinity","infinity"]],[39,[1,3]],
[40,[11,10]]]
```

Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both factors or we could be at infinity mod one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute $12P$, where $P = (1, 3)$ is on the elliptic curve $y^2 \equiv x^3 - 5x + 13 \pmod{209}$:

```
> multell([1,3], 12, -5, 13, 11*19)
["factor=", 19]
```

Now let's compute the successive multiples to see what happened along the way:

```
> multsell([1,3], 12, -5, 13, 11*19)
[[1,[1,3]],[2,[91,27]],[3,[118,133]],[4,
[148,182]],[5,[20,35]],
[6,["factor=",19]]]
```

When we computed $6P$, we ended up at infinity mod 19. Let's see what is happening mod the two prime factors of

209, namely 19 and 11:

```
> multsell([1,3], 12, -5, 13, 19)
[[1,[1,3]],[2,[15,8]],[3,[4,0]],[4,[15,11]],[5,
[1,16]],
[6,["infinity","infinity"]],[7,[1,3]],[8,[15,8]],
[9,[4,0]],
[10,[15,11]],[11,[1,16]],[12,
["infinity","infinity"]]]
```



```
> multsell([1,3], 24, -5, 13, 11)
[[1,[1,3]],[2,[3,5]],[3,[8,1]],[4,[5,6]],[5,
[9,2]],
[6,[6,10]],
[7,[2,0]],[8,[6,1]],[9,[9,9]],[10,[5,5]],[11,
[8,10]],
[12,[3,6]],
[13,[1,8]],[14,['infinity','infinity']],[15,
[1,3]],
[16,[3,5]],
[17,[8,1]],[18,[5, 6]],[19,[9, 2]],[20,[6,10]],
[21,[2,0]],
[22,[6,1]],[23,[9,9]],[24,[5,5]]]
```

After six steps, we were at infinity mod 19, but it takes 14 steps to reach infinity mod 11. To find $6P$, we needed to invert a number that was 0 mod 19 and nonzero mod 11. This couldn't be done, but it yielded the factor 19. This is the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and a point on each curve. For example, let's take $P = (2, 4)$ and the elliptic curve

$$y^2 \equiv x^3 - 10x + b \pmod{193279}.$$

For P to lie on the curve, we take $b = 28$. We'll also take

$$\begin{aligned}y^2 &\equiv x^3 + 11x - 11, & P &= (1, 1) \\y^2 &\equiv x^3 + 17x - 14, & P &= (1, 2).\end{aligned}$$

Now we compute multiples of the point P . We do the analog of the $p - 1$ method, so we choose a bound B , say $B = 12$, and compute $B!P$.

```
> multell([2,4], factorial(12), -10, 28, 193279)
["factor=",347]

> multell([1,1], factorial(12), 11, -11, 193279)
[13862,35249]

> multell([1,2], factorial(12), 17, -14, 193279)
["factor=",557]
```

Let's analyze in more detail what happened in these examples.

On the first curve, $266P$ ends up at infinity mod 557 and $35P$ is infinity mod 347. Since $266 = 2 \cdot 7 \cdot 19$, it has a prime factor larger than $B = 12$, so $B!P$ is not infinity mod 557. But 35 divides $B!$, so $B!P$ is infinity mod 347.

On the second curve, $356P = \text{infinity mod } 347$ and $561P = \text{infinity mod } 557$. Since $356 = 4 \cdot 89$ and $561 = 3 \cdot 11 \cdot 17$, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have $331P = \text{infinity mod } 347$ and $272P = \text{infinity mod } 557$. Since 331 is prime and $272 = 16 \cdot 17$, we don't expect to find the factorization with this curve. However, by chance, an intermediate step in the calculation of $B!P$ yielded the factorization. Here's what happened. At one step, the program required adding the points $(184993, 13462)$ and $(20678, 150484)$. These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line

through these two points is defined mod 347 but is 0/0 mod 557. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from [Section 21.5](#). For more details, see the text. The elliptic curve is $y^2 \equiv x^3 + 3x + 45 \pmod{8831}$ and the point is $G = (4, 11)$. Alice's message is the point $P_m = (5, 1743)$.

Bob has chosen his secret random number $a_B = 3$ and has computed $a_b G$:

```
> multell([4,11], 3, 3, 45, 8831)
[413,1808]
```

Bob publishes this point. Alice chooses the random number $k = 8$ and computes kG and $P_m + k(a_B G)$:

```
> multell([4,11], 8, 3, 45, 8831)
[5415,6321]
>
addell([5,1743],multell([413,1808],8,3,45,8831),3
,45,8831)
[6626,3576]
```

Alice sends (5415,6321) and (6626,3576) to Bob, who multiplies the first of these point by a_B :

```
> multell([5415,6321], 3, 3, 45, 8831)
```

```
[673,146]
```

Bob then subtracts the result from the last point Alice sends him. Note that he subtracts by adding the point with the second coordinate negated:

```
> addell([6626,3576], [673,-146], 3, 45, 8831)
```

```
[5,1743]
```

Bob has therefore received Alice's message.

Example 48

Let's reproduce the numbers in the example of a Diffie-Hellman key exchange from [Section 21.5](#): The elliptic curve is $y^2 \equiv x^3 + x + 7206 \pmod{7211}$ and the point is $G = (3, 5)$. Alice chooses her secret $N_A = 12$ and Bob chooses his secret $N_B = 23$. Alice calculates

```
> multell([3,5], 12, 1, 7206, 7211)
```

```
[1794,6375]
```

She sends (1794,6375) to Bob. Meanwhile, Bob calculates

```
> multell([3,5], 23, 1, 7206, 7211)
```

```
[3861, 1242]
```

and sends (3861,1242) to Alice. Alice multiplies what she receives by N_A and Bob multiplies what he receives by N_B :

```
> multell([3861,1242], 12, 1, 7206, 7211)
[1472,2098]

> multell([1794,6375], 23, 1, 7206, 7211)
[1472,2098]
```

Therefore, Alice and Bob have produced the same key.

Appendix C MATLAB[®] Examples

These computer examples are written for MATLAB. If you have MATLAB available, you should try some of them on your computer. For information on getting started with MATLAB, see [Section C.1](#). Several functions have been written to allow for experimentation with MATLAB. The MATLAB functions associated with this book are available at

bit.ly/2HyvR8n

We recommend that you create a directory or folder to store these files and download them to that directory or folder. One method for using these functions is to launch MATLAB from the directory where the files are stored, or launch MATLAB and change the current directory to where the files are stored. In some versions of MATLAB the working directory can be changed by changing the current directory on the command bar. Alternatively, one can add the path to that directory in the MATLAB path by using the *path* function or the Set Path option from the File menu on the command bar.

If MATLAB is not available, it is still possible to read the examples. They provide examples for several of the concepts presented in the book. Most of the examples used in the MATLAB appendix are similar to the examples in the Mathematica and Maple appendices. MATLAB, however, is limited in the size of the numbers it can handle. The maximum number that MATLAB can represent in its default mode is roughly 16 digits and larger numbers are approximated. Therefore, it is

necessary to use the symbolic mode in MATLAB for some of the examples used in this book.

A final note before we begin. It may be useful when doing the MATLAB exercises to change the formatting of your display. The command

```
>> format rat
```

sets the formatting to represent numbers using a fractional representation. The conventional *short* format represents large numbers in scientific notation, which often doesn't display some of the least significant digits. However, in both formats, the calculations, when not in symbolic mode, are done in floating point decimals, and then the rational format changes the answers to rational numbers approximating these decimals.

C.1 Getting Started with MATLAB

MATLAB is a programming language for performing technical computations. It is a powerful language that has become very popular and is rapidly becoming a standard instructional language for courses in mathematics, science, and engineering. MATLAB is available on most campuses, and many universities have site licenses allowing MATLAB to be installed on any machine on campus.

In order to launch MATLAB on a PC, double click on the MATLAB icon. If you want to run MATLAB on a Unix system, type *matlab* at the prompt. Upon launching MATLAB, you will see the MATLAB prompt:

```
>>
```

which indicates that MATLAB is waiting for a command for you to type in. When you wish to quit MATLAB, type *quit* at the command prompt.

MATLAB is able to do the basic arithmetic operations such as addition, subtraction, multiplication, and division. These can be accomplished by the operators +, -, *, and /, respectively. In order to raise a number to a power, we use the operator ^ . Let us look at an example:

If we type $2^7 + 125/5$ at the prompt and press the *Enter* key

```
>> 2^7 + 125/5
```

then MATLAB will return the answer:

```
ans =  
153
```

Notice that in this example, MATLAB performed the exponentiation first, the division next, and then added the two results. The order of operations used in MATLAB is the one that we have grown up using. We can also use parentheses to change the order in which MATLAB calculates its quantities. The following example exhibits this:

```
>> 11*( (128/(9+7) - 2^(72/12)))  
  
ans =  
-616
```

In these examples, MATLAB has called the result of the calculations *ans*, which is a variable that is used by MATLAB to store the output of a computation. It is possible to assign the result of a computation to a specific variable. For example,

```
>> spot=17  
  
spot =  
17
```

assigns the value of 17 to the variable *spot*. It is possible to use variables in computations:

```
>> dog=11  
  
dog =  
11  
  
>> cat=7  
  
cat =  
7  
  
>> animals=dog+cat
```

```
animals =  
18
```

MATLAB also operates like an advanced scientific calculator since it has many functions available to it. For example, we can do the standard operation of taking a square root by using the *sqrt* function, as in the following example:

```
>> sqrt(1024)  
  
ans =  
32
```

There are many other functions available. Some functions that will be useful for this book are *mod*, *factorial*, *factor*, *prod*, and *size*.

Help is available in MATLAB. You may either type *help* at the prompt, or pull down the Help menu. MATLAB also provides help from the command line by typing *help commandname*. For example, to get help on the function *mod*, which we shall be using a lot, type the following:

```
>> help mod
```

MATLAB has a collection of toolboxes available. The toolboxes consist of collections of functions that implement many application-specific tasks. For example, the Optimization toolbox provides a collection of functions that do linear and nonlinear optimization. Generally, not all toolboxes are available. However, for our purposes, this is not a problem since we will only need general MATLAB functions and have built our own functions to explore the number theory behind cryptography.

The basic data type used in MATLAB is the matrix. The MATLAB programming language has been written to use

matrices and vectors as the most fundamental data type. This is natural since many mathematical and scientific problems lend themselves to using matrices and vectors.

Let us start by giving an example of how one enters a matrix in MATLAB. Suppose we wish to enter the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \end{bmatrix}$$

into MATLAB. To do this we type:

```
>> A = [1 1 1 1; 1 2 4 8; 1 3 9 27; 1 4 16 64]
```

at the prompt. MATLAB returns

```
A =
1 1 1 1
1 2 4 8
1 3 9 27
1 4 16 64
```

There are a few basic rules that are used when entering matrices or vectors. First, a vector or matrix is started by using a square bracket [and ended using a square bracket]. Next, blanks or commas separate the elements of a row. A semicolon is used to end each row. Finally, we may place a semicolon at the very end to prevent MATLAB from displaying the output of the command.

To define a row vector, use blanks or commas. For example,

```
>> x = [2, 4, 6, 8, 10, 12]
x =
2 4 6 8 10 12
```

To define a column vector, use semicolons. For example,

```
>> y=[1;3;5;7]  
  
y =  
    1  
    3  
    5  
    7
```

In order to access a particular element of y , put the desired index in parentheses. For example, $y(1) = 1$, $y(2) = 3$, and so on.

MATLAB provides a useful notation for addressing multiple elements at the same time. For example, to access the third, fourth, and fifth elements of x , we would type

```
>> x(3:5)  
  
ans =  
    6    8    10
```

The $3:5$ tells MATLAB to start at 3 and count up to 5. To access every second element of x , you can do this by

```
>> x(1:2:6)  
  
ans =  
    2    6    10
```

We may do this for the array also. For example,

```
>> A(1:2:4,2:2:4)  
  
ans =  
    1    1  
    3   27
```

The notation `1:n` may also be used to assign to a variable.
For example,

```
>> x=1:7
```

returns

```
x =
1 2 3 4 5 6 7
```

MATLAB provides the `size` function to determine the dimensions of a vector or matrix variable. For example, if we want the dimensions of the matrix A that we entered earlier, then we would do

```
>> size(A)
ans =
4 4
```

It is often necessary to display numbers in different formats. MATLAB provides several output formats for displaying the result of a computation. To find a list of formats available, type

```
>> help format
```

The `short` format is the default format and is very convenient for doing many computations. However, in this book, we will be representing long whole numbers, and the `short` format will cut off some of the trailing digits in a number. For example,

```
>> a=1234567899
a =
1.2346e+009
```

Instead of using the *short* format, we shall use the *rational* format. To switch MATLAB to using the rational format, type

```
>> format rat
```

As an example, if we do the same example as before, we now get different results:

```
>> a=1234567899  
a =  
1234567899
```

This format is also useful because it allows us to represent fractions in their fractional form, for example,

```
>> 111/323  
ans =  
111/323
```

In many situations, it will be convenient to suppress the results of a computation. In order to have MATLAB suppress printing out the results of a command, a semicolon must follow the command. Also, multiple commands may be entered on the same line by separating them with commas. For example,

```
>> dogs=11, cats=7; elephants=3, zebras=19;  
dogs =  
11  
elephants =  
3
```

returns the values for the variables *dogs* and *elephants* but does not display the values for *cats* and *zebras*.

MATLAB can also handle variables that are made of text. A string is treated as an array of characters. To assign a string to a variable, enclose the text with single quotes. For example,

```
>> txt='How are you today?'
```

returns

```
txt =  
How are you today?
```

A string has size much like a vector does. For example, the size of the variable txt is given by

```
>> size(txt)  
ans =  
1 18
```

It is possible to edit the characters one by one. For example, the following command changes the first word of txt:

```
>> txt(1)='W'; txt(2)='h'; txt(3)='o'  
  
txt =  
Who are you today?
```

As you work in MATLAB, it will remember the commands you have entered as well as the values of the variables you have created. To scroll through your previous commands, press the up-arrow and down-arrow. In order to see the variables you have created, type *who* at the prompt. A similar command *whos* gives the variables, their size, and their type information.

Notes. 1. To use the commands that have been written for the examples, you should run MATLAB in the

directory into which you have downloaded the file from
the Web site bit.ly/2HyvR8n

2. Some of the examples and computer problems use long ciphertexts, etc. For convenience, these have been stored in the file `ciphertexts.m`, which can be loaded by typing `ciphertexts` at the prompt. The ciphertexts can then be referred to by their names. For example, see [Computer Example 4 for Chapter 2](#).

C.2 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext kddkmu.

Decrypt it by trying all possibilities.

```
>> allshift('kddkmu')
```

```
kddkmu
leelnv
mffmow
nggnpx
ohhoqy
piiprz
qjjqsa
rkkrtb
sllsuc
tmmmtvd
unnuwe
voovxf
wppwyg
xqqxzh
yrryai
zsszbj
attack
buubdl
cvvcem
dwwdfn
exxego
fyffhp
gzzgiq
haahjr
ibbiks
jccjlt
```

As you can see, `attack` is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message `cleopatra` using the affine function $7x + 8$:

```
>> affinecrypt('cleopatra',7,8)  
ans =  
'whkcjilxi'
```

Example 3

The ciphertext `mzdvezc` was encrypted using the affine function $5x + 12$. Decrypt it.

SOLUTION

First, solve $y \equiv 5x + 12 \pmod{26}$ for x to obtain $x \equiv 5^{-1}(y - 12)$. We need to find the inverse of 5 (mod 26):

```
>> powermod(5,-1,26)  
ans =  
21
```

Therefore, $x \equiv 21(y - 12) \equiv 21y - 12 \cdot 21$. To change $-12 \cdot 21$ to standard form:

```
>> mod(-12*21,26)  
ans =  
8
```

Therefore, the decryption function is $x \equiv 21y + 8$. To decrypt the message:

```
>> affinecrypt('mzdvezc',21,8)  
ans =  
  
'anthony'
```

In case you were wondering, the plaintext was encrypted as follows:

```
>> affinecrypt('anthony',5,12)  
ans =  
  
'mzdvezc'
```

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in [Section 2.3](#) to decrypt the ciphertext. In the file `ciphertexts.m`, the ciphertext is stored under the name `vvhq`. If you haven't already done so, load the file `ciphertexts.m`:

```
>> ciphertexts
```

Now we can use the variable `vvhq` to obtain the ciphertext:

```
>> vvhq  
  
vvhqwwvrhmustgjgthkihtssejchlsfcbgvwcrlyqtfsvgahw  
kcuuhwauglqhnslrljs  
hbltspisprdxljsveeghlqwkasskuwepwqtwvsgoelkcqyfn  
svwljsniqkgngnrgybw1  
wgoviokhkazkqkxzgyhcecmeiujojqkwfwefqhkijrclrlkbi  
enqfrjljsdhgrhlsfq  
twlauqrhwdmwlgusgikkflryvcwvsgpmlkassjvoqxeggvey  
ggzmljcxxljsvpaivw
```

```
ikvrdrygfrjljslveggveygeiapuuisfpbtgnwwmuczrvtwg  
lrwugumnczvile
```

We now find the frequencies of the letters in the ciphertext. We use the function *frequency*. The *frequency* command was written to display automatically the letter and the count next to it. We therefore have put a semicolon at the end of the command to prevent MATLAB from displaying the count twice.

```
>> fr=frequency(vvhq);  
a 8  
b 5  
c 12  
d 4  
e 15  
f 10  
g 27  
h 16  
i 13  
j 14  
k 17  
l 25  
m 7  
n 7  
o 5  
p 9  
q 14  
r 17  
s 24  
t 8  
u 12  
v 22  
w 22  
x 5  
y 8  
z 5
```

Let's compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

```
>> coinc(vvhq,1)  
ans =  
14
```

```
>> coinc(vvhq,2)
```

```
ans =  
14
```

```
>> coinc(vvhq,3)
```

```
ans =  
16
```

```
>> coinc(vvhq,4)
```

```
ans =  
14
```

```
>> coinc(vvhq,5)
```

```
ans =  
24
```

```
>> coinc(vvhq,6)
```

```
ans =  
12
```

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to 1 mod 5). The function *choose* will do this for us. The function *choose(txt,m,n)* extracts every letter from the string txt that has positions congruent to n mod m.

```
>> choose(vvhq,5,1)
```

```
ans =  
vvutcccqgcunjtpjgkuqpknjkkygkkgcjfqrkqjrqudukvpkv  
ggjjivgjgg pfncwuce
```

We now do a frequency count of the preceding substring. To do this, we use the *frequency* function and use ans as input. In MATLAB, if a command is issued without declaring a variable for the result, MATLAB will put the output in the variable ans.

```
>> frequency(ans);
```

```
a    0
b    0
c    7
d    1
e    1
f    2
g    9
h    0
i    1
j    8
k    8
l    0
m    0
n    3
o    0
p    4
q    5
r    2
s    0
t    3
u    6
v    5
w    1
x    0
y    1
z    0
```

To express this as a vector of frequencies, we use the *vigvec* function. The *vigvec* function will not only display the frequency counts just shown, but will return a vector that contains the frequencies. In the following output, we have suppressed the table of frequency counts since they appear above and have reported the results in the *short* format.

```
>> vigvec(vvhq,5,1)
```

```
ans =
0
0
0.1045
0.0149
0.0149
0.0299
0.1343
```

```
0  
0.0149  
0.1194  
0.1194  
0  
0  
0.0448  
0  
0.0597  
0.0746  
0.0299  
0  
0.0448  
0.0896  
0.0746  
0.0149  
0  
0.0149  
0
```

(If we are working in rational format, these numbers are displayed as rationals.) The dot products of this vector with the displacements of the alphabet frequency vector are computed as follows:

```
>> corr(ans)  
  
ans =  
0.0250  
0.0391  
0.0713  
0.0388  
0.0275  
0.0380  
0.0512  
0.0301  
0.0325  
0.0430  
0.0338  
0.0299  
0.0343  
0.0446  
0.0356  
0.0402  
0.0434  
0.0502  
0.0392  
0.0296  
0.0326  
0.0392
```

```
0.0366  
0.0316  
0.0488  
0.0349
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

```
>> max(ans)  
  
ans =  
0.0713
```

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter *c*. A procedure similar to the one just used (using *vigvec(vvhq, 5,2), . . . , vigvec(vvhq,5,5)*) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

```
>> vigenere(vvhq, -[2,14,3,4,18])  
  
ans =  
  
themethodusedfortheprparationandreadingofcodemes  
sagesissimpleinthe  
extremeandatthesametimeimpossibleoftranslationunl  
essthekeyisknownth  
eeasewithwhichthekeymaybechangedisanotherpointinf  
avoroftheadoptiono  
fthiscodebythosedesiringtotransmitimportantmessag  
eswithouttheslight  
estdangeroftheirmessagesbeingreadbypoliticalorbus  
inessrivalsetc
```

For the record, the plaintext was originally encrypted by the command

```
>> vigenere(ans, [2,14,3,4,18])  
  
ans =
```

vhqvvvhmusggjgthkihtssejchlsfcbgvwcrlyqtfsvgahw
kuhwauglqhnsrljs
hbltspisprdxljsveeghlqwkasskuwepwqtwwspgoelkcqyfn
svwljsniqkgnrgybw1
wgoviokhkazkqkxzgyhcecmeyiujoqkwfwefqhkijrclrlkbi
enqfrjljsdhgrhlsfq
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxeggvey
ggzmljcxxljsvpaivw
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrvtwg
lrwugumnczvile

C.3 Examples for Chapter 3

Example 5

Find $\gcd(23456; 987654)$.

```
>> gcd(23456,987654)  
ans =  
2
```

If larger integers are used, they should be expressed in symbolic mode; otherwise, only the first 16 digits of the entries are used accurately. The present calculation could have been done as

```
>> gcd(sym('23456'),sym('987654'))  
ans =  
2
```

Example 6

Solve $23456x + 987654y = 2$ in integers x, y .

```
>> [a,b,c]=gcd(23456,987654)  
a =  
2  
b =  
-3158  
c =  
75
```

This means that 2 is the gcd and
 $23456 \cdot (-3158) + 987654 \cdot 75 = 2$.

Example 7

Compute $234 \cdot 456 \pmod{789}$.

```
>> mod(234*456,789)
```

```
ans =
189
```

Example 8

Compute $234567^{876543} \pmod{565656565}$.

```
>>
powermod(sym('234567'),sym('876543'),sym('5656565
65'))
```

```
ans =
5334
```

Example 9

Find the multiplicative inverse of
 $87878787 \pmod{9191919191}$.

```
>> invmodn(sym('87878787'),sym('9191919191'))
```

```
ans =
7079995354
```

Example 10

Solve $7654x \equiv 2389 \pmod{65537}$.

SOLUTION

To solve this problem, we follow the method described in [Section 3.3](#). We calculate 7654^{-1} and then multiply it by 2389:

```
>> invmodn(7654,65537)  
  
ans =  
      54637  
  
>> mod(ans*2389,65537)  
  
ans =  
      43626
```

Example 11

Find x with

$$x \equiv 2 \pmod{78}, \quad x \equiv 5 \pmod{97}, \quad x \equiv 1 \pmod{119}.$$

SOLUTION

To solve the problem we use the function *crt*.

```
>> crt([2 5 1],[78 97 119])  
  
ans =  
      647480
```

We can check the answer:

```
>> mod(647480,[78 97 119])  
  
ans =  
      2     5     1
```

Example 12

Factor 123450 into primes.

```
>> factor(123450)  
  
ans =  
2 3 5 5 823
```

This means that $123450 = 2^1 3^1 5^2 823^1$.

Example 13

Evaluate $\phi(12345)$.

```
>> eulerphi(12345)  
  
ans =  
6576
```

Example 14

Find a primitive root for the prime 65537.

```
>> primitiveroot(65537)  
  
ans =  
3
```

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix

$$\begin{pmatrix} 13 & 12 & 35 \\ 41 & 53 & 62 \\ 71 & 68 & 10 \end{pmatrix} \quad (\text{mod } 999).$$

SOLUTION

First, we enter the matrix as M .

```
>> M=[13 12 35; 41 53 62; 71 68 10];
```

Next, invert the matrix without the mod:

```
>> Minv=inv(M)

Minv =
233/2158      -539/8142      103/3165
-270/2309     139/2015       -40/2171
209/7318      32/34139      -197/34139
```

We need to multiply by the determinant of M in order to clear the fractions out of the numbers in $Minv$. Then we need to multiply by the inverse of the determinant mod 999.

```
>> Mdet=det(M)

Mdet =
-34139

>> invmodn(Mdet,999)

ans =
589
```

The answer is given by

```
>> mod(Minv*589*Mdet,999)

ans =
772      472      965
```

641	516	851
150	133	149

Therefore, the inverse matrix mod 999 is

$$\begin{pmatrix} 772 & 472 & 965 \\ 641 & 516 & 851 \\ 150 & 133 & 149 \end{pmatrix}.$$

In many cases, it is possible to determine by inspection the common denominator that must be removed. When this is not the case, note that the determinant of the original matrix will always work as a common denominator.

Example 16

Find a square root of 26951623672 mod the prime $p = 98573007539$.

SOLUTION

Since $p \equiv 3 \pmod{4}$, we can use the proposition of Section 3.9:

```
>> powermod(sym('26951623672'),
  (sym('98573007539')+1)/4,sym('98573007539'))  
  
ans =  
98338017685
```

The other square root is minus this one:

```
>> mod(-ans,32579)  
  
ans =  
234989854
```

Example 17

Let $n = 34222273 = 9803 \cdot 3491$. Find all four solutions of $x^2 \equiv 19101358 \pmod{34222273}$.

SOLUTION

First, find a square root mod each of the two prime factors, both of which are congruent to $3 \pmod{4}$:

```
>> powermod(19101358,(9803+1)/4,9803)  
  
ans =  
    3998  
>> powermod(19101358,(3491+1)/4,3491)  
  
ans =  
    1318
```

Therefore, the square roots are congruent to $\pm 3998 \pmod{9803}$ and are congruent to $\pm 1318 \pmod{3491}$. There are four ways to combine these using the Chinese remainder theorem:

```
>> crt([3998 1318],[9803 3491])  
  
ans =  
    43210  
  
>> crt([-3998 1318],[9803 3491])  
  
ans =  
    8397173  
  
>> crt([3998 -1318],[9803 3491])  
  
ans =  
    25825100  
  
>> crt([-3998 -1318],[9803 3491])  
  
ans =  
    34179063
```

These are the four desired square roots.

C.4 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod{2}.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is [1, 0, 1, 0, 0] and the initial values are given by the vector [0, 1, 0, 0, 0]. Type

```
>> lfsr([1 0 1 0 0],[0 1 0 0 0],50)

ans =
Columns 1 through 12
0 1 0 0 0 0 1 0 0 1 0 1
Columns 13 through 24
1 0 0 1 1 1 1 1 0 0 0 1
Columns 25 through 36
1 0 1 1 1 0 1 0 1 0 0 0
Columns 37 through 48
0 1 0 0 1 0 1 1 0 0 1 1
Columns 49 through 50
1 1
```

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recursion that generates this sequence.

SOLUTION

First, we find a candidate for the length of the recurrence. The command *lfsrlength*(*v*, *n*) calculates the determinants mod 2 of the first *n* matrices that appear in the procedure described in [Section 5.2](#) for the sequence *v*. Recall that the last nonzero determinant gives the length of the recurrence.

```
>> lfsrlength([1 0 1 0 1 1 1 0 0 0 0 1 1 1 0 1 0
1 0 1],10)
Order Determinant
 1      1
 2      1
 3      0
 4      1
 5      0
 6      1
 7      0
 8      0
 9      0
10      0
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

```
>> lfsrsolve([1 0 1 0 1 1 1 0 0 0 0 1 1 1 0 1 0 1
0 1],6)

ans =
 1  0  1  1  1  0
```

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod{2}.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR

output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning of the LFSR output:

```
>> x=mod([1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0]+[0 1  
1 0 1 0 1 0 0 1 1 0 0 0 1],2)  
  
x =  
Columns 1 through 12  
1 0 0 1 0 1 1 1 0 1 0 0  
Columns 13 through 17  
0 1 1 0 1
```

This is the beginning of the LFSR output. Let's find the length of the recurrence:

```
>> lfsrlength(x,8)  
Order Determinant  
1 1  
2 0  
3 1  
4 0  
5 1  
6 0  
7 0  
8 0
```

We guess the length is 5. To find the coefficients of the recurrence:

```
>> lfsrsolve(x,5)  
  
ans =  
1 1 0 0 1
```

Now we can generate the full output of the LFSR using the coefficients we just found plus the first five terms of the LFSR output:

```
>> lfsr([1 1 0 0 1],[1 0 0 1 0],40)

ans =
Columns 1 through 12
1 0 0 1 0 1 1 0 1 0 0 1
Columns 13 through 24
0 1 1 0 1 0 0 1 0 1 1 0
Columns 25 through 36
1 0 0 1 0 1 1 0 1 0 0 1
Columns 37 through 40
0 1 1 0
```

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

```
>> mod(ans+[0 1 1 0 1 0 1 0 1 0 0 1 1 1 0 0 0 1 0 1
0 1 0 1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0],2)

ans =
Columns 1 through 12
1 1 1 1 1 1 0 0 0 0 0 0
Columns 13 through 24
1 1 1 0 0 0 1 1 1 1 0 0
Columns 25 through 36
0 0 1 1 1 1 1 1 0 0 0 0
Columns 37 through 40
0 0 0 0
```

This is the plaintext.

C.5 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}.$$

Decrypt it.

SOLUTION

A matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is entered as $[a, b; c, d]$. Type

$M * N$ to multiply matrices M and N . Type $v * M$ to multiply a vector v on the right by a matrix M .

First, we put the above matrix in the variable M .

```
>> M=[1 2 3; 4 5 6; 7 8 10]
```

M =

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{matrix}$$

Next, we need to invert the matrix mod 26:

```
>> Minv=inv(M)
```

Minv =

$$\begin{matrix} -2/3 & -4/3 & 1 \\ -2/3 & 11/3 & -2 \\ 1 & -2 & 1 \end{matrix}$$

Since we are working mod 26, we can't stop with numbers like $2/3$. We need to get rid of the denominators and reduce mod 26. To do so, we multiply by 3 to extract the numerators of the fractions, then multiply by the inverse of 3 mod 26 to put the "denominators" back in (see [Section 3.3](#)):

```
>> M1=Minv*3
```

M1 =

$$\begin{matrix} -2 & -4 & 3 \\ -2 & 11 & -6 \\ 3 & -6 & 3 \end{matrix}$$

```
>> M2=mod(round(M1*9,26))
```

M2 =

$$\begin{matrix} 8 & 16 & 1 \\ 8 & 21 & 24 \\ 1 & 24 & 1 \end{matrix}$$

Note that we used the function *round* in calculating $M2$. This was done since MATLAB performs its calculations in floating point and calculating the inverse matrix $Minv$ produces numbers that are slightly different from whole numbers. For example, consider the following:

```
>> a=1.99999999;display([a, mod(a,2),  
mod(round(a),2)])
```

```
2.0000 2.0000 0
```

The matrix $M2$ is the inverse of the matrix M mod 26. We can check this as follows:

```
>> mod(M2*M,26)  
ans =  
1 0 0  
0 1 0  
0 0 1
```

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

```
>> mod([22,9,0]*M2,26)
```

```
ans =  
14 21 4
```

```
>> mod([12,3,1]*M2,26)
```

```
ans =  
17 19 7
```

```
>> mod([10,3,4]*M2,26)
```

```
ans =  
4 7 8
```

```
>> mod([8,1,17]*M2,26)
```

```
ans =  
11 11 23
```

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. This can be changed back to letters:

```
>> int2text([14 21 4 17 19 7 4 7 8 11 11 23])  
ans =  
overthehillx
```

Note that the final x was appended to the plaintext in order to complete a block of three letters.

C.6 Examples for Chapter 9

Example 22

Two functions, *nextprime* and *randprime*, can be used to generate prime numbers. The function *nextprime* takes a number n as input and attempts to find the next prime after n . The function *randprime* takes a number n as input and attempts to find a random prime between 1 and n . It uses the Miller-Rabin test described in Chapter 9.

```
>> nextprime(346735)  
  
ans =  
346739  
  
>> randprime(888888)  
  
ans =  
737309
```

For larger inputs, use symbolic mode:

It is interesting to note the difference that the ' ' makes when entering a large integer:

```
>> nextprime(sym('123456789012345678901234567890'))
ans =
123456789012345678901234567907

>> nextprime(sym(123456789012345678901234567890))
ans =
123456789012345677877719597071
```

In the second case, the input was a number, so only the first 16 digits of the input were used correctly when changing to symbolic mode, while the first case regarded the entire input as a string and therefore used all of the digits.

Example 23

Suppose you want to change the text hellohowareyou to numbers:

```
>> text2int1('hellohowareyou')
ans =
805121215081523011805251521
```

Note that we are now using $a = 1, b = 2, \dots, z = 26$, since otherwise a 's at the beginnings of messages would disappear. (A more efficient procedure would be to work in base 27, so the numerical form of the message would be

$$8 + 5 \cdot 27 + 12 \cdot 27^2 + \dots + 21 \cdot 27^{13} = 87495221502384554951$$

. Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

```
>> int2text1(805121215081523011805251521)  
  
ans =  
'hellohowareyou'
```

Example 24

Encrypt the message `hi` using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the message to numbers:

```
>> text2int1('hi')  
  
ans =  
809
```

Now, raise it to the e th power mod n :

```
>> powermod(ans,17,823091)  
  
ans =  
596912
```

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d . To do this, we need to find $\phi(823091)$. One way is

```
>> eulerphi(823091)
```

```
ans =
821184
```

Another way is to factor n as $p \cdot q$ and then compute $(p - 1)(q - 1)$:

```
>> factor(823091)

ans =
659    1249

>> 658*1248

ans =
821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute the following (note that we are finding the inverse of $e \pmod{\phi(n)}$, not \pmod{n}):

```
>> invmodn(17,821184)

ans =
48305
```

Therefore, $d = 48305$. To decrypt, raise the ciphertext to the d th power mod n :

```
>> powermod(596912,48305,823091)

ans =
809
```

Finally, change back to letters:

```
>> int2text1(ans)

ans =
hi
```

Example 26

Encrypt `hellohowareyou` using RSA with $n = 823091$ and $e = 17$.

SOLUTION

First, change the plaintext to numbers:

```
>> text2int1('hellohowareyou')

ans =
805121215081523011805251521
```

Suppose we simply raised this to the e th power mod n :

```
>> powermod(ans,17,823091)

ans =
447613
```

If we decrypt (we know d from [Example 25](#)), we obtain

```
>> powermod(ans,48305,823091)

ans =
628883
```

This is not the original plaintext. The reason is that the plaintext is larger than n , so we have obtained the plaintext mod n :

```
>> mod(text2int1('hellohowareyou'),823091)

ans =
628883
```

We need to break the plaintext into blocks, each less than n . In our case, we use three letters at a time:

80512 121508 152301 180525 1521

```
>> powermod(80512,17,823091)

ans =
757396

>> powermod(121508,17,823091)

ans =
164513

>> powermod(152301,17,823091)

ans =
121217

>> powermod(180525,17,823091)

ans =
594220

>> powermod(1521,17,823091)

ans =
442163
```

The ciphertext is therefore

757396164513121217594220442163. Note that there is no reason to change this back to letters. In fact, it doesn't correspond to any text with letters.

Decrypt each block individually:

```
>> powermod(757396,48305,823091)

ans =
80512

>> powermod(164513,48305,823091)

ans =
121508
```

Etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in Section

9.5. These are stored under the names *rsan*, *rsaе*, *rsap*, *rsaq*:

```
>> rsan

ans =

114381625757888676692357799761466120102182967212
42362562561842935
7069352457338978305971235639587050589890751475992
90026879543541

>> rsaе

ans =
9007
```

Example 27

Encrypt each of the messages *b*, *ba*, *bar*, *bard* using *rsan* and *rsaе*.

```
>> powermod(text2int1('b'), rsaе, rsan)

ans =

7094675846761266859837016499155078618287633106068
52354105647041144
8678226171649720012215533234846201405328798758089
9263765142534

>> powermod(text2int1('ba'), rsaе, rsan)

ans =

3504513060897510032501170944987195427378820475394
85930603136976982
2762175980602796227053803156556477335203367178226
1305796158951

>> powermod(txt2int1('bar'), rsaе, rsan)

ans =

4481451286385510107600453085949210934242953160660
74090703605434080
0084364598688040595310281831282258636258029878444
```

```
1151922606424  
  
>> powermod(text2int1('bard'), rsae, rsan)  
  
ans =  
  
242380777851166642320286251209031739348521295905  
62707831349916142  
5605432329717980492895807344575266302644987398687  
7989329909498
```

Observe that the ciphertexts are all the same length.
There seems to be no easy way to determine the length of
the corresponding plaintext.

Example 28

Using the factorization $rsan = rsap \cdot rsaq$, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see [Section 9.5](#)).

SOLUTION

First, we find the decryption exponent:

```
>> rsad=invmodn(rsae,-1,(rsap-1)*(rsaq-1));
```

Note that we use the final semicolon to avoid printing out the value. If you want to see the value of $rsad$, see [Section 9.5](#), or don't use the semicolon. To decrypt the ciphertext, which is stored as $rsaci$, and change to letters:

```
>> int2text1(powermod(rsaci, rsad, rsan))  
  
ans =  
the magic words are squeamish ossifrage
```

Example 29

Encrypt the message *rsaencryptsmessageswell* using *rsan* and *rsae*.

```
>> ci =
powermod(text2int1('rsaencryptsmessageswell'),
rsae, rsan)
ci =
9463942034900225931630582353924949641464096993400
17097214043524182
7195065425436558490601396632881775353928311265319
7553130781884
```

We called the ciphertext *ci* because we need it in Example 30.

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption exponent *rsad*. Therefore, we compute

```
>> powermod(ans, rsad, rsan)

ans =
1819010514031825162019130519190107051923051212

>> int2text1(ans)

ans =
rsaencryptsmessageswell
```

Suppose we lose the final 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4): '`>> powermod((ci - 4)/10, rsad, rsan)`'

```
ans =  
4795299917319598866490235262952548640911363389437  
562984685490797  
0588412300373487969657794254117158956921267912628  
461494475682806
```

If we try to change this to letters, we get a weird-looking answer. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that

$n = 11313771275590312567$ is the product of two

primes and that $\phi(n) = 11313771187608744400$.

Factor n .

SOLUTION

We know (see [Section 9.1](#)) that p and q are the roots of

$X^2 - (n - \phi(n) + 1)X + n$. Therefore, we compute

(`vpa` is for variable precision arithmetic)

```
>> digits(50); syms y; vpasolve(y^2-  
(sym('11313771275590312567') -  
sym('11313771187608744400')+1)*y+sym('11313771275  
590312567'),y)  
  
ans =  
128781017.0 87852787151.0
```

Therefore, $n = 128781017 \cdot 87852787151$. We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know $rsae$ and $rsad$. Use these to factor $rsan$.

SOLUTION

We use the $a^r \equiv 1$ factorization method from [Section 9.4](#). First write $rsae \cdot rsad - 1 = 2^s m$ with m odd. One way to do this is first to compute

```
>> rsae*rsad - 1

ans =

9610344196177822661569190233595838341098541290518
783302506446040
411598557508735265915617489855734299513159468043
1086921245830097664

>> ans/2

ans =

4805172098088911330784595116797919170549270645259
391651253223020
2057799278754367632957808744927867149756579734021
5543460622915048832

>> ans/2

ans =

2402586049044455665392297558398959585274635322629
695825626611510
1028899639377183816478904372463933574878289867010
7771730311457524416
```

We continue this way for six more steps until we get

```
ans =

3754040701631961977175464934998374351991617691608
899727541580484
5357655686526849713248288081974896210747327917204
33933286116523819
```

This number is m . Now choose a random integer a .
Hoping to be lucky, we choose 13. As in the $a^r \equiv 1$
factorization method, we compute

```
>>b0=powermod(13, ans, rsan)

b0 =
2757436850700653059224349486884716119842309570730
780569056983964
7030183109839862370800529338092984795490192643587
960859870551239
```

Since this is not $\pm 1 \pmod{rsan}$, we successively
square it until we get ± 1 :

```
>> b1=powermod(b0,2,rsan)

b1 =
4831896032192851558013847641872303455410409906994
084622549470277
6654996412582955636035266156108686431194298574075
854037512277292

>> b2=powermod(b1,2,rsan)

b2 =
7817281415487735657914192805875400002194878705648
382091793062511
5215181839742056013275521913487560944732073516487
722273875579363

>> b3=powermod(b2, 2, rsan)

b3 =
4283619120250872874219929904058290020297622291601
776716755187021
6509444518239462186379470569442055101392992293082
259601738228702

>> b4=powermod(b3, 2, rsan)

b4 =
1
```

Since the last number before the 1 was not $\pm 1 \pmod{rsan}$, we have an example of $x \not\equiv \pm 1 \pmod{rsan}$ with $x^2 \equiv 1$. Therefore, $\gcd(x - 1, rsan)$ is a nontrivial factor of $rsan$:

```
>> gcd(b3 - 1, rsan)
ans =
3276913299326670954996198819083446141317764296799
2942539798288533
```

This is $rsaq$. The other factor is obtained by computing $rsan/rsaq$:

```
>> rsan/ans
ans =
3490529510847650949147849619903898133417764638493
387843990820577
```

This is $rsap$.

Example 33

Suppose you know that

$$150883475569451^2 \equiv 16887570532858^2 \pmod{205611444308117}.$$

Factor 205611444308117 .

SOLUTION

We use the Basic Principle of [Section 9.4](#).

```
>> g= gcd(150883475569451-
16887570532858,205611444308117)
g =
23495881
```

This gives one factor. The other is

```
>> 205611444308117/g  
  
ans =  
8750957
```

We can check that these factors are actually primes, so we can't factor any further:

```
>> primetest(ans)  
  
ans =  
1  
  
>> primetest(g)  
  
ans =  
1
```

Example 34

Factor

$n = 376875575426394855599989992897873239$
by the $p - 1$ method.

SOLUTION

Let's choose our bound as $B = 100$, and let's take $a = 2$, so we compute $2^{100!} \pmod{n}$:

```
>>  
powermod(2,factorial(100),sym('376875575426394855  
59998999 2897873239'))  
  
ans =  
369676678301956331939422106251199512
```

Then we compute the gcd of $2^{100!} - 1$ and n :

```
>> gcd(ans - 1,  
sym('376875575426394855599989992897873239')  
  
ans =  
430553161739796481
```

This is a factor p . The other factor q is

```
>>  
sym('376875575426394855599989992897873239')/ans  
  
ans =  
875328783798732119
```

Let's see why this worked. The factorizations of $p - 1$ and $q - 1$ are

```
>> factor(sym('430553161739796481') - 1)  
  
ans =  
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,  
2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 7, 7, 7, 7, 11,  
11, 11, 47]  
  
>> factor(sym('875328783798732119') - 1)  
  
ans =  
[ 2, 61, 20357, 39301, 8967967]
```

We see that $100!$ is a multiple of $p - 1$, so $2^{100!} \equiv 1 \pmod{p}$. However, $100!$ is not a multiple of $q - 1$, so it is likely that $2^{100!} \not\equiv 1 \pmod{q}$. Therefore, both $2^{100!} - 1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p .

C.7 Examples for Chapter 10

Example 35

Let's solve the discrete log problem $2^x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take $N = 12$ since $N^2 > p - 1 = 130$ and we form two lists. The first is $2^j \pmod{131}$ for $0 \leq j \leq 11$:

```
>> for k=0:11;z=[k,  
powermod(2,k,131)];disp(z);end;  
  
0 1  
1 2  
2 4  
3 8  
4 16  
5 32  
6 64  
7 128  
8 125  
9 119  
10 107  
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \leq j \leq 11$:

```
>> for k=0:11;z=[k,  
mod(71*invmodn(powermod(2,12*k,131),131),131)];  
disp(z);end;  
  
0 71  
1 17  
2 124  
3 26  
4 128  
5 86  
6 111  
7 93  
8 85
```

9	96
10	130
11	116

The number 128 is on both lists, so we see that

$2^7 \equiv 71 \cdot 2^{-12*4} \pmod{131}$. Therefore,

$$71 \equiv 2^{7+4*12} \equiv 2^{55} \pmod{131}.$$

C.8 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod_{i=1}^{22} (1 - i/365)$ (note that the product stops at $i = 22$, not $i = 23$). Subtracting from 1 gives the probability that at least two have the same birthday:

```
>> 1-prod( 1 - (1:22)/365)

ans =
0.5073
```

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
>> 1-prod( 1 - (1:9999)/10^7

ans =
0.9933
```

Note that the number of phones is about three times the square root of the number of possibilities. This means

that we expect the probability to be high, which it is. From [Section 12.1](#), we have the estimate that if there are around $\sqrt{2(\ln 2)10^7} \approx 3723$ phones, there should be a 50% chance of a match. Let's see how accurate this is:

```
>> 1-prod( 1 - (1:3722)/10^7)
```

```
ans =  
0.4999
```

C.9 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme. Everything is mod the prime $p = 987541$. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

The function *interppoly(x,f,m)* calculates the interpolating polynomial that passes through the points (x_j, f_j) . The arithmetic is done mod m .

In order to use this function, we need to make a vector that contains the x values, and another vector that contains the share values. This can be done using the following two commands:

```
>> x=[9853 4421 6543 93293 12398];
>> s=[853 4387 1234 78428 7563];
```

Now we calculate the coefficients for the interpolating polynomial.

```
>> y=interppoly(x,s,987541)
y =
    678987    14728    1651    574413    456741
```

The first value corresponds to the constant term in the interpolating polynomial and is the secret value.
Therefore, 678987 is the secret.

C.10 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from [Section 18.2](#). There are five cards: ten, jack, queen, king, ace. We have chosen to abbreviate them by the following: ten, ace, que, jac, kin. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 300649. You are supposed to guess which one is the ace.

First, the cards are entered in and converted to numerical values by the following steps:

```
>> cards=['ten';'ace';'que';'jac';'kin'];

>> cvals=text2int1(cards)

cvals =
```



```
200514
10305
172105
100103
110914
```

Next, we pick a random exponent k that will be used in the hiding operation. We use the semicolon after *khide* so that we cannot cheat and see what value of k is being used.

```
>> p=300649;
```

```
>> k=khide(p);
```

Now, shuffle the disguised cards (their numbers are raised to the k th power mod p and then randomly permuted):

```
>> shufvals=shuffle(cvals,k,p)

shufvals =
226536
226058
241033
281258
116809
```

These are the five cards. None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

```
>> reveal(shufvals,k,p)

ans =

jac
que
ten
kin
ace
```

Let's play again:

```
>> k=khide(p);

» shufvals=shuffle(cvals,k,p)

shufvals =
117135
144487
108150
266322
264045
```

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

```
>> reveal(shufvals,k,p)

ans =

kin
jac
ten
que
ace
```

Perhaps you need some help. Let's play one more time:

```
>> k=khide(p);

» shufvals=shuffle(cvals,k,p)

shufvals =
    108150
    144487
    266322
    264045
    117135
```

We now ask for advice:

```
>> advise(shufvals,p);
```

```
Ace Index: 4
```

We are advised that the fourth card is the ace. Let's see:

```
>> reveal(shufvals,k,p)

ans =

ten
jac
que
```

```
ace  
kin
```

How does this work? Read the part on “How to Cheat” in [Section 18.2](#). Note that if we raise the numbers for the cards to the $(p - 1)/2$ power mod p , we get

```
>> powermod(cvals,(p-1)/2,p)  
  
ans =  
     1  
300648  
     1  
     1  
     1
```

Therefore, only the ace is a quadratic nonresidue mod p .

C.11 Examples for Chapter 21

Example 40

We want to graph the elliptic curve

$$y^2 = x(x - 1)(x + 1).$$

First, we create a string v that contains the equation we wish to graph.

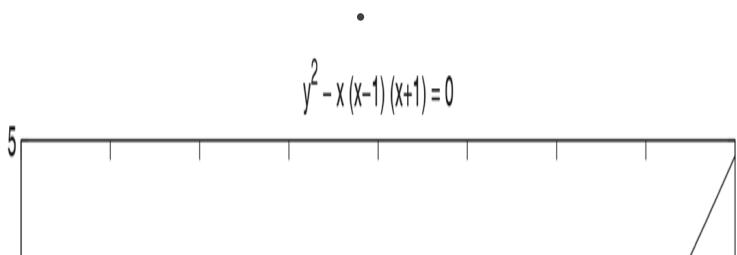
```
>> v='y^2 - x*(x-1)*(x+1)';
```

Next we use the *ezplot* command to plot the elliptic curve.

```
>> ezplot(v, [-1,3,-5,5])
```

The plot appears in Figure C.1. The use of $[-1, 3, -5, 5]$ in the preceding command is to define the limits of the x -axis and y -axis in the plot.

Figure C.1 Graph of the Elliptic Curve $y^2 = x(x - 1)(x + 1)$



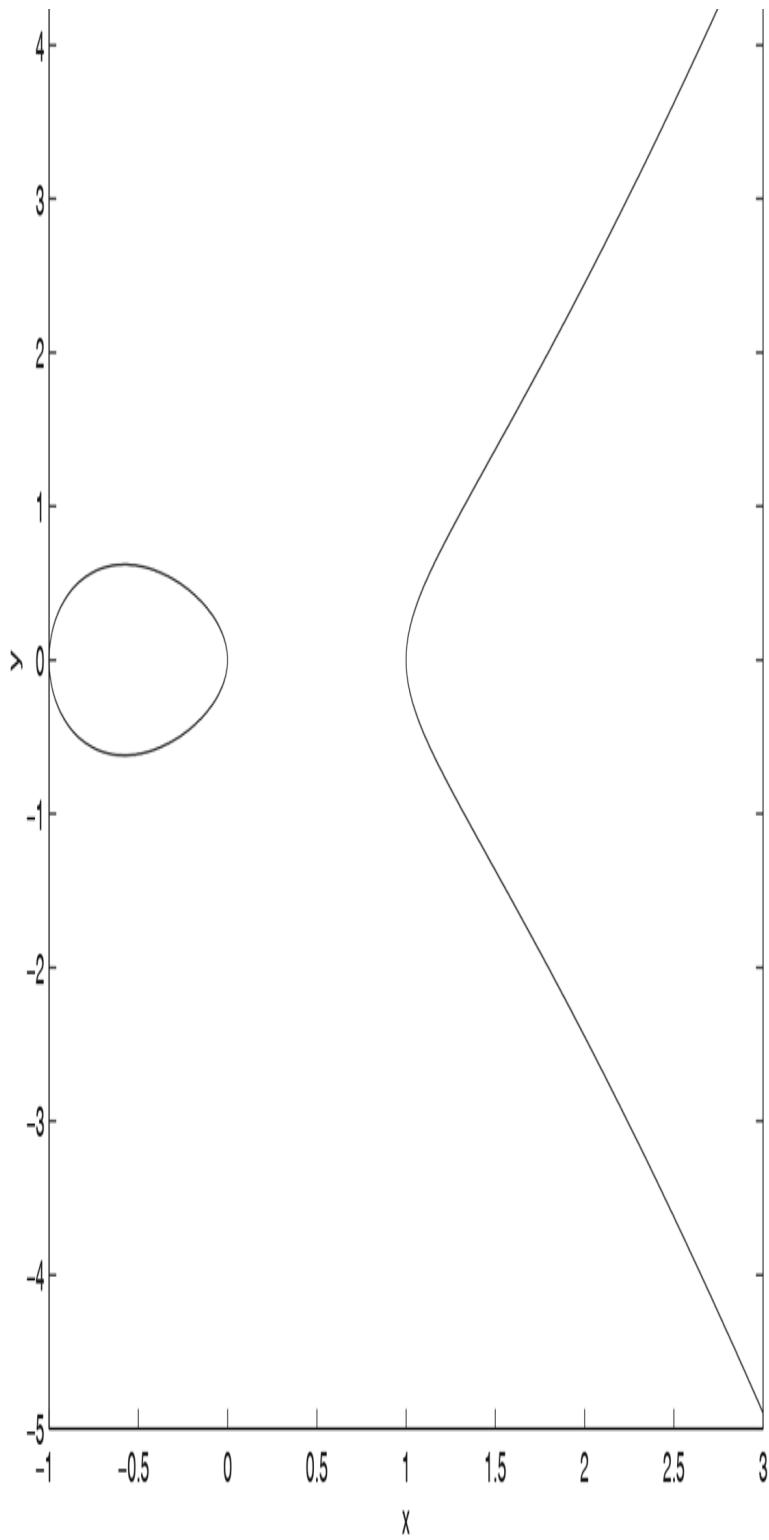


Figure C.1 Full Alternative Text

Example 41

Add the points $(1,3)$ and $(3,5)$ on the elliptic curve
 $y^2 \equiv x^3 + 24x + 13 \pmod{29}$.

```
>> addell([1,3],[3,5],24,13,29)  
ans =  
26      1
```

You can check that the point $(26,1)$ is on the curve:
 $26^3 + 24 \cdot 26 + 13 \equiv 1^2 \pmod{29}$. (Note:
`addell([x,y],[u,v],b,c,n)` is only programmed
to work for odd n .)

Example 42

Add $(1,3)$ to the point at infinity on the curve of the
previous example.

```
>> addell([1,3],[inf,inf],24,13,29)  
ans =  
1      3
```

As expected, adding the point at infinity to a point P
returns the point P .

Example 43

Let $P = (1, 3)$ be a point on the elliptic curve
 $y^2 \equiv x^3 + 24x + 13 \pmod{29}$. Find $7P$.

```
>> multell([1,3],7,24,13,29)  
ans =  
15      6
```

Example 44

Find $k(1, 3)$ for $k = 1, 2, 3, \dots, 40$ on the curve of the previous example.

```
>> multsell([1,3],40,24,13,29)

ans =
 1:    1    3
 2:   11   10
 3:   23   28
 4:    0   10
 5:   19    9
 6:   18   19
 7:   15    6
 8:   20   24
 9:    4   12
10:   4   17
11:   20    5
12:   15   23
13:   18   10
14:   19   22
15:    0   19
16:   23    1
17:   11   19
18:    1   26
19:   inf   Inf
20:    1    3
21:   10   10
22:   23   28
23:    0   10
24:   19    7
25:   18   19
26:   15    6
27:   20   24
28:    4   12
29:    4   17
30:   20    5
31:   15   23
32:   18   10
33:   19   22
34:    0   19
35:   23    1
36:   11   19
37:    1   26
38:   inf   inf
39:    1    3
40:   10   10
```

Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both factors or we could be at infinity mod one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute $12P$, where $P = (1, 3)$ is on the elliptic curve $y^2 \equiv x^3 - 5x + 13 \pmod{209}$:

```
>> multell([1,3],12,-5,13,11*19)

Elliptic Curve addition produced a factor of n,
factor= 19
Multell found a factor of n and exited

ans =
[]
```

Now let's compute the successive multiples to see what happened along the way:

```
>> multsell([1,3],12,-5,13,11*19)

Elliptic Curve addition produced a factor of n,
factor= 19
Multsell ended early since it found a factor

ans =
1:    1      3
2:   91     27
3:  118    133
4:  148    182
5:   20     35
```

When we computed $6P$, we ended up at infinity mod 19. Let's see what is happening mod the two prime factors of 209, namely 19 and 11:

```
>> multsell([1,3],20,-5,13,19)
```

```
ans =
1:   1   3
2:  15   8
3:   4   0
4:  15  11
5:   1  16
6: Inf Inf
7:   1   3
8:  15   8
9:   4   0
10: 15  11
11: 15   8
12: Inf Inf
13:   1   3
14: 15   8
15:   4   0
16: 15  11
17:   1  16
18: Inf Inf
19:   1   3
20: 15   8
```

```
>> multsell([1,3],20,-5,13,11)
```

```
ans =
1:   1   3
2:   3   5
3:   8   1
4:   5   6
5:   9   2
6:   6  10
7:   2   0
8:   6   1
9:   9   9
10:  5   5
11:  8  10
12:  3   6
13:  1   8
14: Inf Inf
15:   1   3
16:   3   5
17:   8   1
18:   5   6
19:   9   2
20:   6  10
```

After six steps, we were at infinity mod 19, but it takes 14 steps to reach infinity mod 11. To find $6P$, we needed to invert a number that was 0 mod 19 and nonzero mod 11.

This couldn't be done, but it yielded the factor 19. This is the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and a point on each curve. For example, let's take $P = (2, 4)$ and the elliptic curve

$$y^2 \equiv x^3 - 10x + b \pmod{193279}.$$

For P to lie on the curve, we take $b = 28$. We'll also take

$$\begin{aligned} y^2 &\equiv x^3 + 11x - 11, & P &= (1, 1), \\ y^2 &\equiv x^3 + 17x - 14, & P &= (1, 2). \end{aligned}$$

Now we compute multiples of the point P . We do the analog of the $p - 1$ method, so we choose a bound B , say $B = 12$, and compute $B!P$.

```
>> multell([2,4],factorial(12),-10,28,193279)

Elliptic Curve addition produced a factor of n,
factor= 347
Multell found a factor of n and exited

ans =
[]

>> multell([1,1],factorial(12),11,-11,193279)

ans =
13862      35249

» multell([1,2],factorial(12),17,-14,193279)

Elliptic Curve addition produced a factor of n,
factor= 557
Multell found a factor of n and exited
```

```
ans =  
[]
```

Let's analyze in more detail what happened in these examples.

On the first curve, $266P$ ends up at infinity mod 557 and $35P$ is infinity mod 347. Since $272 = 2 \cdot 7 \cdot 9$, it has a prime factor larger than $B = 12$, so $B!P$ is not infinity mod 557. But 35 divides $B!$, so $B!P$ is infinity mod 347.

On the second curve, $356P = \text{infinity mod } 347$, and $561P = \text{infinity mod } 557$. Since $356 = 4 \cdot 89$ and $561 = 3 \cdot 11 \cdot 17$, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have $331P = \text{infinity mod } 347$ and $272P = \text{infinity mod } 557$. Since 331 is prime and $272 = 16 \cdot 17$, we don't expect to find the factorization with this curve. However, by chance, an intermediate step in the calculation of $B!P$ yielded the factorization. Here's what happened. At an intermediate step in the calculation, the program required adding the points $(184993, 13462)$ and $(20678, 150484)$. These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line through these two points is defined mod 347 but is $0/0 \bmod 557$. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from [Section 21.5](#). For more details, see the text. The elliptic curve is $y^2 \equiv x^3 + 3x + 45 \pmod{8831}$ and the point is

$G = (4, 11)$. Alice's message is the point
 $P_m = (5, 1743)$.

Bob has chosen his secret random number $a_B = 3$ and has computed $a_B G$:

```
>> multell([4,11],3,3,45,8831)

ans =
    413      1808
```

Bob publishes this point. Alice chooses the random number $k = 8$ and computes kG and $P_m + k(a_B G)$:

```
>> multell([4,11],8,3,45,8831)

ans =
    5415      6321

>>
addell([5,1743],multell([413,1808],8,3,45,8831),3
,45,8831)

ans =
    6626      3576
```

Alice sends $(5415, 6321)$ and $(6626, 3576)$ to Bob, who multiplies the first of these points by a_B :

```
>> multell([5415,6321],3,3,45,8831) ans = 673 146
```

Bob then subtracts the result from the last point Alice sends him. Note that he subtracts by adding the point with the second coordinate negated:

```
>> addell([6626,3576],[673,-146],3,45,8831)

ans =
    5 1743
```

Bob has therefore received Alice's message.

Example 48

Let's reproduce the numbers in the example of a Diffie-Hellman key exchange from [Section 21.5](#): The elliptic curve is $y^2 \equiv x^3 + x + 7206 \pmod{7211}$ and the point is $G = (3, 5)$. Alice chooses her secret $N_A = 12$ and Bob chooses his secret $N_B = 23$. Alice calculates

```
>> multell([3,5],12,1,7206,7211)  
  
ans =  
1794 6375
```

She sends (1794,6375) to Bob. Meanwhile, Bob calculates

```
>> multell([3,5],23,1,7206,7211)  
  
ans =  
3861 1242
```

and sends (3861,1242) to Alice. Alice multiplies what she receives by N_A and Bob multiplies what he receives by N_B :

```
>> multell([3861,1242],12,1,7206,7211)  
  
ans =  
1472 2098  
  
>> multell([1794,6375],23,1,7206,7211)  
  
ans =  
1472 2098
```

Therefore, Alice and Bob have produced the same key.

Appendix D Sage Examples

Sage is an open-source computer algebra package. It can be downloaded for free from www.sagemath.org/ or it can be accessed directly online at the website <https://sagecell.sagemath.org/>. The computer computations in this book can be done in Sage, especially by those comfortable with programming in Python. In the following, we give examples of how to do some of the basic computations. Much more is possible. See www.sagemath.org/ or search the Web for other examples. Another good place to start learning Sage in general is [Bard] (there is a free online version).

D.1 Computations for Chapter 2

Shift ciphers

Suppose you want to encrypt the plaintext This is the plaintext with a shift of 3. We first encode it as an alphabetic string of capital letters with the spaces removed. Then we shift each letter by three positions:

```
S=ShiftCryptosystem(AlphabeticStrings())
P=S.encoding("This is the plaintext")
C=S.enciphering(3,P);C
```

When this is evaluated, we obtain the ciphertext

```
WKLVLVWKHSODLQWHAW
```

To decrypt, we can shift by 23 or do the following:

```
S.deciphering(3,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Suppose we don't know the key and we want to decrypt by trying all possible shifts:

```
S.brute_force(C)
```

Evaluation yields

```
0: WKLVLVWKHSODLQWHAW,  
1: VJKUKUVJGRNCKPVGZV,  
2: UIJTJTIQMBJOUFYU,  
3: THISISTHEPLAINTEXT,  
4: SGHRHRSGDOKZHMSDWS,  
5: RFGQGQRFCNJYGLRCVR,  
6: etc.  
  
24: YMNXNXYMJuQFNSYJCY,  
25: XLMWMWXLITPEMRXIBX
```

Affine ciphers

Let's encrypt the plaintext `This is the plaintext` using the affine function $3x + 1 \bmod 26$:

```
A=AffineCryptosystem(AlphabeticStrings())  
P=A.encoding("This is the plaintext")  
C=A.enciphering(3,1,P);C
```

When this is evaluated, we obtain the ciphertext

```
GWZDZDGWNUIBZOGNSG
```

To decrypt, we can do the following:

```
A.deciphering(3,1,C)
```

When this is evaluated, we obtain

THISISTHEPLAINTEXT

We can also find the decryption key:

A.inverse_key(3,1)

This yields

(9, 17)

Of course, if we “encrypt” the ciphertext using $9x + 17$, we obtain the plaintext:

A.enciphering(9,17,C)

Evaluate to obtain

THISISTHEPLAINTEXT

Vigenère ciphers

Let’s encrypt the plaintext `This is the plaintext` using the keyword `ace` (that is, shifts of 0, 2, 4). Since we need to express the keyword as an alphabetic string, it is efficient to add a symbol for these strings:

```
AS=AlphabeticStrings()
V=VigenereCryptosystem(AS,3)
K=AS.encoding("ace")
P=V.encoding("This is the plaintext")
C=V.enciphering(K,P);C
```

The “3” in the expression for V is the length of the key.
When the above is evaluated, we obtain the ciphertext

```
TJMSKWTJIPNEIPXEZX
```

To decrypt, we can shift by 0, 24, 22 (= ayw) or do the following:

```
V.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Now let’s try the example from [Section 2.3](#). The ciphertext can be cut and pasted from ciphertexts.m in the MATLAB files (or, with a little more difficulty, from the Mathematica or Maple files). A few control symbols need to be removed in order to make the ciphertext a single string.

```
vvhq="vhqvwvrhmusgjgthkihtssejchlsfcbgvwcrlyqtfs . . . czvile"
```

(We omitted part of the ciphertext in the above in order to save space.) Now let’s compute the matches for various displacements. This is done by forming a string that displaces the ciphertext by i positions by adding i blank spaces at the beginning and then counting matches.

```
for i in range(0,7):
C2 = [" "]*i + list(C)
count = 0
```

```
for j in range(len(C)):  
    if C2[j] == C[j]:  
        count += 1  
print i, count
```

The result is

```
0 331  
1 14  
2 14  
3 16  
4 14  
5 24  
6 12
```

The 331 is for a displacement of 0, so all 331 characters match. The high number of matches for a displacement of 5 suggests that the key length is 5. We now want to determine the key.

First, let's choose every fifth letter, starting with the first (counted as 0 for Sage). We extract these letters, put them in a list, then count the frequencies.

```
V1=list(C[0::5])  
dict((x, V1.count(x)) for x in V1)
```

The result is

```
C: 7,  
D: 1,  
E: 1,  
F: 2,  
G: 9,  
I: 1,  
J: 8,  
K: 8,  
N: 3,  
P: 4,  
Q: 5,  
R: 2,  
T: 3,
```

```
U: 6,  
V: 5,  
W: 1,  
Y: 1
```

Note that A, B, H, L, M, O, S, X, Z do not occur among the letters, hence are not listed. As discussed in [Subsection 2.3.2](#), the shift for these letters is probably 2. Now, let's choose every fifth letter, starting with the second (counted as 1 for Sage). We compute the frequencies:

```
V2=list(C[1::5])  
dict((x, V2.count(x)) for x in V2)  
  
A: 3,  
B: 3,  
C: 4,  
F: 3,  
G: 10,  
H: 6,  
M: 2,  
O: 3,  
P: 1,  
Q: 2,  
R: 3,  
S: 12,  
T: 3,  
U: 2,  
V: 3,  
W: 3,  
Y: 1,  
Z: 2
```

As in [Subsection 2.3.2](#), the shift is probably 14. Continuing in this way, we find that the most likely key is {2, 14, 3, 4, 18}, which is `codes`. Let's decrypt:

```
V=VigenereCryptosystem(AS,5)  
  
K=AS.encoding("codes")  
P=V.deciphering(K,C);P  
  
THEMETHODUSEDFORTHEPREPARATIONANDREADINGOFCODEMES  
. . . ALSETC
```


D.2 Computations for Chapter 3

To find the greatest common divisor, type the following first line and then evaluate:

```
gcd(119, 259)  
7
```

To find the next prime greater than or equal to a number:

```
next_prime(1000)  
1009
```

To factor an integer:

```
factor(2468)  
2^2 * 617
```

Let's solve the simultaneous congruences

:

```
crt(1,3,5,7)  
31
```

To solve the three simultaneous congruences

:

```
a= crt(1,3,5,7)
```

```
crt(a,0,35,11)  
66
```

Compute :

```
mod(123,789)456  
699
```

Compute so that :

```
mod(65,987)(-1)  
410
```

Let's check the answer:

```
mod(65*410, 987)  
1
```

D.3 Computations for Chapter 5

LFSR

Consider the recurrence relation

$x_{n+4} \equiv x_n + x_{n+1} + x_{n+3} \pmod{2}$, with initial values 1, 0, 0, 0. We need to use 0s and 1s, but we need to tell Sage that they are numbers mod 2. One way is to define “o” (that’s a lower-case “oh”) and “l” (that’s an “ell”) to be 0 and 1 mod 2:

```
F=GF(2)
o=F(0); l=F(1)
```

We also could use F(0) every time we want to enter a 0, but the present method saves some typing. Now we specify the coefficients and initial values of the recurrence relation, along with how many terms we want. In the following, we ask for 20 terms:

```
s=lfsr_sequence([l,l,o,l],[l,o,o,o],20);s
```

This evaluates to

```
[1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
1, 1, 1, 0]
```

Suppose we are given these terms of a sequence and we want to find what recurrence relation generates it:

```
berlekamp_massey(s)
```

This evaluates to

```
x^4 + x^3 + x + 1
```

When this is interpreted as $x^4 \equiv 1 + 1x + 0x^2 + 1x^3 \pmod{2}$, we see that the coefficients 1, 1, 0, 1 of the polynomial give the coefficients of recurrence relation. In fact, it gives the smallest relation that generates the sequence.

Note: Even though the output for `s` has 0s and 1s, if we try entering the command

`berlekamp_massey([1,1,0,1,1,0])` we get $x^3 - 1$. If we instead enter

`berlekamp_massey([1,1,0,1,1,0])`, we get $x^2 + x + 1$. Why? The first is looking at a sequence of integers generated by the relation $x_{n+3} = x_n$ while the second is looking at the sequence of integers mod 2 generated by $x_{n+2} \equiv x_n + x_{n+1} \pmod{2}$. Sage defaults to integers if nothing is specified. But it remembers that the 0s and 1s that it wrote in `s` are still integers mod 2.

D.4 Computations for Chapter 6

Hill ciphers

Let's encrypt the plaintext `This is the plaintext` using a 3×3 matrix. First, we need to specify that we are working with such a matrix with entries in the integers mod 26:

```
R=IntegerModRing(26)
M=MatrixSpace(R,3,3)
```

Now we can specify the matrix that is the encryption key:

```
K=M([[1,2,3],[4,5,6],[7,8,10]]);K
```

Evaluate to obtain

```
[ 1 2 3]
[ 4 5 6]
[ 7 8 10]
```

This is the encryption matrix. We can now encrypt:

```
H=HillCryptosystem(AlphabeticStrings(),3)
P=H.encoding("This is the plaintext")
C=H.enciphering(K,P);C
```

If the length of the plaintext is not a multiple of 3 (= the size of the matrix), then extra characters need to be appended to achieve this. When the above is evaluated, we obtain the ciphertext

```
ZHXUMWXBJJHHHLZGVPC
```

Decrypt:

```
H.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

We could also find the inverse of the encryption matrix mod 26:

```
K1=K.inverse();K1
```

This evaluates to

```
[ 8 16 1]  
[ 8 21 24]  
[ 1 24 1]
```

When we evaluate

```
H.enciphering(K,C):C
```

we obtain

THISISTHEPLAINTEXT

D.5 Computations for Chapter 9

Suppose someone unwisely chooses RSA primes and to be consecutive primes:

```
p=nextprime(987654321*10^50+12345);  
q=nextprime(p+1)  
n=p*q
```

Let's factor the modulus

without using the factor command:

Of course, the fact that π and e are consecutive primes is important for this calculation to work. Note that we needed to specify 70-digit accuracy so that round-off error would not give us the wrong starting point for looking for the next prime. These factors we obtained match the original π and e , up to order:

D.6 Computations for Chapter 10

Let's solve the discrete log problem $2^x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take $N = 12$ since $N^2 > p - 1 = 130$ and we form two lists. The first is $2^j \pmod{131}$ for $0 \leq j \leq 11$:

```
for i in range(0,12): print i, mod(2,131)i

0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 125
9 119
10 107
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \leq j \leq 11$:

```
for i in range(0,12): print i, mod(71*mod(2,131)
(-12*i),131)

0 71
1 17
2 124
3 26
4 128
5 86
6 111
7 93
8 85
9 96
```

10	130
11	116

The number 128 is on both lists, so we see that
 $2^7 \equiv 71 \cdot 2^{-12*4} \pmod{131}$. Therefore,

$$71 \equiv 2^{7+4*12} \equiv 2^{55} \pmod{131}.$$

D.7 Computations for Chapter 12

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
i = var('i') 1-product(1.-i/10^7,i,1,9999)
0.9932699132835016
```

D.8 Computations for Chapter 17

Lagrange interpolation

Suppose we want to find the polynomial of degree at most 3 that passes through the points $(1, 1), (2, 2), (3, 21), (5, 12)$ mod the prime $p = 37$.

We first need to specify that we are working with polynomials in $x \bmod 37$. Then we compute the polynomial:

```
R=PolynomialRing(GF(37),"x")
f=R.lagrange_polynomial([(1,1),(2,2),(3,21),
(5,12)]);f
```

This evaluates to

```
22*x^3 + 25*x^2 + 31*x + 34
```

If we want the constant term:

```
f(0)
43
```

D.9 Computations for Chapter 18

Here is a game you can play. It is essentially the simplified version of poker over the telephone from [Section 18.2](#).

There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 24691313099. You are supposed to guess which one is the ace.

The cards are represented by `ten = 200514`, etc., because `t` is the 20th letter, `e` is the 5th letter, and `n` is the 14th letter.

Type the following into Sage. The value of k forces the randomization to have 248 as its starting point, so the random choices of e and `shuffle` do not change when we repeat the process with this value of k . Varying k gives different results.

```
cards=[200514,10010311,1721050514,11091407,10305]
p=24691313099
k=248 set_random_seed(k)
e=randint(10,10^7)
def pow(list,ex):
    ret = []
    for i in list:
        ret.append(mod(i,p)^ex)
    return ret
s=pow(cards,2*e+1)
shuffle(s)
print(s)
```

Evaluation yields

```
[10426004161, 16230228497, 12470430058,
3576502017, 2676896936]
```

These are the five cards. None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

Add the following line to the end of the program:

```
print(pow(s,mod(2*e+1,p-1)^(-1)))
```

and evaluate again:

```
[10426004161, 16230228497, 12470430058,
3576502017, 2676896936]
[10010311, 200514, 11091407, 10305, 1721050514]
```

The first line is the shuffled and hidden cards. The second line removed the k th power and reveals the cards. The fourth card is the ace. Were you lucky?

If you change the value of k , you can play again, but let's figure out how to cheat. Remove the last line of the program that you just added and replace it with

```
pow(s,(p-1)/2)
```

When the program is evaluated, we obtain

```
[10426004161, 16230228497, 12470430058,
3576502017, 2676896936]
[1, 1, 1, 24691313098, 1]
```

Why does this tell us the ace is in the fourth position?

Read the part on “How to Cheat” in [Section 18.2](#). Raise the numbers for the cards to the $(p - 1)/2$ power mod p (you can put this extra line at the end of the program and ignore the previous output):

```
print(pow(cards,(p-1)/2))
[1, 1, 1, 1, 24691313098]
```

We see that the prime p was chosen so that only the ace is a quadratic nonresidue mod p .

If you input another value of k and play the game again, you’ll have a similar situation, but with the cards in a different order.

D.10 Computations for Chapter 21

Elliptic curves

Let's set up the elliptic curve $y^2 \equiv x^3 + 2x + 3 \pmod{7}$:

```
E=EllipticCurve(IntegerModRing(7),[2,3])
```

The entry `[2, 3]` gives the coefficients a, b of the polynomial $x^3 + ax + b$. More generally, we could use the vector `[a, b, c, d, e]` to specify the coefficients of the general form $y^2 + axy + cy \equiv x^3 + bx^2 + dx + e$. We could also use `GF(7)` instead of `IntegerModRing(7)` to specify that we are working mod 7. We could replace the 7 in `IntegerModRing(7)` with a composite modulus, but this will sometimes result in error messages when adding points (this is the basis of the elliptic curve factorization method).

We can list the points on E :

```
E.points()
[(0:1:0), (2:1:1), (2:6:1), (3:1:1), (3:6:1),
(6:0:1)]
```

These are given in projective form. The point $(0:1:0)$ is the point at infinity. The point $(2:6:1)$ can also be written as $[2, 6]$. We can add points:

```
E([2,1])+E([3,6])
```

```
(6 : 0 : 1)  
E([0,1,0])+E([2,6])  
(2 : 6 : 1)
```

In the second addition, we are adding the point at infinity to the point $(2, 6)$ and obtaining $(2, 6)$. This is an example of $\infty + P = P$. We can multiply a point by an integer:

```
5*E([2,1])  
(2 : 6 : 1)
```

We can list the multiples of a point in a range:

```
for i in range(10):  
    print(i,i*E([2,6]))  
  
(0, (0 : 1 : 0))  
(1, (2 : 6 : 1))  
(2, (3 : 1 : 1))  
(3, (6 : 0 : 1))  
(4, (3 : 6 : 1))  
(5, (2 : 1 : 1))  
(6, (0 : 1 : 0))  
(7, (2 : 6 : 1))  
(8, (3 : 1 : 1))  
(9, (6 : 0 : 1))
```

The indentation of the `print` line is necessary since it indicates that this is iterated by the `for` command. To count the number of points on E :

```
E.cardinality()  
6
```

Sage has a very fast point counting algorithm (due to Atkins, Elkies, and Schoof; it is much more sophisticated than listing the points, which would be infeasible). For example,

```
p=next_prime(10^50)
E1=EllipticCurve(IntegerModRing(p),[2,3])
n=E1.cardinality()
p, n, n-(p+1)
(100000000000000000000000000000000000000000000000000000000000000
151,
99999999999999999999999999999911231473313376108623203
2,
-887685266866238913768120)
```

As you can see, the number of points on this curve (the second output line) is close to $p + 1$. In fact, as predicted by Hasse's theorem, the difference $n - (p + 1)$ (on the last output line) is less in absolute value than

$$2\sqrt{p} \approx 2 \times 10^{25}.$$

Appendix E Answers and Hints for Selected Odd-Numbered Exercises

Chapter 2

1. 1. Among the shifts of *EVIRE*, there are two words: *arena* and *river*. Therefore, Anthony cannot determine where to meet Caesar.
2. 3. The decrypted message is 'cat'.
3. 5. The ciphertext is *QZNHOBXZD*. The decryption function is $21x + 9$.
4. 7. The encryption function is therefore $11x + 14$.
5. 9. The plaintext is *happy*.
6. 11. Successively encrypting with two affine functions is the same as encrypting with a single affine function. There is therefore no advantage of doing double encryption in this case.
7. 13. Mod 27, there are 486 keys. Mod 29, there are 812 keys.
8. 15.
 1. The possible values for α are 1,7,11,13,17,19,23,29.
 2. There are many such possible answers, for example $x = 1$ and $x = 4$ will work. These correspond to the letters 'b' and 'e'.
9. 19.
 2. The key is *AB*. The original plaintext is *BBBBBBBABBB*.
10. 21. The key is probably *AB*.
11. 27. *EK IO IR NO AN HF YG BZ YB LF GM ZN AG ND OD VC MK*
12. 29. *AAXFFGDGAFAX*

Chapter 3

1. 1.

1. One possibility: $1 = (-1) \cdot 101 + 6 \cdot 17$.

2. 6

2. 3.

1. $d = 13$

2. Decryption is performed by raising the ciphertext to the 13th power mod 31.

3. 5.

1. $x \equiv 22 \pmod{59}$, or
 $x \equiv 22, 81, 140, 199 \pmod{236}$.

2. No solutions.

4. 7.

1. If $n = ab$ with $1 < a, b < n$, then either $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$. Otherwise, $ab > (\sqrt{n})(\sqrt{n}) = n$. If $1 < a < \sqrt{n}$, let p be a prime factor of a .

2. $\gcd(30030, 257) = 1$.

3. No prime less than or equal to $\sqrt{257} \approx 16.03$ divides 257 because of the gcd calculation.

5. 9.

1. The gcd is 257.
2. $4883 = 257 \cdot 19$ and $4369 = 257 \cdot 17$.

6. 11.

1. The gcd is 1.
2. The gcd is 1.
3. The gcd is 1.

7. 13.

1. Use the Corollary in Section 3.2.

2. Imitate the proof of the Corollary in [Section 3.2](#).

8. [17.](#) $x \equiv 23 \pmod{70}$.

9. [19.](#) The smallest number is 58 and the next smallest number is 118.

10. [21.](#)

1. $x \equiv 43, 56, 87, 100 \pmod{143}$.

2. $x \equiv 44, 99 \pmod{143}$.

11. [23.](#) The remainder is 8.

12. [25.](#) The last two digits are 29.

13. [27.](#) Use Fermat's theorem when $a \not\equiv 0 \pmod{p}$.

14. [29.](#)

1. $7^7 \equiv 3 \pmod{4}$.

2. The last digit is 3.

15. [39.](#)

$$1. \begin{pmatrix} 5 & 21 \\ 22 & 5 \end{pmatrix}.$$

2.

$b \equiv 0, 2, 4, 6, 8, 10, 12, 16, 18, 20, 22, 24 \pmod{26}$

.

16. [41.](#) 2 and 13

17. [43.](#)

1. No solutions.

2. There are solutions.

3. No solutions.

Chapter 4

1. 1.

1. O.

2. $P(M = \text{cat} \mid C = \text{mfp}) \neq P(M = \text{cat}).$

2. 3. The conditional probability is 0. Affine ciphers do not have perfect secrecy.

3. 5.

1. $1/2.$

2. $m_0 = HI, m_1 = BE$ is a possibility.

4. 11.

1. Possible.

2. Possible.

3. Impossible.

5. 13. X

Chapter 5

1. 1.] The next four terms of the sequence are 1, 0, 0, 1.

2. 3. $c_0 = 1, c_1 = 0, c_2 = 0, c_3 = 1$.

3. 5. $c_0 = 2$ and $c_1 = 1$.

4. 7. $k_{n+2} \equiv k_n$.

5. 9. $c_0 = 4$ and $c_1 = 4$.

Chapter 6

1. 1. eureka.

$$2. 3. \begin{pmatrix} 12 & 3 \\ 11 & 2 \end{pmatrix}.$$

$$3. 5. \begin{pmatrix} 7 & 2 \\ 13 & 5 \end{pmatrix}.$$

4. 7.

$$1. \begin{pmatrix} 10 & 9 \\ 13 & 23 \end{pmatrix}.$$

$$2. \begin{pmatrix} 10 & 19 \\ 13 & 19 \end{pmatrix}.$$

5. 9. Use *aabaab*.

6. 13.

1. Alice's method is more secure.

2. Compatibility with single encryption.

7. 19. The j th and the $(j + 1)$ st blocks.

Chapter 7

1. 1.

1. Switch left and right halves and use the same procedure as encryption. Then switch the left and right of the final output.
2. After two rounds, the ciphertext alone lets you determine M_0 and therefore $M_1 \oplus K$, but not M_1 or K individually. If you also know the plaintext, you know M_1 are therefore can deduce K .
3. Three rounds is very insecure.

2. 3. The ciphertext from the second message can be decrypted to yield the password.

3. 5.

1. The keys for each round are all 1s, so using them in reverse order doesn't change anything.

2. All os.

4. 7. Show that when P and K are used, the input to the S-boxes is the same as when P and K are used.

Chapter 8

1. 1.

1. We have $W(4) = W(0) \oplus T(W(0)) = T(W(0))$. In the notation in Subsection 8.2.5, $a = b = c = d = 0$.

The S -box yields

$e = f = g = h = 99$ (base 10) = 01100011 (binary)

. The round constant is

$r(4) = 00000010^0 = 00000001$. We have

$e \oplus r(4) = 01100100$. Therefore,

$$W(4) = T(W(0)) = \begin{array}{c} 01100100 \\ 01100011 \\ 01100011 \\ 01100011 \end{array} .$$

2. 3.

1. Since addition in $GF(2^8)$ is the same as \oplus , we have

$$f(x_1) \oplus f(x_2) = \alpha(x_1 + x_2) = \alpha(x_3 + x_4) = f(x_3) \oplus f(x_4)$$

.

Chapter 9

1. 1. The plaintext is $1415 = no$.

2. 3.

1. $d = 27$.

2. Imitate the proof that RSA decryption works.

3. 5. The correct plaintext is 9 .

4. 7. Use $d = 67$.

5. 9. Solve $de \equiv 1 \pmod{p-1}$.

6. 11. Bob computes b_1 with $bb_1 \equiv 1 \pmod{\phi(n)}$ and raises e to the power b_1 .

7. 13. Divide the decryption by 2.

8. 15. It does not increase security.

9. 17. $e = 1$ sends plaintext, and $e = 2$ doesn't satisfy $\gcd(e, (p-1)(q-1)) = 1$.

10. 21. Compute a gcd.

11. 23. We have $(516107 \cdot 187722)^2 \equiv (2 \cdot 7)^2 \pmod{n}$.

12. 25. Combine the first three congruences. Ignore the fourth congruence.

13. 27. Use the Chinese Remainder Theorem to find x with $x \equiv 7 \pmod{p}$ and $x \equiv -7 \pmod{q}$.

14. 31. There are integers x and y such that $xe_A + ye_B = 1$.

15. 33. *HELLO*

16. 41. 12345.

17. 45. Find d' with $d'e \equiv 1 \pmod{12345}$.

18. 47.

2. 1000000 messages.

Chapter 10

1. 1.

1. $L_2(3) = 4$.

2. $2^7 \equiv 11 \pmod{13}$.

2. 3.

1. $6^5 \equiv 10$.

2. x is odd.

3. 5. x is even.

4. 7. (a), (b) $L_2(24) = 72$.

5. 9. $x = 122$.

6. 13. Alice sends 10 to Bob and Bob sends 5 to Alice. Their shared secret is 6.

7. 15. Eve computes b_1 with $bb_1 \equiv 1 \pmod{p-1}$.

Chapter 11

1. 1. It is easy to construct collisions: $h(x) = h(x + p - 1)$, for example. (However, it is fairly quickly computed (though not fast enough for real use), and it is preimage resistant.)

2. 3.

1. Finding square roots mod pq is computationally equivalent to factoring.

2. $h(x) \equiv h(n - x)$ for all x

3. 5. It satisfies (1), but not (2) and (3).

4. 9. (a) and (b) Let h be either of the hash functions. Given y of length n , we have $h(y \parallel 000 \dots) = y$.

5. 11. Collision resistance.

Chapter 12

1. $\frac{165}{288} \approx .573$

2. 5. $1 - e^{-7.3 \times 10^{-47}} \approx 0.$

3. 7. It is very likely that two choose the same prime.

4. 9. The probability is approximately $1 - e^{-500} \approx 0$ (or $1 - e^{-450} \approx 0$ if you take numbers of exactly 15 digits).

Chapter 13

1. 1. Use the congruence defining s to solve for a .
2. 3. See Section 13.4.
3. 7.
 1. Let $m_1 \equiv mr_1r^{-1} \pmod{p-1}$.
4. 9. Imitate the proof for the usual ElGamal signatures.
5. 13. Eve notices that $\beta = r$.

Chapter 14

1. 1. Use the Birthday attack. Eve will probably factor some moduli.

2. 3.

1. 0101 and 0110.

3. 5.

1. Enigma does not encrypt a letter to itself, so DOG is impossible.
2. If the first of two long plaintexts is encrypted with Enigma, it is very likely that at least one letter of the second plaintext will match a letter of the ciphertext. More precisely, each individual letter of the second plaintext that doesn't match the first plaintext has probability around $1/26$ of matching, so the probability is $1 - (25/26)^{90} \approx 0.97$ that there is a match between the second plaintext and the ciphertext. Therefore, Enigma does not have ciphertext indistinguishability.

Chapter 15

1. 1. $K_{AB} = g_A(r_B) = 21$, $K_{AC} = g_A(r_C) = 7$ and
 $K_{BC} = g_B(r_C) = 29$.

2. 3. $a = 8, b = 8, c = 23$.

Chapter 16

1. 1.

1. We have

$$r \equiv \alpha_1(cx + w) + \alpha_2 \equiv Hx + \alpha_1w + \alpha_2 \pmod{q}.$$

Therefore

$$g^r \equiv g^{w\alpha_1}g^{\alpha_2}g^{xH} \equiv g_w^{\alpha_1}g^{\alpha_2}g^{xH} \equiv ah^H \pmod{p}.$$

2. Since $c_1 \equiv w + xc \pmod{q}$, we have

$$\alpha_1c_1 \equiv w\alpha_1 + xH \pmod{q}. \text{ Therefore,}$$

$$r \equiv \alpha_1c_1 + \alpha_2 \equiv xH + w\alpha_1 + \alpha_2 \pmod{q}.$$

Multiply by s and raise Ig_2 to these exponents to obtain

$$(Ig_2)^r s \equiv (Ig_2)^{xsH}(Ig_2)^{ws\alpha_1}(Ig_2)^{s\alpha_2} \pmod{p}.$$

This may be rewritten as

$$A^r \equiv z^H b \pmod{p}.$$

3. Since $r_1 \equiv usd + x_1$ and $r_2 \equiv sd + x_2 \pmod{q}$, we have

$$g_1^{r_1}g_2^{r_2} \equiv (g_1^u g_2)^{sd} g_1^{x_1}g_2^{x_2} \equiv (Ig_2^{sd}g_1^{x_1}g_2^{x_2}) \equiv A^d B \pmod{p}.$$

2. 3.

1. The only place r_1 and r_2 are used in the verification procedure is in checking that $g_1^{r_1}g_2^{r_2} \equiv A^d B$.

2. The Spender spends the coin correctly once, using r_1, r_2 . The Spender then chooses any two random numbers r'_1, r'_2 with $r'_1 + r'_2 = r_1 + r_2$ and uses the coin with the Vendor, with r'_1, r'_2 in place of r_1, r_2 . All the verification equations work.

3. 5. r_2 and r'_2 are essentially random numbers (depending on hash values involving the clock), the probability is around $1/q$ that $r_2 \equiv r'_2 \pmod{q}$. Since q is large, $1/q$ is small.

4. 7. Fred only needs to keep the hash of the file on his own computer.

Chapter 17

1. One possibility is to take $p = 7$ and choose the polynomial $s(x) = 5 + x \pmod{7}$ (where 5 is chosen randomly). Then the secret value is $s(0) = 5$, and we may choose the shares (1,6), (2,0), (3,1), and (4,2).

2. $3^* = 63$

3. The polynomial is

$$8 \frac{(x-3)(x-5)}{(1-3)(1-5)} + 10 \frac{(x-1)(x-5)}{(3-1)(3-5)} + 11 \frac{(x-1)(x-3)}{(5-1)(5-3)}.$$

The secret is $13 \pmod{17}$.

4. $M = 77, 57, 37, 17$.

5. Take a (10, 30) scheme and give the general 10 shares, the colonels five shares each, and the clerks two each.

6. Start by splitting the launch code into three equal components using a three-party secret splitting scheme.

Chapter 19

1. 3.

1. Nelson computes a square root of $y \bmod p$ and $\bmod q$, then combines them to obtain a square root of $y \bmod n$.
2. Use the $x^2 \equiv y^2 \pmod{n}$ factorization method.
3. No.

2. 5.

Step 4: Victor randomly chooses $i = 1$ or 2 and asks Peggy for r_i .

Step 5: Victor checks that $x_i \equiv r_i^e \pmod{n}$.

They repeat steps 1 through 5 at least 7 times (since $(1/2)^7 < .01$).

3. 7.

1. One way: Step 4: Victor chooses $i \neq j$ at random and asks for r_i and r_j . Then five repetitions are enough.
Another way: Victor asks for only one of the r_k 's. Then twelve repetitions suffice.
2. Choose r_1, r_2 . Then solve for r_3 .

Chapter 20

1. 1. $H(X_1) = H(X_2) = 1$, and $H(X_1, X_2) = 2$.

2. 3. $H(X) = 2$.

3. 5. $H(Y) \leq H(X)$.

4. 7.

1. 2.9710

2. 2.9517

5. 9.

$$1. H(P) = -\left(\frac{1}{3} \log_2 \frac{1}{3} + \frac{2}{3} \log_2 \frac{2}{3}\right).$$

2. This system matches up with the one-time pad, and hence $H(P|C) = H(P)$.

6. 13.

1. $H(X) = \log_2 36$.

2. Use Fermat's Theorem to obtain $H(Y) = 0$.

Chapter 21

1. 3.

1. $(3, 2), (3, 5), (5, 2), (5, 5), (6, 2), (6, 5), \infty$.

2. $(3, 5)$.

3. $(5, 2)$.

2. 5.

1. $(2, 2)$.

2. She factors 35.

3. 7. One example: $y^2 \equiv x^3 + 17$.

4. 9.

1. $3Q = (-1, 0)$.

5. 15. $y \equiv \pm 4, \pm 11 \pmod{35}$

Chapter 22

1. 3. Compute $\tilde{e}(aA, B)$ and $\tilde{e}(A, bB)$.

2. 7.

1. Eve knows rP_0, P_1, k . She computes

$$\tilde{e}(rP_0, P_1)^k = \tilde{e}(kP_0, P_1)^r = \tilde{e}(H_1(\text{bob@computer.com}), P_1)^r = g^r.$$

Eve now computes $H_2(g^r)$ and XORs it with t to get m .

3. 9. See Claim 2 in Section 9.1.

Chapter 23

1. 1. The basis $(0, 1), (-2, 0)$ is reduced. The vector $(0, 1)$ is a shortest vector.

Chapter 24

1. 1.

1. The original message is 0,1,0,0.

2. The original message is 0,1,0,1.

2. 3.

1. $n = 5$ and $k = 2$.

2. $G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$

3.

(0, 0, 0, 0, 0), (1, 1, 0, 1, 0), (1, 0, 1, 0, 1), (0, 1, 1, 1, 1)

.

4. $R = \frac{\log_2 (4)}{5} = 0.4.$

3. 5.

2. $d(C) = 2.$

4. 13. $1 + X + X^2 + X^3$ is in C and the other two polynomials are not in C .

5. 19. The error is in the 3rd position. The corrected vector is (1,0,0,1,0,1,1).

Chapter 25

1. 1.

1. The period is 4.

2. $m = 8$.

3. $r = 4$.

Appendix F Suggestions for Further Reading

For the history of cryptography, see [Kahn] and [Bauer].

For additional treatment of topics in the present book, and many other topics, see [Stinson], [Stinson1], [Schneier], [Mao], and [Menezes et al.]. These books also have extensive bibliographies.

An approach emphasizing algebraic methods is given in [Koblitz].

For the theoretical foundations of cryptology, see [Goldreich1] and [Goldreich2]. See [Katz-Lindell] for an approach based on security proofs.

Books that are oriented toward protocols and practical network security include [Stallings], [Kaufman et al.], and [Aumasson].

For guidelines on properly applying cryptographic algorithms, the reader is directed to [Ferguson-Schneier]. For a general discussion on securing computing platforms, see [Pfleeger-Pfleeger].

The Internet, of course, contains a wealth of information about cryptographic issues. The Cryptology ePrint Archive server at <http://eprint.iacr.org/> contains very recent research. Also, the conference proceedings CRYPTO, EUROCRYPT, and ASIACRYPT (published in Springer-Verlag's Lecture Notes in Computer Science series) contain many interesting reports on recent developments.

Bibliography

- [Adrian et al.] Adrian et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," <https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf>.
- [Agrawal et al.] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Annals of Math.* 160 (2004), 781–793.
- [Alford et al.] W. R. Alford, A. Granville, and C. Pomerance, "On the difficulty of finding reliable witnesses," *Algorithmic Number Theory, Lecture Notes in Computer Science* 877, Springer-Verlag, 1994, pp. 1–16.
- [Alford et al. 2] W. R. Alford, A. Granville, and C. Pomerance, "There are infinitely many Carmichael numbers," *Annals of Math.* 139 (1994), 703–722.
- [Atkins et al.] D. Atkins, M. Graff, A. Lenstra, P. Leyland, "The magic words are squeamish ossifrage," *Advances in Cryptology – ASIACRYPT '94, Lecture Notes in Computer Science* 917, Springer-Verlag, 1995, pp. 263–277.
- [Aumasson] J-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press, 2017.
- [Bard] G. Bard, *Sage for Undergraduates*, Amer. Math. Soc., 2015.
- [Bauer] C.Bauer, *Secret History: The Story of Cryptology*, CRC Press, 2013.
- [Beker-Piper] H. Beker and F. Piper, *Cipher Systems: The Protection of Communications*, Wiley-Interscience, 1982.
- [Bellare et al.] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology (Crypto 96 Proceedings)*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.
- [Bellare-Rogaway] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," *First ACM Conference on Computer and Communications Security*, ACM Press, New York, 1993, pp. 62–73.
- [Bellare-Rogaway2] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," *Advances in Cryptology – EUROCRYPT '94, Lecture Notes in Computer Science* 950, Springer-Verlag, 1995, pp. 92–111.

- [Berlekamp] E. Berlekamp, Algebraic Coding Theory, McGraw-Hill, 1968.
- [Bernstein et al.] Post-Quantum Cryptography, Bernstein, Daniel J., Buchmann, Johannes, Dahmen, Erik (Eds.), Springer-Verlag, 2009.
- [Bitcoin] bitcoin, <https://bitcoin.org/en/>
- [Blake et al.] I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, 1999.
- [Blom] R. Blom, “An optimal class of symmetric key generation schemes,” Advances in Cryptology – EUROCRYPT’84, Lecture Notes in Computer Science 209, Springer-Verlag, 1985, pp. 335–338.
- [Blum-Blum-Shub] L. Blum, M. Blum, and M. Shub, “A simple unpredictable pseudo-random number generator,” SIAM Journal of Computing 15(2) (1986), 364–383.
- [Boneh] D. Boneh, “Twenty years of attacks on the RSA cryptosystem,” Amer. Math. Soc. Notices 46 (1999), 203–213.
- [Boneh et al.] D. Boneh, G. Durfee, and Y. Frankel, “An attack on RSA given a fraction of the private key bits,” Advances in Cryptology – ASIACRYPT ’98, Lecture Notes in Computer Science 1514, Springer-Verlag, 1998, pp. 25–34.
- [Boneh-Franklin] D. Boneh and M. Franklin, “Identity based encryption from the Weil pairing,” Advances in Cryptology – CRYPTO ’01, Lecture Notes in Computer Science 2139, Springer-Verlag, 2001, pp. 213–229.
- [Boneh-Joux-Nguyen] D. Boneh, A. Joux, P. Nguyen, “Why textbook ElGamal and RSA encryption are insecure,” Advances in Cryptology – ASIACRYPT ’00, Lecture Notes in Computer Science 1976, Springer-Verlag, 2000, pp. 30–43.
- [Brands] S. Brands, “Untraceable off-line cash in wallets with observers,” Advances in Cryptology – CRYPTO’93, Lecture Notes in Computer Science 773, Springer-Verlag, 1994, pp. 302–318.
- [Campbell-Wiener] K. Campbell and M. Wiener, “DES is not a group,” Advances in Cryptology – CRYPTO ’92, Lecture Notes in Computer Science 740, Springer-Verlag, 1993, pp. 512–520.
- [Canetti et al.] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle methodology, revisited,” Proceedings of the thirtieth annual ACM symposium on theory of computing, ACM Press, 1998, pp. 209–218.
- [Chabaud] F. Chabaud, “On the security of some cryptosystems based on error-correcting codes,” Advances in Cryptology –

EUROCRYPT'94, Lecture Notes in Computer Science 950,
Springer-Verlag, 1995, pp. 131–139.

- [Chaum et al.] D. Chaum, E. van Heijst, and B. Pfitzmann, “Cryptographically strong undeniable signatures, unconditionally secure for the signer,” Advances in Cryptology – CRYPTO ’91, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 470–484.
- [Cohen] H. Cohen, A Course in Computational Number Theory, Springer-Verlag, 1993.
- [Coppersmith1] D. Coppersmith, “The Data Encryption Standard (DES) and its strength against attacks,” IBM Journal of Research and Development, vol. 38, no. 3, May 1994, pp. 243–250.
- [Coppersmith2] D. Coppersmith, “Small solutions to polynomial equations, and low exponent RSA vulnerabilities,” J. Cryptology 10 (1997), 233–260.
- [Cover-Thomas] T. Cover and J. Thomas, Elements of Information Theory, Wiley Series in Telecommunications, 1991.
- [Crandall-Pomerance] R. Crandall and C. Pomerance, Prime Numbers, a Computational Perspective, Springer-Telos, 2000.
- [Crosby et al.] Crosby, S. A., Wallach, D. S., and Riedi, R. H. “Opportunities and limits of remote timing attacks,” ACM Trans. Inf. Syst. Secur. 12, 3, Article 17 (January 2009), 29 pages.
- [Damgård et al.] I. Damgård, P. Landrock, and C. Pomerance, “Average case error estimates for the strong probable prime test,” Mathematics of Computation 61 (1993), 177–194.
- [Dawson-Nielsen] E. Dawson and L. Nielsen, “Automated Cryptanalysis of XOR Plaintext Strings,” Cryptologia 20 (1996), 165–181.
- [Diffie-Hellman] W. Diffie and M. Hellman, “New directions in cryptography,” IEEE Trans. in Information Theory, 22 (1976), 644–654.
- [Diffie-Hellman2] W. Diffie and M. Hellman, “Exhaustive cryptanalysis of the NBS data encryption standard,” Computer 10(6) (June 1977), 74–84
- [Ekert-Josza] A. Ekert and R. Jozsa, “Quantum computation and Shor’s factoring algorithm,” Reviews of Modern Physics, 68 (1996), 733–753.
- [FIPS 186-2] FIPS 186-2, Digital signature standard (DSS), Federal Information Processing Standards Publication 186, U.S. Dept. of Commerce/National Institute of Standards and Technology, 2000.

- [FIPS 202] FIPS PUB 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Federal Information Processing Standards Publication 202, U.S. Dept. of Commerce/National Institute of Standards and Technology, 2015, available at <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.
- [Ferguson-Schneier] N. Ferguson and B. Schneier, Practical Cryptography, Wiley, 2003.
- [Fortune-Merritt] S. Fortune and M. Merritt, “Poker Protocols,” Advances in Cryptology – CRYPTO’84, Lecture Notes in Computer Science 196, Springer-Verlag, 1985, pp. 454–464.
- [Gaines] H. Gaines, Cryptanalysis, Dover Publications, 1956.
- [Gallager] R. G. Gallager, Information Theory and Reliable Communication, Wiley, 1969.
- [Genkin et al.] D. Genkin, A. Shamir, and E. Tromer, “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis,” December 18, 2013, available at www.cs.tau.ac.il/~tromer/papers/acoustic-20131218.pdf
- [Gilmore] Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design, Electronic Frontier Foundation, J. Gilmore (editor), O’Reilly and Associates, 1998.
- [Girault et al.] M. Girault, R. Cohen, and M. Campana, “A generalized birthday attack,” Advances in Cryptology – EUROCRYPT’88, Lecture Notes in Computer Science 330, Springer-Verlag, 1988, pp. 129–156.
- [Goldreich1] O. Goldreich, Foundations of Cryptography: Volume 1, Basic Tools, Cambridge University Press, 2001.
- [Goldreich2] O. Goldreich, Foundations of Cryptography: Volume 2, Basic Applications, Cambridge University Press, 2004.
- [Golomb] S. Golomb, Shift Register Sequences, 2nd ed., Aegean Park Press, 1982.
- [Hankerson et al.] D. Hankerson, A. Menezes, and S. Vanstone, Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.
- [Hardy-Wright] G. Hardy and E. Wright, An Introduction to the Theory of Numbers. Fifth edition, Oxford University Press, 1979.
- [Heninger et al.] N. Heninger, Z. Durumeric, E. Wustrow, J. A. Halderman, “Mining your and : Detection of widespread weak key in network devices,” Proc. 21st USENIX Security Symposium, Aug. 2012; available at <https://factorable.net>.
- [HIP] R. Moskowitz and P. Nikander, “Host Identity Protocol (HIP) Architecture,” May 2006; available at <https://>

tools.ietf.org/html/rfc4423

- [Joux] A. Joux, “Multicollisions in iterated hash functions. Application to cascaded constructions,” Advances in Cryptology – CRYPTO 2004, Lecture Notes in Computer Science 3152, Springer, 2004, pp. 306–316.
- [Kahn] D. Kahn, The Codebreakers, 2nd ed., Scribner, 1996.
- [Kaufman et al.] C. Kaufman, R. Perlman, M. Speciner, Private Communication in a Public World. Second edition, Prentice Hall PTR, 2002.
- [Kilian-Rogaway] J. Kilian and P. Rogaway, “How to protect DES against exhaustive key search (an analysis of DESX),” J. Cryptology 14 (2001), 17–35.
- [Koblitz] N. Koblitz, Algebraic Aspects of Cryptography, Springer-Verlag, 1998.
- [Kocher] P. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” Advances in Cryptology – CRYPTO ’96, Lecture Notes in Computer Science 1109, Springer, 1996, pp. 104–113.
- [Kocher et al.] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” Advances in Cryptology – CRYPTO ’99, Lecture Notes in Computer Science 1666, Springer, 1999, pp. 388–397.
- [Konikoff-Toplosky] J. Konikoff and S. Toplosky, “Analysis of Simplified DES Algorithms,” Cryptologia 34 (2010), 211–224.
- [Kozaczuk] W. Kozaczuk, Enigma: How the German Machine Cipher Was Broken, and How It Was Read by the Allies in World War Two; edited and translated by Christopher Kasparek, Arms and Armour Press, London, 1984.
- [KraftW] J. Kraft and L. Washington, An Introduction to Number Theory with Cryptography, CRC Press, 2018.
- [Lenstra et al.] A. Lenstra, X. Wang, B. de Weger, “Colliding X.509 certificates,” preprint, 2005.
- [Lenstra2012 et al.] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, “Ron was wrong, Whit is right,” <https://eprint.iacr.org/2012/064.pdf>.
- [Lin-Costello] S. Lin and D. J. Costello, Jr., Error Control Coding: Fundamentals and Applications, Prentice Hall, 1983.
- [MacWilliams-Sloane] F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Codes, North-Holland, 1977.
- [Mantin-Shamir] I. Mantin and A. Shamir, “A practical attack on broadcast RC4,” In: FSE 2001, 2001.

- [Mao] W. Mao, *Modern Cryptography: Theory and Practice*, Prentice Hall PTR, 2004.
- [Matsui] M. Matsui, “Linear cryptanalysis method for DES cipher,” *Advances in Cryptology – EUROCRYPT’93*, Lecture Notes in Computer Science 765, Springer-Verlag, 1994, pp. 386–397.
- [Menezes et al.] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
- [Merkle-Hellman] R. Merkle and M. Hellman, “On the security of multiple encryption,” *Comm. of the ACM* 24 (1981), 465–467.
- [Mikle] O. Mikle, “Practical Attacks on Digital Signatures Using MD5 Message Digest,” *Cryptology ePrint Archive*, Report 2004/356, <http://eprint.iacr.org/2004/356>, 2nd December 2004.
- [Nakamoto] S. Nakamoto, ”Bitcoin: A Peer-to-peer Electronic Cash System,” available at <https://bitcoin.org/bitcoin.pdf>
- [Narayanan et al.] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction* (with a preface by Jeremy Clark), Princeton University Press 2016.
- [Nelson-Gailly] M. Nelson and J.-L. Gailly, *The Data Compression Book*, M&T Books, 1996.
- [Nguyen-Stern] P. Nguyen and J. Stern, “The two faces of lattices in cryptology,” *Cryptography and Lattices*, International Conference, CaLC 2001, Lecture Notes in Computer Science 2146, Springer-Verlag, 2001, pp. 146–180.
- [Niven et al.] I. Niven, H. Zuckerman, and H. Montgomery, *An Introduction to the Theory of Numbers*, Fifth ed., John Wiley & Sons, Inc., New York, 1991.
- [Okamoto-Ohta] T. Okamoto and K. Ohta, “Universal electronic cash,” *Advances in Cryptology – CRYPTO’91*, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 324–337.
- [Pfleeger-Pfleeger] C. Pfleeger, S. Pfleeger, *Security in Computing*. Third edition, Prentice Hall PTR, 2002.
- [Pomerance] C. Pomerance, “A tale of two sieves,” *Notices Amer. Math. Soc.* 43 (1996), no. 12, 1473–1485.
- [Quisquater et al.] J.-J. Quisquater and L. Guillou, “How to explain zero-knowledge protocols to your children,” *Advances in Cryptology – CRYPTO ’89*, Lecture Notes in Computer Science 435, Springer-Verlag, 1990, pp. 628–631.
- [Rieffel-Polak] E. Rieffel and W. Polak, “An Introduction to Quantum Computing for Non-Physicists,” available at

xxx.lanl.gov/abs/quant-ph/9809016.

- [Rosen] K. Rosen, Elementary Number Theory and its Applications. Fourth edition, Addison-Wesley, Reading, MA, 2000.
- [Schneier] B. Schneier, Applied Cryptography, 2nd ed., John Wiley, 1996.
- [Shannon1] C. Shannon, “Communication theory of secrecy systems,” Bell Systems Technical Journal 28 (1949), 656–715.
- [Shannon2] C. Shannon, “A mathematical theory of communication,” Bell Systems Technical Journal, 27 (1948), 379–423, 623–656.
- [Shoup] V. Shoup, “OAEP Reconsidered,” CRYPTO 2001 (J. Kilian (ed.)), Springer LNCS 2139, Springer-Verlag Berlin Heidelberg, 2001, pp. 239–259.
- [Stallings] W. Stallings, Cryptography and Network Security: Principles and Practice, 3rd ed., Prentice Hall, 2002.
- [Stevens et al.] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, “The first collision for full SHA-1,” <https://shattered.io/static/shattered.pdf>.
- [Stinson] D. Stinson, Cryptography: Theory and Practice. Second edition, Chapman & Hall/CRC Press, 2002.
- [Stinson1] D. Stinson, Cryptography: Theory and Practice, CRC Press, 1995.
- [Thompson] T. Thompson, From Error-Correcting Codes through Sphere Packings to Simple Groups, Carus Mathematical Monographs, number 21, Mathematical Assoc. of America, 1983.
- [van der Lubbe] J. van der Lubbe, Basic Methods of Cryptography, Cambridge University Press, 1998.
- [van Oorschot-Wiener] P. van Oorschot and M. Wiener, “A known-plaintext attack on two-key triple encryption,” Advances in Cryptology – EUROCRYPT ’90, Lecture Notes in Computer Science 473, Springer-Verlag, 1991, pp. 318–325.
- [Wang et al.] X. Wang, D. Feng, X. Lai, H. Yu, “Collisions for hash functions MD-4, MD-5, HAVAL-128, RIPEMD,” preprint, 2004.
- [Wang et al. 2] X. Wang, Y. Yin, H. Yu, “Finding collisions in the full SHA1,” to appear in CRYPTO 2005.
- [Washington] L. Washington, Elliptic Curves: Number Theory and Cryptography, Chapman & Hall/CRC Press, 2003.
- [Welsh] D. Welsh, Codes and Cryptography, Oxford, 1988.

- [Wicker] S. Wicker, Error Control Systems for Digital Communication and Storage, Prentice Hall, 1995.
- [Wiener] M. Wiener, “Cryptanalysis of short RSA secret exponents,” IEEE Trans. Inform. Theory, 36 (1990), 553–558.
- [Williams] H. Williams, Edouard Lucas and Primality Testing, Wiley-Interscience, 1998.
- [Wu1] T. Wu, “The secure remote password protocol,” In: Proc. of the Internet Society Network and Distributed Security Symposium, 97–111, March 1998.
- [Wu2] T. Wu, “SRP-6: Improvements and refinements to the Secure Remote Password protocol,” 2002; available through <http://srp.stanford.edu/design.html>

Index

- (n, M, d) code, 445
- $GF(2^8)$, 73, 163
- $GF(4)$, 69, 397, 479
- $[n, k, d]$ code, 451
- \mathbf{Z}_p , 71
- \oplus , 108, 137
- $\phi(n)$, 57
- $a^r \equiv 1$ factorization method, 175, 192, 518, 544, 578
- $p - 1$ method, 188, 393, 396, 519, 546, 580
- q -ary code, 442
- $x^2 \equiv y^2$ factorization method, 494, 501
- G_{23} , 463
- G_{24} , 459
- 3DES, 155
- absorption, 238
- acoustic cryptanalysis, 183
- addition law, 384, 388, 404
- AddRoundKey, 161
- ADFGX cipher, 27
- Adleman, 171
- Advanced Encryption Standard (AES), 137, 160, 293
- Aesop, 37
- affine cipher, 12, 133, 380
- Agrawal, 188
- Alice, 2
- anonymity, 325, 329
- ASCII, 88

- asymptotic bounds, 449
- Athena, 299
- Atkins, 193
- attacks, 3
- attacks on RSA, 177, 426
- authenticated key agreement, 292
- authenticated key distribution, 295
- authentication, 8, 25, 196, 310, 314
- automatic teller machine, 357, 361

- baby step, giant step, 215, 249, 392
- basic principle, 55, 58, 184, 189
- basis, 421
- Batey, 283
- Bayes's theorem, 367
- BCH bound, 472
- BCH codes, 472
- Bellare, 180, 251, 255
- Berlekamp, 483
- Berson, 357
- Bertoni, 237
- Bidzos, 283
- Biham, 136, 140
- bilinear, 409
- bilinear Diffie-Hellman, 412
- bilinear pairing, 409
- binary, 88
- binary code, 442
- birthday attack, 246, 249, 250, 265, 274
- birthday paradox, 193, 246, 268, 284, 286
- bit, 88
- bit commitment, 218
- Bitcoin, 326, 330
- Blakley secret sharing scheme, 344, 348
- Bletchley Park, 33, 283
- blind signature, 271
- blind signature, restricted, 320, 325
- block cipher, 118, 137, 256

- block code, [443](#)
- blockchains, [262](#), [328](#), [334](#)
- Blom key pre-distribution scheme, [294](#)
- BLS signatures, [414](#)
- Blum-Blum-Shub, [106](#)
- Bob, [2](#)
- bombes, [33](#)
- Boneh, [177](#), [412](#), [414](#), [417](#)
- bounded storage, [90](#)
- bounds on codes, [446](#)
- Brands, [319](#), [320](#)
- breaking DES, [152](#)
- brute force attack, [6](#), [169](#)
- burst errors, [480](#)
- byte, [88](#)

- Caesar cipher, [11](#)
- Canetti, [253](#), [255](#)
- Carmichael number, [80](#)
- CBC-MAC, [256](#)
- certificate, [303](#)–[307](#), [309](#), [312](#)
- certification authority (CA), [303](#), [304](#)
- certification hierarchy, [304](#)
- certification path, [309](#)
- CESG, [171](#), [195](#)
- chain rule, [370](#)
- challenge-response, [357](#)
- characteristic 2, [396](#)
- Chaum, [228](#), [271](#), [319](#)
- cheating, [354](#)
- check symbols, [453](#)
- Chinese remainder theorem, [52](#), [53](#), [84](#), [209](#)
- chosen ciphertext attack, [3](#)
- chosen plaintext attack, [3](#)
- CI Game, [97](#), [252](#), [289](#)
- cipher block chaining (CBC), [119](#), [123](#), [134](#), [256](#)
- cipher feedback (CFB), [119](#), [123](#), [134](#), [168](#)
- ciphers, [5](#)
- ciphertext, [2](#)
- ciphertext indistinguishability, [97](#), [181](#), [207](#), [289](#)
- ciphertext only attack, [3](#)
- Cliff, [299](#)
- closest vector problem, [434](#), [435](#)
- Cocks, [171](#), [195](#)

- code, 442
- code rate, 440, 445, 446
- codes, 5
- codeword, 437, 443
- coding gain, 442
- coding theory, 1, 437
- coin, 321
- collision, 227, 228
- collision resistant, 226, 229
- composite, 41
- compression function, 231, 234, 249
- computational Diffie-Hellman, 220, 222, 412
- computationally infeasible, 195, 196
- conditional entropy, 370
- conditional probability, 94, 367
- confidentiality, 8
- confusion, 119
- congruence, 47
- continued fractions, 76, 82, 83, 178, 500
- convolutional codes, 483
- Coppersmith, 149, 151, 177
- correct errors, 444
- coset, 455
- coset leader, 455, 456
- counter mode (CTR), 128
- CRC-32, 287
- cryptanalysis, 1
- cryptocurrencies, 329
- cryptography, 1

- cryptology, 1
- cyclic codes, 466

- Daemen, [160](#), [237](#)
- Damgård, [231](#)
- Data Encryption Standard (DES), [131](#), [136](#), [145](#), [156](#)
- Daum, [232](#)
- decision Diffie-Hellman, [220](#), [222](#), [420](#)
- decode, [442](#)
- DES Challenge, [153](#)
- DES Cracker, [153](#)
- DESX, [129](#)
- detect errors, [444](#)
- deterministic, [249](#)
- Di Crescenzo, [417](#)
- dictionary attack, [155](#)
- differential cryptanalysis, [140](#), [168](#)
- Diffie, [152](#), [171](#), [195](#), [292](#)
- Diffie-Hellman, [220](#), [222](#), [411](#), [420](#)
- Diffie-Hellman key exchange, [219](#), [291](#), [401](#), [526](#), [554](#), [590](#)
- diffusion, [118](#), [168](#), [169](#)
- digital cash, [320](#)
- digital signature, [8](#), [292](#), [401](#)
- Digital Signature Algorithm (DSA), [215](#), [275](#), [402](#), [406](#)
- digram, [21](#), [27](#), [121](#), [376](#)
- Ding, [90](#)
- discrete logarithm, [74](#), [84](#), [211](#), [228](#), [249](#), [272](#), [324](#), [352](#), [361](#), [363](#), [391](#), [399](#), [410](#)
- Disparition, La, [15](#)
- divides, [40](#)
- dot product, [18](#), [35](#), [442](#), [453](#), [456](#), [489](#)

- double encryption, [129](#), [130](#), [149](#), [198](#), [203](#)
- dual code, [456](#), [485](#)
- dual signature, [315](#)

- electronic cash, 9
- electronic codebook (ECB), 119, 122
- Electronic Frontier Foundation, 153
- electronic voting, 207
- ElGamal cryptosystem, 196, 221, 400, 525, 553, 589
- ElGamal signature, 271, 276, 278, 279, 401, 407
- elliptic curve cryptosystems, 399
- elliptic curves, 189, 384
- elliptic integral, 386
- Ellis, 171
- encode, 442
- Enigma, 29, 282
- entropy, 367, 368
- entropy of English, 376
- entropy rate, 382
- equivalent codes, 445
- error correcting codes, 437, 442
- error correction, 26
- error propagation, 119
- Euclidean algorithm, 43, 82, 85
- Euler's ϕ -function, 57, 81, 174
- Euler's theorem, 57, 173
- Eve, 2
- everlasting security, 90
- existential forgery, 278
- expansion permutation, 145
- extended Euclidean algorithm, 44, 72

- factor base, [190](#), [216](#)
- factoring, [188](#), [191](#), [212](#), [393](#), [494](#)
- factorization records, [191](#), [212](#)
- Feige-Fiat-Shamir identification, [359](#)
- Feistel system, [137](#), [138](#), [168](#)
- Feng, [227](#)
- Fermat factorization, [188](#)
- Fermat prime, [82](#)
- Fermat's theorem, [55](#), [184](#)
- Feynman, [488](#)
- Fibonacci numbers, [78](#)
- field, [70](#)
- finite field, [69](#), [163](#), [396](#), [397](#), [468](#)
- Flame, [227](#)
- flipping coins, [349](#)
- football, [218](#)
- Fourier transform, [495](#), [499](#), [502](#)
- fractions, [51](#)
- Franklin, [412](#)
- fraud control, [324](#)
- frequencies of letters, [14](#), [21](#)
- frequency analysis, [21](#)

- Gadsby, [14](#)
- games, [9](#), [349](#)
- generating matrix, [452](#)
- generating polynomial, [468](#)
- Gentry, [206](#)
- Gilbert-Varshamov bound, [448](#), [450](#), [485](#)
- Golay code, [459](#)
- Goldberg, [283](#)
- Goldreich, [253](#)
- Goppa codes, [449](#), [481](#)
- Graff, [193](#)
- Grant, [300](#)
- greatest common divisor (gcd), [42](#), [85](#)
- group, [149](#)
- Guillou, [357](#)

- Hadamard code, 441, 450
- Halevi, 253
- Hamming bound, 447
- Hamming code, 439, 450, 457, 485
- Hamming distance, 443
- Hamming sphere, 446
- Hamming weight, 452
- hash function, 226–228, 230, 231, 233, 251, 253, 273, 312, 315, 336, 361
- hash pointer, 262
- Hasse's theorem, 391, 407, 408
- Hellman, 129, 152, 171, 195, 213
- Hess, 415
- hexadecimal, 234
- Hill cipher, 119
- HMAC, 255
- Holmes, Sherlock, 23
- homomorphic encryption, 206
- hot line, 90
- Huffman codes, 371, 378

- IBM, 136, 160
- ID-based signatures, 415
- identification scheme, 9, 359
- identity-based encryption, 412
- independent, 94, 366
- index calculus, 216, 391, 399
- indistinguishability, 252
- infinity, 385
- information rate, 445
- information symbols, 453
- information theory, 365
- initial permutation, 145, 149
- integrity, 8, 228, 314
- intruder-in-the-middle, 59, 290, 291, 317
- inverting matrices, 61
- irreducible polynomial, 71
- ISBN, 440, 484
- IV, 123, 126, 231, 249, 256, 285

- Jacobi symbol, [64](#), [66](#), [187](#)

- Joux, [249](#), [411](#)

- Kayal, [188](#)
- Keccak, [237](#)
- Kerberos, [299](#)
- Kerckhoffs's principle, [4](#)
- ket, [489](#)
- key, [2](#)
- key agreement, [293](#)
- key distribution, [293](#), [491](#)
- key establishment, [9](#)
- key exchange, [401](#)
- key length, [5](#), [16](#), [384](#), [482](#)
- key permutation, [148](#)
- key pre-distribution, [294](#)
- key schedule, [165](#), [169](#)
- keyring, [309](#)
- keyword search, [417](#)
- knapsack problem, [195](#)
- known plaintext attack, [3](#)
- Koblitz, [384](#), [392](#)
- Kocher, [181](#)
- Krawczyk, [255](#)

- Lagrange interpolation, 341, 343
- Lai, 227
- Lamport, 260
- lattice, 421
- lattice reduction, 422
- ledger, 328, 331
- Legendre symbol, 64, 82
- length, 107
- length extension attack, 232, 255
- Lenstra, A., 193, 227, 284, 425
- Lenstra, H. W., 384, 425
- Leyland, 193
- linear code, 451
- linear congruential generator, 105
- linear cryptanalysis, 144, 168
- linear feedback shift register (LFSR), 74, 107
- LLL algorithm, 425, 427
- Lovász, L., 425
- LUCIFER, 137
- Luks, 232
- Lynn, 414

- MAC, 255, 313
- Mantin, 114
- Maple, 527
- Mariner, 441
- MARS, 160
- Massey, 59, 483
- Mathematica, 503
- MATLAB, 555
- matrices, 61
- Matsui, 144
- Mauborgne, 88
- Maurer, 90
- McEliece cryptosystem, 197, 435, 480
- MD5, 227, 233
- MDS code, 446, 450
- meet-in-the-middle attack, 129, 130, 133
- Menezes, 400, 410
- Merkle, 129, 231
- Merkle tree, 263, 337
- Merkle-Damgård, 231
- message authentication code (MAC), 255, 313
- message digest, 226
- message recovery scheme, 273
- Mikle, 227
- Miller, 384
- Miller-Rabin primality test, 185, 209
- minimum distance, 444
- mining, 327

- MIT, [299](#)
- MixColumns transformation, [161](#), [164](#), [169](#)
- mod, [47](#)
- modes of operation, [122](#)
- modular exponentiation, [54](#), [84](#), [182](#), [276](#)
- Morse code, [372](#)
- MOV attack, [410](#)
- multicollision, [249](#), [253](#), [268](#)
- multiple encryption, [129](#)
- multiplicative inverse, [49](#), [73](#), [165](#)

- Nakamoto, 330
- National Bureau of Standards (NBS), 136, 152
- National Institute of Standards and Technology (NIST), 136, 152,
160, 233, 384, 397
- National Security Agency (NSA), 37, 136, 152, 233
- nearest neighbor decoding, 444
- Needham–Schroeder protocol, 297
- Netscape, 283
- Newton interpolating polynomial, 348
- NIST, 237
- non-repudiation, 8, 196
- nonce, 296, 327, 337, 339
- NP-complete, 456
- NTRU, 197, 429
- number field sieve, 191

- OAEP, 180, 252
- Ohta, 318
- Okamoto, 318, 410
- Omura, 59
- one-time pad, 88, 89, 91, 97, 252, 282, 286, 375, 380
- one-way function, 105, 155, 196, 212, 219, 226
- order, 83, 404
- Ostrovsky, 417
- output feedback (OFB), 126

- padding, 180, 235, 238, 255
- Paillier cryptosystem, 206
- Painvin, 28
- pairing, 409
- parity check, 438
- parity check matrix, 453, 470
- passwords, 155, 224, 256, 258, 260
- Peeters, 237
- Peggy, 357
- Pell's equation, 83
- perfect code, 447, 485, 486
- perfect secrecy, 95, 373, 374
- Persiano, 417
- Pfitzmann, 228
- plaintext, 2, 392
- plaintext-aware encryption, 181
- Playfair cipher, 26
- Pohlig-Hellman algorithm, 213, 224, 276, 391, 407
- point at infinity, 385
- poker, 351
- polarization, 489
- Pollard, 188
- post-quantum cryptography, 435
- PostScript, 232
- preimage resistant, 226
- Pretty Good Privacy (PGP), 309
- PRGA, 114
- primality testing, 183

- prime, 41
- prime number theorem, 41, 185, 245, 279
- primitive root, 59, 82, 83
- primitive root of unity, 472
- probabilistic, 249
- probability, 365
- provable security, 94
- pseudoprime, 185
- pseudorandom, 238, 284
- pseudorandom bits, 105
- public key cryptography, 4, 171, 195
- Public Key Infrastructure (PKI), 303

- quadratic reciprocity, [67](#)
- quadratic residue, [354](#), [355](#)
- quadratic residuosity problem, [69](#)
- quadratic sieve, [205](#)
- quantum computing, [493](#)
- quantum cryptography, [488](#), [491](#)
- quantum Fourier transform, [499](#)
- qubit, [491](#)
- Quisquater, [357](#)

- R Game, 98, 114
- Ró ycki, 29
- Rabin, 84, 90
- random oracle model, 251
- random variable, 366
- RC4, 113, 285
- RC6, 160
- recurrence relation, 74, 107
- reduced basis, 422
- redundancy, 379
- Reed-Solomon codes, 479
- registration authority (RA), 304
- Rejewski, 29, 32
- relatively prime, 42
- repetition code, 437, 450
- restricted blind signature, 320, 325
- Rijmen, 160
- Rijndael, 160
- Rivest, 113, 129, 171, 233
- Rogaway, 180, 251
- root of unity, 472
- rotation, 230
- rotor machines, 29
- round constant, 165, 169
- round key, 165
- RoundKey addition, 164
- RSA, 97, 171, 172, 174, 222, 225, 283, 293, 376, 426, 433
- RSA challenge, 192

- RSA signature, [194](#), [270](#), [310](#)
- run length coding, [381](#)

- S-box, 139, 148–150, 163, 165, 168
- Safavi-Naini, 415
- Sage, 591
- salt, 155, 258
- Saxena, 188
- Schacham, 414
- Scherbis, 29
- Schnorr identification scheme, 363
- secret sharing, 9, 340
- secret splitting, 340
- Secure Electronic Transaction (SET), 314
- Secure Hash Algorithm (SHA), 227, 233, 235
- Secure Remote Password (SRP) protocol, 258
- Security Sockets Layer (SSL), 312
- seed, 98, 105
- self-dual code, 457
- sequence numbers, 296
- Serge, 299
- Serpent, 160
- SHA-3, 237
- SHAKE, 238
- Shamir, 59, 114, 136, 140, 171, 412
- Shamir threshold scheme, 341
- Shannon, 118, 365, 368, 371, 377, 378
- shift cipher, 11, 97
- ShiftRows transformation, 161, 164, 169
- Shor, 488, 493
- Shor’s algorithm, 493, 497

- shortest vector, 424, 425
- shortest vector problem, 422
- side-channel attacks, 183
- signature with appendix, 273
- Singleton bound, 446
- singular curves, 395
- smooth, 394
- Solovay-Strassen, 187
- sphere packing bound, 447
- sponge function, 237
- square roots, 53, 62, 85, 218, 349, 358, 362
- squeamish ossifrage, 194
- squeezing, 238
- state, 237
- station-to-station (STS) protocol, 292
- stream cipher, 104
- strong pseudoprime, 185, 209
- strongly collision resistant, 226
- SubBytes transformation, 161, 163
- substitution cipher, 20, 380
- supersingular, 410, 419
- Susilo, 415
- Sybil attack, 338
- symmetric key, 4, 196
- syndrome, 455, 456
- syndrome decoding, 456
- systematic code, 453

- ternary code, [442](#)
- three-pass protocol, [58](#), [198](#), [282](#), [317](#)
- threshold scheme, [341](#)
- ticket-granting service, [300](#)
- timestamps, [296](#)
- timing attacks, [181](#)
- Transmission Control Protocol (TCP), [483](#)
- Transport Layer Security (TLS), [312](#)
- trapdoor, [136](#), [196](#), [197](#)
- treaty verification, [194](#)
- Trent, [300](#)
- triangle inequality, [444](#)
- trigram, [121](#), [376](#)
- tripartite Diffie-Hellman, [411](#)
- triple encryption, [129](#), [131](#)
- trust, [304](#), [309](#)
- trusted authority, [292](#), [294](#), [295](#), [300](#), [361](#)
- Turing, [33](#)
- two lists, [180](#)
- two-dimensional parity code, [438](#)
- Twofish, [160](#)

- unicity distance, [379](#)

- Unix, [156](#)

- Van Assche, 237
- van Heijst, 228
- van Oorschot, 292
- Vandermonde determinant, 342, 473
- Vanstone, 400, 410
- variance, 182
- Vernam, 88
- Verser, 153
- Victor, 357
- Vigenère cipher, 14
- Void, A, 15
- Voyager, 459

- Wagner, [283](#)
- Wang, [227](#)
- weak key, [151](#), [158](#), [159](#), [168](#)
- web of trust, [309](#)
- Weil pairing, [410](#)
- WEP, [284](#)
- Wiener, [152](#), [178](#), [292](#)
- World War I, [5](#), [26](#)
- World War II, [3](#), [29](#), [33](#), [294](#)
- WPA, [284](#)

- X.509, [227](#), [245](#), [284](#), [304–307](#), [312](#)
- XOR, [89](#), [137](#)

- Yin, 227

- Yu, 227

- zero-knowledge, 357
- Zhang, 415
- Zimmerman, 309
- Zygalski, 29

Contents

1. Introduction to Cryptography with Coding Theory
2. Contents
3. Preface
4. Chapter 1 Overview of Cryptography and Its Applications
 1. 1.1 Secure Communications
 1. 1.1.1 Possible Attacks
 2. 1.1.2 Symmetric and Public Key Algorithms
 3. 1.1.3 Key Length
 2. 1.2 Cryptographic Applications
5. Chapter 2 Classical Cryptosystems
 1. 2.1 Shift Ciphers
 2. 2.2 Affine Ciphers
 3. 2.3 The Vigenère Cipher
 1. 2.3.1 Finding the Key Length
 2. 2.3.2 Finding the Key: First Method
 3. 2.3.3 Finding the Key: Second Method
 4. 2.4 Substitution Ciphers
 5. 2.5 Sherlock Holmes
 6. 2.6 The Playfair and ADFGX Ciphers
 7. 2.7 Enigma
 8. 2.8 Exercises
 9. 2.9 Computer Problems
6. Chapter 3 Basic Number Theory
 1. 3.1 Basic Notions
 1. 3.1.1 Divisibility
 2. 3.1.2 Prime Numbers
 3. 3.1.3 Greatest Common Divisor
 2. 3.2 The Extended Euclidean Algorithm
 3. 3.3 Congruences
 1. 3.3.1 Division
 2. 3.3.2 Working with Fractions

- 4. 3.4 The Chinese Remainder Theorem
- 5. 3.5 Modular Exponentiation
- 6. 3.6 Fermat's Theorem and Euler's Theorem

- 1. 3.6.1 Three-Pass Protocol

- 7. 3.7 Primitive Roots
- 8. 3.8 Inverting Matrices Mod n
- 9. 3.9 Square Roots Mod n
- 10. 3.10 Legendre and Jacobi Symbols
- 11. 3.11 Finite Fields

- 1. 3.11.1 Division
- 2. 3.11.2 $GF(2^8)$
- 3. 3.11.3 LFSR Sequences

- 12. 3.12 Continued Fractions
- 13. 3.13 Exercises
- 14. 3.14 Computer Problems

- 7. Chapter 4 The One-Time Pad

- 1. 4.1 Binary Numbers and ASCII
- 2. 4.2 One-Time Pads
- 3. 4.3 Multiple Use of a One-Time Pad
- 4. 4.4 Perfect Secrecy of the One-Time Pad
- 5. 4.5 Indistinguishability and Security
- 6. 4.6 Exercises

- 8. Chapter 5 Stream Ciphers

- 1. 5.1 Pseudorandom Bit Generation
- 2. 5.2 Linear Feedback Shift Register Sequences
- 3. 5.3 RC4
- 4. 5.4 Exercises
- 5. 5.5 Computer Problems

- 9. Chapter 6 Block Ciphers

- 1. 6.1 Block Ciphers
- 2. 6.2 Hill Ciphers
- 3. 6.3 Modes of Operation
 - 1. 6.3.1 Electronic Codebook (ECB)
 - 2. 6.3.2 Cipher Block Chaining (CBC)
 - 3. 6.3.3 Cipher Feedback (CFB)
 - 4. 6.3.4 Output Feedback (OFB)
 - 5. 6.3.5 Counter (CTR)
- 4. 6.4 Multiple Encryption
- 5. 6.5 Meet-in-the-Middle Attacks

6. 6.6 Exercises

7. 6.7 Computer Problems

10. Chapter 7 The Data Encryption Standard

1. 7.1 Introduction

2. 7.2 A Simplified DES-Type Algorithm

3. 7.3 Differential Cryptanalysis

1. 7.3.1 Differential Cryptanalysis for Three Rounds

2. 7.3.2 Differential Cryptanalysis for Four Rounds

4. 7.4 DES

1. 7.4.1 DES Is Not a Group

5. 7.5 Breaking DES

6. 7.6 Password Security

7. 7.7 Exercises

8. 7.8 Computer Problems

11. Chapter 8 The Advanced Encryption Standard: Rijndael

1. 8.1 The Basic Algorithm

2. 8.2 The Layers

1. 8.2.1 The SubBytes Transformation

2. 8.2.2 The ShiftRows Transformation

3. 8.2.3 The MixColumns Transformation

4. 8.2.4 The RoundKey Addition

5. 8.2.5 The Key Schedule

6. 8.2.6 The Construction of the S-Box

3. 8.3 Decryption

4. 8.4 Design Considerations

5. 8.5 Exercises

12. Chapter 9 The RSA Algorithm

1. 9.1 The RSA Algorithm

2. 9.2 Attacks on RSA

1. 9.2.1 Low Exponent Attacks

2. 9.2.2 Short Plaintext

3. 9.2.3 Timing Attacks

3. 9.3 Primality Testing

4. 9.4 Factoring

- 1. [9.4.1 \$x\$ ----- \$y\$](#)
- 2. [9.4.2 Using \$a^r\$](#)

- 5. [9.5 The RSA Challenge](#)
- 6. [9.6 An Application to Treaty Verification](#)
- 7. [9.7 The Public Key Concept](#)
- 8. [9.8 Exercises](#)
- 9. [9.9 Computer Problems](#)

13. Chapter 10 Discrete Logarithms

- 1. [10.1 Discrete Logarithms](#)
- 2. [10.2 Computing Discrete Logs](#)
 - 1. [10.2.1 The Pohlig-Hellman Algorithm](#)
 - 2. [10.2.2 Baby Step, Giant Step](#)
 - 3. [10.2.3 The Index Calculus](#)
 - 4. [10.2.4 Computing Discrete Logs Mod 4](#)
- 3. [10.3 Bit Commitment](#)
- 4. [10.4 Diffie-Hellman Key Exchange](#)
- 5. [10.5 The ElGamal Public Key Cryptosystem](#)
 - 1. [10.5.1 Security of ElGamal Ciphertexts](#)
- 6. [10.6 Exercises](#)
- 7. [10.7 Computer Problems](#)

14. Chapter 11 Hash Functions

- 1. [11.1 Hash Functions](#)
- 2. [11.2 Simple Hash Examples](#)
- 3. [11.3 The Merkle-Damgård Construction](#)
- 4. [11.4 SHA-2](#)
 - 1. [Padding and Preprocessing](#)
 - 2. [The Algorithm](#)
- 5. [11.5 SHA-3/Keccak](#)
- 6. [11.6 Exercises](#)

15. Chapter 12 Hash Functions: Attacks and Applications

- 1. [12.1 Birthday Attacks](#)
 - 1. [12.1.1 A Birthday Attack on Discrete Logarithms](#)
 - 2. [12.2 Multicollisions](#)
 - 3. [12.3 The Random Oracle Model](#)

4. [12.4 Using Hash Functions to Encrypt](#)
5. [12.5 Message Authentication Codes](#)

1. [12.5.1 HMAC](#)
2. [12.5.2 CBC-MAC](#)

6. [12.6 Password Protocols](#)

1. [12.6.1 The Secure Remote Password protocol](#)
2. [12.6.2 Lamport's protocol](#)

7. [12.7 Blockchains](#)

8. [12.8 Exercises](#)
9. [12.9 Computer Problems](#)

16. Chapter 13 Digital Signatures

1. [13.1 RSA Signatures](#)
2. [13.2 The ElGamal Signature Scheme](#)
3. [13.3 Hashing and Signing](#)
4. [13.4 Birthday Attacks on Signatures](#)
5. [13.5 The Digital Signature Algorithm](#)
6. [13.6 Exercises](#)
7. [13.7 Computer Problems](#)

17. Chapter 14 What Can Go Wrong

1. [14.1 An Enigma “Feature”](#)
2. [14.2 Choosing Primes for RSA](#)
3. [14.3 WEP](#)
 1. [14.3.1 CRC-32](#)
4. [14.4 Exercises](#)

18. Chapter 15 Security Protocols

1. [15.1 Intruders-in-the-Middle and Impostors](#)
 1. [15.1.1 Intruder-in-the-Middle Attacks](#)
 2. [15.2 Key Distribution](#)
 1. [15.2.1 Key Pre-distribution](#)
 2. [15.2.2 Authenticated Key Distribution](#)
 3. [15.3 Kerberos](#)
 4. [15.4 Public Key Infrastructures \(PKI\)](#)
 5. [15.5 X.509 Certificates](#)
 6. [15.6 Pretty Good Privacy](#)

- 7. [15.7 SSL and TLS](#)
- 8. [15.8 Secure Electronic Transaction](#)
- 9. [15.9 Exercises](#)

19. Chapter 16 Digital Cash

- 1. [16.1 Setting the Stage for Digital Economies](#)
- 2. [16.2 A Digital Cash System](#)
 - 1. [16.2.1 Participants](#)
 - 2. [16.2.2 Initialization](#)
 - 3. [16.2.3 The Bank](#)
 - 4. [16.2.4 The Spender](#)
 - 5. [16.2.5 The Merchant](#)
 - 6. [16.2.6 Creating a Coin](#)
 - 7. [16.2.7 Spending the Coin](#)
 - 8. [16.2.8 The Merchant Deposits the Coin in the Bank](#)
 - 9. [16.2.9 Fraud Control](#)
 - 10. [16.2.10 Anonymity](#)
- 3. [16.3 Bitcoin Overview](#)
 - 1. [16.3.1 Some More Details](#)
- 4. [16.4 Cryptocurrencies](#)
- 5. [16.5 Exercises](#)

20. Chapter 17 Secret Sharing Schemes

- 1. [17.1 Secret Splitting](#)
- 2. [17.2 Threshold Schemes](#)
- 3. [17.3 Exercises](#)
- 4. [17.4 Computer Problems](#)

21. Chapter 18 Games

- 1. [18.1 Flipping Coins over the Telephone](#)
- 2. [18.2 Poker over the Telephone](#)
 - 1. [18.2.1 How to Cheat](#)
- 3. [18.3 Exercises](#)

22. Chapter 19 Zero-Knowledge Techniques

- 1. [19.1 The Basic Setup](#)
- 2. [19.2 The Feige-Fiat-Shamir Identification Scheme](#)
- 3. [19.3 Exercises](#)

23. Chapter 20 Information Theory

- 1. 20.1 Probability Review
- 2. 20.2 Entropy
- 3. 20.3 Huffman Codes
- 4. 20.4 Perfect Secrecy
- 5. 20.5 The Entropy of English

- 1. 20.5.1 Unicity Distance

- 6. 20.6 Exercises

24. Chapter 21 Elliptic Curves

- 1. 21.1 The Addition Law
- 2. 21.2 Elliptic Curves Mod p

- 1. 21.2.1 Number of Points Mod p
- 2. 21.2.2 Discrete Logarithms on Elliptic Curves
- 3. 21.2.3 Representing Plaintext

- 3. 21.3 Factoring with Elliptic Curves

- 1. 21.3.1 Singular Curves

- 4. 21.4 Elliptic Curves in Characteristic 2
- 5. 21.5 Elliptic Curve Cryptosystems

- 1. 21.5.1 An Elliptic Curve ElGamal Cryptosystem
- 2. 21.5.2 Elliptic Curve Diffie-Hellman Key Exchange
- 3. 21.5.3 ElGamal Digital Signatures

- 6. 21.6 Exercises

- 7. 21.7 Computer Problems

25. Chapter 22 Pairing-Based Cryptography

- 1. 22.1 Bilinear Pairings
- 2. 22.2 The MOV Attack
- 3. 22.3 Tripartite Diffie-Hellman
- 4. 22.4 Identity-Based Encryption
- 5. 22.5 Signatures

- 1. 22.5.1 BLS Signatures
- 2. 22.5.2 A Variation
- 3. 22.5.3 Identity-Based Signatures

- 6. 22.6 Keyword Search

- 7. 22.7 Exercises

26. Chapter 23 Lattice Methods

- 1. 23.1 Lattices
- 2. 23.2 Lattice Reduction
 - 1. 23.2.1 Two-Dimensional Lattices
 - 2. 23.2.2 The LLL algorithm
- 3. 23.3 An Attack on RSA
- 4. 23.4 NTRU
 - 1. 23.4.1 An Attack on NTRU
- 5. 23.5 Another Lattice-Based Cryptosystem
- 6. 23.6 Post-Quantum Cryptography?
- 7. 23.7 Exercises

27. Chapter 24 Error Correcting Codes

- 1. 24.1 Introduction
- 2. 24.2 Error Correcting Codes
- 3. 24.3 Bounds on General Codes
 - 1. 24.3.1 Upper Bounds
 - 2. 24.3.2 Lower Bounds
- 4. 24.4 Linear Codes
 - 1. 24.4.1 Dual Codes
- 5. 24.5 Hamming Codes
- 6. 24.6 Golay Codes
 - 1. Decoding
- 7. 24.7 Cyclic Codes
- 8. 24.8 BCH Codes
 - 1. 24.8.1 Decoding BCH Codes
- 9. 24.9 Reed-Solomon Codes
- 10. 24.10 The McEliece Cryptosystem
- 11. 24.11 Other Topics
- 12. 24.12 Exercises
- 13. 24.13 Computer Problems

28. Chapter 25 Quantum Techniques in Cryptography

- 1. 25.1 A Quantum Experiment
- 2. 25.2 Quantum Key Distribution

3. [25.3 Shor's Algorithm](#)

1. [25.3.1 Factoring](#)
2. [25.3.2 The Discrete Fourier Transform](#)
3. [25.3.3 Shor's Algorithm](#)
4. [25.3.4 Final Words](#)

4. [25.4 Exercises](#)

29. [Appendix A Mathematica[®] Examples](#)

1. [A.1 Getting Started with Mathematica](#)
2. [A.2 Some Commands](#)
3. [A.3 Examples for Chapter 2](#)
4. [A.4 Examples for Chapter 3](#)
5. [A.5 Examples for Chapter 5](#)
6. [A.6 Examples for Chapter 6](#)
7. [A.7 Examples for Chapter 9](#)
8. [A.8 Examples for Chapter 10](#)
9. [A.9 Examples for Chapter 12](#)
10. [A.10 Examples for Chapter 17](#)
11. [A.11 Examples for Chapter 18](#)
12. [A.12 Examples for Chapter 21](#)

30. [Appendix B Maple[®] Examples](#)

1. [B.1 Getting Started with Maple](#)
2. [B.2 Some Commands](#)
3. [B.3 Examples for Chapter 2](#)
4. [B.4 Examples for Chapter 3](#)
5. [B.5 Examples for Chapter 5](#)
6. [B.6 Examples for Chapter 6](#)
7. [B.7 Examples for Chapter 9](#)
8. [B.8 Examples for Chapter 10](#)
9. [B.9 Examples for Chapter 12](#)
10. [B.10 Examples for Chapter 17](#)
11. [B.11 Examples for Chapter 18](#)
12. [B.12 Examples for Chapter 21](#)

31. [Appendix C MATLAB[®] Examples](#)

1. [C.1 Getting Started with MATLAB](#)
2. [C.2 Examples for Chapter 2](#)
3. [C.3 Examples for Chapter 3](#)
4. [C.4 Examples for Chapter 5](#)
5. [C.5 Examples for Chapter 6](#)
6. [C.6 Examples for Chapter 9](#)
7. [C.7 Examples for Chapter 10](#)
8. [C.8 Examples for Chapter 12](#)
9. [C.9 Examples for Chapter 17](#)
10. [C.10 Examples for Chapter 18](#)
11. [C.11 Examples for Chapter 21](#)

[32. Appendix D Sage Examples](#)

1. [D.1 Computations for Chapter 2](#)
2. [D.2 Computations for Chapter 3](#)
3. [D.3 Computations for Chapter 5](#)
4. [D.4 Computations for Chapter 6](#)
5. [D.5 Computations for Chapter 9](#)
6. [D.6 Computations for Chapter 10](#)
7. [D.7 Computations for Chapter 12](#)
8. [D.8 Computations for Chapter 17](#)
9. [D.9 Computations for Chapter 18](#)
10. [D.10 Computations for Chapter 21](#)

[33. Appendix E Answers and Hints for Selected Odd-Numbered Exercises](#)

[34. Appendix F Suggestions for Further Reading](#)

[35. Bibliography](#)

[36. Index](#)

Landmarks

1. [Frontmatter](#)
2. [Start of Content](#)
3. [backmatter](#)

1. [i](#)

2. [ii](#)

3. [iii](#)

4. [iv](#)

5. [v](#)

6. [vi](#)

7. [vii](#)

8. [viii](#)

9. [ix](#)

10. [x](#)

11. [xi](#)

12. [xii](#)

13. [xiii](#)

14. [xiv](#)

15. [1](#)

16. [2](#)

17. [3](#)

18. [4](#)

19. [5](#)

20. [6](#)

21. [7](#)

22. [8](#)

23. [9](#)

24. [10](#)

25. [11](#)

26. 12
27. 13
28. 14
29. 15
30. 16
31. 17
32. 18
33. 19
34. 20
35. 21
36. 22
37. 23
38. 24
39. 25
40. 26
41. 27
42. 28
43. 29
44. 30
45. 31
46. 32
47. 33
48. 34
49. 35
50. 36
51. 37
52. 38
53. 39
54. 40
55. 41
56. 42
57. 43
58. 44
59. 45
60. 46
61. 47
62. 48
63. 49
64. 50
65. 51
66. 52
67. 53
68. 54
69. 55
70. 56
71. 57
72. 58
73. 59
74. 60
75. 61
76. 62
77. 63
78. 64
79. 65

80. 66
81. 67
82. 68
83. 69
84. 70
85. 71
86. 72
87. 73
88. 74
89. 75
90. 76
91. 77
92. 78
93. 79
94. 80
95. 81
96. 82
97. 83
98. 84
99. 85
100. 86
101. 87
102. 88
103. 89
104. 90
105. 91
106. 92
107. 93
108. 94
109. 95
110. 96
111. 97
112. 98
113. 99
114. 100
115. 101
116. 102
117. 103
118. 104
119. 105
120. 106
121. 107
122. 108
123. 109
124. 110
125. 111
126. 112
127. 113
128. 114
129. 115
130. 116
131. 117
132. 118
133. 119

134. 120
135. 121
136. 122
137. 123
138. 124
139. 125
140. 126
141. 127
142. 128
143. 129
144. 130
145. 131
146. 132
147. 133
148. 134
149. 135
150. 136
151. 137
152. 138
153. 139
154. 140
155. 141
156. 142
157. 143
158. 144
159. 145
160. 146
161. 147
162. 148
163. 149
164. 150
165. 151
166. 152
167. 153
168. 154
169. 155
170. 156
171. 157
172. 158
173. 159
174. 160
175. 161
176. 162
177. 163
178. 164
179. 165
180. 166
181. 167
182. 168
183. 169
184. 170
185. 171
186. 172
187. 173

188. 174
189. 175
190. 176
191. 177
192. 178
193. 179
194. 180
195. 181
196. 182
197. 183
198. 184
199. 185
200. 186
201. 187
202. 188
203. 189
204. 190
205. 191
206. 192
207. 193
208. 194
209. 195
210. 196
211. 197
212. 198
213. 199
214. 200
215. 201
216. 202
217. 203
218. 204
219. 205
220. 206
221. 207
222. 208
223. 209
224. 210
225. 211
226. 212
227. 213
228. 214
229. 215
230. 216
231. 217
232. 218
233. 219
234. 220
235. 221
236. 222
237. 223
238. 224
239. 225
240. 226
241. 227

242. 228
243. 229
244. 230
245. 231
246. 232
247. 233
248. 234
249. 235
250. 236
251. 237
252. 238
253. 239
254. 240
255. 241
256. 242
257. 243
258. 244
259. 245
260. 246
261. 247
262. 248
263. 249
264. 250
265. 251
266. 252
267. 253
268. 254
269. 255
270. 256
271. 257
272. 258
273. 259
274. 260
275. 261
276. 262
277. 263
278. 264
279. 265
280. 266
281. 267
282. 268
283. 269
284. 270
285. 271
286. 272
287. 273
288. 274
289. 275
290. 276
291. 277
292. 278
293. 279
294. 280
295. 281

296. 282
297. 283
298. 284
299. 285
300. 286
301. 287
302. 288
303. 289
304. 290
305. 291
306. 292
307. 293
308. 294
309. 295
310. 296
311. 297
312. 298
313. 299
314. 300
315. 301
316. 302
317. 303
318. 304
319. 305
320. 306
321. 307
322. 308
323. 309
324. 310
325. 311
326. 312
327. 313
328. 314
329. 315
330. 316
331. 317
332. 318
333. 319
334. 320
335. 321
336. 322
337. 323
338. 324
339. 325
340. 326
341. 327
342. 328
343. 329
344. 330
345. 331
346. 332
347. 333
348. 334
349. 335

350. 336
351. 337
352. 338
353. 339
354. 340
355. 341
356. 342
357. 343
358. 344
359. 345
360. 346
361. 347
362. 348
363. 349
364. 350
365. 351
366. 352
367. 353
368. 354
369. 355
370. 356
371. 357
372. 358
373. 359
374. 360
375. 361
376. 362
377. 363
378. 364
379. 365
380. 366
381. 367
382. 368
383. 369
384. 370
385. 371
386. 372
387. 373
388. 374
389. 375
390. 376
391. 377
392. 378
393. 379
394. 380
395. 381
396. 382
397. 383
398. 384
399. 385
400. 386
401. 387
402. 388
403. 389

404. 390
405. 391
406. 392
407. 393
408. 394
409. 395
410. 396
411. 397
412. 398
413. 399
414. 400
415. 401
416. 402
417. 403
418. 404
419. 405
420. 406
421. 407
422. 408
423. 409
424. 410
425. 411
426. 412
427. 413
428. 414
429. 415
430. 416
431. 417
432. 418
433. 419
434. 420
435. 421
436. 422
437. 423
438. 424
439. 425
440. 426
441. 427
442. 428
443. 429
444. 430
445. 431
446. 432
447. 433
448. 434
449. 435
450. 436
451. 437
452. 438
453. 439
454. 440
455. 441
456. 442
457. 443

458. 444
459. 445
460. 446
461. 447
462. 448
463. 449
464. 450
465. 451
466. 452
467. 453
468. 454
469. 455
470. 456
471. 457
472. 458
473. 459
474. 460
475. 461
476. 462
477. 463
478. 464
479. 465
480. 466
481. 467
482. 468
483. 469
484. 470
485. 471
486. 472
487. 473
488. 474
489. 475
490. 476
491. 477
492. 478
493. 479
494. 480
495. 481
496. 482
497. 483
498. 484
499. 485
500. 486
501. 487
502. 488
503. 489
504. 490
505. 491
506. 492
507. 493
508. 494
509. 495
510. 496
511. 497

512. 498
513. 499
514. 500
515. 501
516. 502
517. 503
518. 504
519. 505
520. 506
521. 507
522. 508
523. 509
524. 510
525. 511
526. 512
527. 513
528. 514
529. 515
530. 516
531. 517
532. 518
533. 519
534. 520
535. 521
536. 522
537. 523
538. 524
539. 525
540. 526
541. 527
542. 528
543. 529
544. 530
545. 531
546. 532
547. 533
548. 534
549. 535
550. 536
551. 537
552. 538
553. 539
554. 540
555. 541
556. 542
557. 543
558. 544
559. 545
560. 546
561. 547
562. 548
563. 549
564. 550
565. 551

566. 552
567. 553
568. 554
569. 555
570. 556
571. 557
572. 558
573. 559
574. 560
575. 561
576. 562
577. 563
578. 564
579. 565
580. 566
581. 567
582. 568
583. 569
584. 570
585. 571
586. 572
587. 573
588. 574
589. 575
590. 576
591. 577
592. 578
593. 579
594. 580
595. 581
596. 582
597. 583
598. 584
599. 585
600. 586
601. 587
602. 588
603. 589
604. 590
605. 591
606. 592
607. 593
608. 594
609. 595
610. 596
611. 597
612. 598
613. 599
614. 600
615. 601
616. 602
617. 603
618. 604
619. 605

620. 606
621. 607
622. 608
623. 609
624. 610
625. 611
626. 612
627. 613
628. 614
629. 615
630. 616
631. 617
632. 618
633. 619
634. 620
635. 621

Long description

The illustration shows Alice is connected to encrypt with an arrow labeled as plaintext pointed towards encrypt. Encrypt is connected to decrypt with an arrow labeled as ciphertext. Decrypt is connected to Bob. Encrypt is connected with Encryption Key with an arrow pointed downward. Decrypt is connected with Decryption Key with an arrow pointed downward. Decrypt. The arrow between Encrypt and Decrypt is connected with Eve with an arrow pointed downward.

Long description

The illustration shows a box with three rows. In the rows, the letters are shown with their frequencies.

In the first row, a: .082, b: .015, c: .028, d: .043, e: .127, f: .022, g: .020, h: .061, i: .070, j: .002. In the second row, k: .008, l: .040, m: .024, n: .067, o: .075, p: .019, q: .001, r: .060, s: .063, t: .091.

In the third row, u: .028, v: .010, w: .023, x: .001, y: .020, z: .001.

Long description

The illustration shows a box. In the box, the letters are shown with their frequencies, e: .127, t: .091, a: .082, o: .075, i: .070, n: .067, s: .063, h: .061, r: .060.

Long description

The table shows counting diagrams with 9 rows labeled as W, B, R, S, I, V, A, P, N and 9 columns labeled as W, B, R, S, I, V, A, P, N. The table shows various corresponding values for each row and each column.

Long description

The table shows 5 versus 5 matrix table with row labeled as A, D, F, G, X and columns labeled as A, D, F, G, X. The table shows various corresponding values for each column and each row.

Long description

The table shows a matrix table with 5 columns labeled as R, H, E, I, N. The table shows various corresponding values for each column.

Long description

The table shows a matrix table with 5 columns labeled as E, H, I, N, R. The table shows various corresponding values for each column.

Long description

The illustration shows four rectangular boxes labeled as R, L, M, N respectively placed vertically parallel to each other. The illustration shows a square box labeled as S beyond the four boxes. S is connected to N, N is connected to M, M is connected to L, L is connected to R with an arrow and vice versa. Beyond S, a rectangular box labeled as K, keyboard is connected to S with an arrow at the very lower position and a bar with four bulbs labeled as glow lamps is connected to S with an arrow at the very upper position.

Long description

A table shows 6 rows and 6 columns for addition mod 6.
The table contains various values for each rows and
columns.

Long description

A table shows 6 rows and 6 columns for multiplication mod 6. The table contains various values for each rows and columns.

Long description

The table shows various symbols and their decimal and binary number in five numbers of rows. The first row contains the symbol exclamation : 33 : 0100001, double inverted : 34 : 0100010, hash : 35 : 0100011, Dollar : 36 : 0100100, percentage : 37 : 0100101, and : 38 : 0100110, single inverted : 39 : 0100111. The second row contains the symbols open bracket : 40 : 0101000, closed bracket : 41 : 0101001, asterisk : 42 : 0101010, plus : 43 : 0101011, coma : 44 : 0101100, minus : 45 : 0101101, dot : 46 : 0101110, slash: 47 : 0101111. The third row contains 0 : 48 : 0110000, 1 : 49 : 011000, 2 : 50 : 0110010, 3 : 51 : 0110011, 4 : 52 : 0110100, 5 : 53 : 0110101, 6 : 54 : 011010, 7 : 55 : 0110111. The fourth row contains 8 : 56 : 0111000, 9 : 57 : 0111001, colon : 58 : 0111010, semi colon : 59 : 0111011, less than : 60 : 0111100, equal : 61 : 0111101, greater than : 62 : 0111110, question mark : 63 : 0111111. The last row contains the symbols at the rate : 64 : 10000000, A : 65 : 1000001, B : 66 : 1000010, C : 67 : 1000011, D : 68 : 1000100, E : 69 : 1000100, F : 70 : 1000110, G : 71 : 1000111.

Long description

The block diagram of a stream cipher encryption shows an arrow pointed to x_{n+2} , x_{n+2} is connected to x_{n+1} with an arrow, and then x_{n+1} is connected to x_n with an arrow which further connects to an adder.

Then, an arrow points towards p_{n+2} which is connected to p_{n+1} with an arrow, and then p_{n+1} points to p_n with an arrow which further connects to an adder. The output of the adder is fed to c_n , from which an output is obtained.

Long description

The image of a linear feedback shift register satisfying $x_{n+3} = x_{n+1} + x_n$ shows that x_{n+2} is connected to x_{n+1} with an arrow which is further connected to x_n with an arrow. x_n and plaintext are fed to another adder from which the output ciphertext is obtained.

Long description

The first block contains P_1 which represents the first plain text which is then connected to block E_K which represents encryption function. Block C_0 which represents initialization vector is connected to block P_1 and E_K through a connector. Block E_K is then connected to block C_1 which represents the first ciphertext. Block that contains P_2 which represents second plain text is connected to E_K through a connector. Block C_1 is connected to the connector that connects block P_2 and block E_K . Again, the block E_K connects the block C_2 which represents the second ciphertext from where the arrow connects to dot-dot-dot.

Long description

The illustration shows eleven blocks connected to each other. The first block labeled as $X_{\text{subscript } 1}$ connects to the second block $E_{\text{subscript }} \dots$ which again, connects to the block that contains $O_{\text{subscript } 1}$ in a register that denotes 8 leftmost bits that connect with a connector $P_{\text{subscript } 1}$ that shows 8 bits and $C_{\text{subscript } 1}$ is connected to the connector. Again, $C_{\text{subscript } 1}$ is connected to register that contains $C_{\text{subscript } 1}$ present in rightmost side of the block $X_{\text{subscript } 2}$ that connects the block $X_{\text{subscript } 2}$ and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block $E_{\text{subscript } K}$ which again connects to the block that contains $O_{\text{subscript } 2}$ in a register that denotes 8 leftmost bits that connects with a connector $P_{\text{subscript } 2}$ that shows 8 bits and $C_{\text{subscript } 2}$ is connected to the connector. Again $C_{\text{subscript } 2}$ is connected to register that contains $C_{\text{subscript } 2}$ present in rightmost side of the block $X_{\text{subscript } 2}$ that connects the block $X_{\text{subscript } 3}$ and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block $E_{\text{subscript } K}$ which again connects to the block that contains $O_{\text{subscript } 3}$ in a register that denotes 8 leftmost bits that connects with a connector $P_{\text{subscript } 2}$ that shows 8 bits and $C_{\text{subscript } 3}$ is connected to the connector which is further connected to dot-dot-dot. The first block $X_{\text{subscript } 1}$ is connected to the next $X_{\text{subscript } 1}$ block that contains $O_{\text{subscript } 1}$ and $X_{\text{subscript } 2}$ block is connected to the next $X_{\text{subscript } 2}$ block that contains $O_{\text{subscript } 2}$.

Long description

The illustration shows eleven blocks connected to each other. The first block labeled as X_{1} connects to the second block E_{1} which again connects to the block that contains O_{1} in a register that denotes 8 leftmost bits that connect with a connector P_{1} that shows 8 bits and C_{1} is connected to the connector. Again, O_{1} is connected to register that contains O_{1} present in rightmost side of the block X_{1} that connects the block X_{2} and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block E_{2} which again connects to the block that contains O_{2} in a register that denotes 8 leftmost bits that connects with a connector P_{2} that shows 8 bits and C_{2} is connected to the connector. Again O_{2} is connected to register that contains O_{2} present in rightmost side of the block X_{2} that connects the block X_{3} and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block E_{3} which again connects to the block that contains O_{3} in a register that denotes 8 leftmost bits that connects with a connector P_{3} that shows 8 bits and C_{3} is connected to the connector which is further connected to dot-dot-dot. The first block X_{1} is connected to the next X_{1} block that contains O_{1} and X_{2} block is connected to the next X_{2} block that contains O_{2} .

Long description

The illustration shows nine blocks that are separated. The first block labeled as X subscript 1 connects to the second block E subscript which again connects to the block that contains O subscript 1 in a register that denotes 8 leftmost bits that connect with a connector P subscript 1 that shows 8 bits and C subscript 1 is connected to the connector. Block X subscript 2 equals X subscript 1 plus 1 connects to the block E subscript K which again connects to the block that contains O subscript 2 in a register that denotes 8 leftmost bits that connect with a connector P subscript 2 that shows 8 bits and C subscript 2 is connected to the connector. Block X subscript 3 equals X subscript 2 plus 1 connects to the block E subscript K which again connects to the block that contains O subscript 3 in a register that denotes 8 leftmost bits that connect with a connector P subscript 2 that shows 8 bits and C subscript 3 is connected to the connector which is further connected to dot-dot-dot.

Long description

The illustration shows two inputs K_i and R_{i-1} fed to a feedback which is labeled as f . Output of feedback is fed to an adder which is denoted as XOR. A output R_i is obtain from the adder. Another input L_{i-1} is fed to the adder. An arrow is shown from R_{i-1} which is labeled as L_i .

Long description

The illustration shows six numbers, 1, 2, 3, 4, 5, 6 on the upper row and eight numbers 1, 2, 4, 3, 4, 3, 5, 6 on lower row respectively. Several arrows are shown between 1 to 1, 2 to 2, 4 to 4, 3 to 3, 4 to 4, 3 to 3, 5 to 5, 6 to 6 from upper row to bottom row.

Long description

Flow chart shows an input $R_i - l$ which is fed to $E(R_i - l)$. Output of $E(R_i - l)$ is fed to an adder. Adder output is subdivided into two parts 4 bits and 4 bits. Output of 4 bits and 4 bits are connected to S_1 and S_2 respectively. Output of S_1 and S_2 is connected to $f(R_i - l, K_i)$. On the right side, another input is fed to an adder which is labeled as K_i .

Long description

Table shows two columns, first four bits along with their frequencies, the values are labeled as 0000: 12, 1000: 33, 0001: 7, 1001: 40, 0010: 8, 1010: 35, 0011: 15, 1011: 35, 0100: 4, 1100: 59, 0101: 3, 1101: 32, 0110: 4, 1110: 28, 0111: 6 and 1111: 39.

Another table shows two columns, last four bits along with their frequencies, the values are labeled as 0000: 14, 1000: 8, 0001: 6, 1001: 16, 0010: 42, 1010: 8, 0011: 10, 1011: 18, 0100: 27, 1100: 8, 0101: 10, 1101: 23, 0110: 8, 1110: 6, 0111: 11, and 1111: 17.

Long description

An input Plaintext is fed to IP. The output of IP is subdivided in two parts L subscript 0 and R subscript 0 respectively. The output of L subscript 0 is connected to an adder whereas the output of R subscript 0 is connected to the adder through a feedback f, where an input K subscript 1 is fed. Output of the adder is connected to R subscript 1 and same of R subscript 0 is connected to L subscript 1. The output of L subscript 1 is connected to an adder whereas the output of R subscript 1 is connected to the adder through a feedback f, where an input K subscript 2 is fed. Output of the adder is connected to R subscript 2 and same of R subscript 1 is connected to L subscript 2. L subscript 2 and R subscript 2 are further connected to L subscript 15 and R subscript 15 respectively. The output of L subscript 15 is connected to an adder whereas the output of R subscript 15 is connected to the adder through a feedback f, where an input K subscript 16 is fed. Output of the adder is connected to R subscript 16 and same of R subscript 15 is connected to L subscript 16. The outputs obtained from R subscript 16 and L subscript 16 are fed to IP superscript negative 1 which is further fed to ciphertext.

Long description

The values are labeled as follows:

Row 1: 58 50 42 34 26 18 10 2 60 52 44 36 28 20 12 4

Row 2: 62 54 46 38 30 22 14 6 64 56 48 40 32 24 16 8

Row 3: 57 49 41 33 25 17 9 1 59 51 43 35 27 19 11 3

Row 4: 61 53 45 37 29 21 13 5 63 55 47 39 31 23 15 7

Long description

The values are labeled as follows:

Row 1: 32 1 2 3 4 5 4 5 6 7 8 9

Row 2: 8 9 10 11 12 13 12 13 14 15 16 17

Row 3: 16 17 18 19 20 21 20 21 22 23 24 25

Row 4: 24 25 26 27 28 29 28 29 30 31 32 1

Long description

The values are labeled as follows:

Row 1: 16 7 20 21 29 12 28 17 1 15 23 26 5 18 31 10

Row 2: 2 8 24 14 32 27 3 9 19 13 30 6 22 11 4 25

Long description

A flow chart starts with an input $R_i - 1$ which is fed to Expander. The output of expander is fed to $E(R_i - 1)$. The output of $E(R_i - 1)$ and another input K_i are fed to an adder. The output of the adder is subdivided into eight 6 bits data which are labeled as $B_1, B_2, B_3, B_4, B_5, B_6, B_7$ and B_8 respectively. The outputs of $B_1, B_2, B_3, B_4, B_5, B_6, B_7$ and B_8 respectively are fed to $S_1, S_2, S_3, S_4, S_5, S_6, S_7$ and S_8 respectively. Again the outputs of $S_1, S_2, S_3, S_4, S_5, S_6, S_7$ and S_8 respectively are fed to eight 4 bits data which are labeled as $C_1, C_2, C_3, C_4, C_5, C_6, C_7$ and C_8 respectively. All the outputs are fed to permutation. The output of permutation is fed to $f(R_i - 1, K_i)$.

Long description

The values are labeled as follows:

Row 1: 57 49 41 33 25 17 9 1 58 50 42 34 26 18

Row 2: 10 2 59 51 43 35 27 19 11 3 60 52 44 36

Row 3: 63 55 47 39 31 23 15 7 62 54 46 38 30 22

Row 4: 14 6 61 53 45 37 29 21 13 5 28 20 12 4

Long description

The values are labeled as follows:

Round: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Shift: 1 1 2 2 2 2 2 1 2 2 2 2 2 2 1

Long description

A table shows information of 48 bits chosen from the 56-bit string C_i D_i and the output are K_i. The table shows four rows and twelve columns.

Row 1: 14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21 and 10.

Row 2: 23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13 and 2.

Row 3: 41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33 and 48.

Row 4: 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29 and 32.

Long description

A table shows information of S-Boxes. The table divided into eight parts depicted as S-box 1, S-box 2, S-box 3, S-box 4, S-box 5, S-box 6, S-box 7 and S-box 8 where each table shows four rows and 16 columns with several numbers.

Long description

A box labeled as plaintext is linked with the box AddRoundKey, The box AddRoundKey is labeled as W left parenthesis 0 right parenthesis, W left parenthesis 1 right parenthesis, W left parenthesis 2 right parenthesis, W left parenthesis 3 right parenthesis with a left inward arrow. Below the box AddRoundKey, a block with four boxes labeled as Round 1. In the block the first box is SubBytes which is linked with the below box labeled as ShiftRows. Again the ShiftRows is linked with MixColumns and lastly the MixColumns box is linked with AddRoundKey which is labeled as W left parenthesis 4 right parenthesis, W left parenthesis 5 right parenthesis, W left parenthesis 6 right parenthesis, W left parenthesis 7 right parenthesis with a left inward arrow. Again a block with four boxes labeled as Round 9, starting with the first box labeled as SubBytes which is linked with ShiftRows. Again ShiftRows is linked with MixColumns and lastly the MixColumns box is linked with AddRound Key which is labeled as W left parenthesis 36 right parenthesis, W left parenthesis 37 right parenthesis, W left parenthesis 38 right parenthesis, W left parenthesis 39 right parenthesis with a left inward arrow. Again a block with three boxes labeled as Round 10, starting with the first box labeled as SubBytes which is linked with ShiftRows. Again ShiftRows is linked with AddRound Key which is labeled as W left parenthesis 40 right parenthesis, W left parenthesis 41 right parenthesis, W left parenthesis 42 right parenthesis, W left parenthesis 43 right parenthesis with a left inward arrow. Below the block, a box is linked labeled as Ciphertext.

Long description

A table shows the information of Rijndael Encryption.

The data listed for:

Line 1: ARK, using the 0th round key.

Line 2: Nine rounds of SB, SR, MC, ARK, using round keys 1 to 9.

Line 3: A final round: SB, SR, ARK, using the 10th round key.

Long description

A table shows S-Box for Rijndael that shows various numerical values. The values are listed as

99 124 119 123 242 107 111 197 48 1 103 43 254 215 171
118

202 130 201 125 250 89 71 240 173 212 162 175 156 164
114 192

183 253 147 38 54 63 247 204 52 165 229 241 113 216 49
21

4 199 35 195 24 150 5 154 7 18 128 226 235 39 178 117

9 131 44 26 27 110 90 160 82 59 214 179 41 227 47 132

83 209 0 237 32 252 177 91 106 203 190 57 74 76 88 207

208 239 170 251 67 77 51 133 69 249 2 127 80 60 159 168

81 163 64 143 146 157 56 245 188 182 218 33 16 255 243
210

205 12 19 236 95 151 68 23 196 167 126 61 100 93 25 115

96 129 79 220 34 42 144 136 70 238 184 20 222 94 11 219

224 50 58 10 73 6 36 92 194 211 172 98 145 149 228 121

231 200 55 109 141 213 78 169 108 86 244 234 101 122
174 8

186 120 37 46 28 166 180 198 232 221 116 31 75 189 139
138

112 62 181 102 72 3 246 14 97 53 87 185 134 193 29 158

225 248 152 17 105 217 142 148 155 30 135 233 206 85 40
223

140 161 137 13 191 230 66 104 65 153 45 15 176 84 187 22

Long description

A table shows the information of Rijndael Decryption.

The data listed for

line 1: ARK, using the 10th round key.

Line 2: Nine rounds of ISB, ISR, IMC, IARK, using round keys 9 to 1.

Line 3: A final round: ISB, ISR, ARK, using the 0th round key.

Long description

A table shows information of The RSA Algorithm. The data listed for

Line 1: Bob chooses secret primes p and q and computes n equal pq.

Line 2: Bob chooses e with gcd left parenthesis e comma left parenthesis p minus 1 right parenthesis left parenthesis q minus 1 double right parenthesis equals 1.

Line 3: Bob computers d with de-triple equal 1 left parenthesis mod left parenthesis p minus 1 right parenthesis left parenthesis q minus 1 double right parenthesis.

Line 4: Bob makes n and e public comma and keeps p comma q comma d secret.

Line 5: Alice encrypts m as c triple equal $m^e \pmod{n}$ and sends c to Bob.

Line 6: Bob decrypts by computing $m \equiv c^d \pmod{n}$.

Long description

A table shows information of RSA Encryption Exponents. The data listed for e and percentage as follows:

Column 1: 65537, 17, 41, 3, 19, 25, 5, 7, 11, 257 and others.

Column 2: 95.4933 percent, 3.1035 percent, 0.4574 percent, 0.3578 percent, 0.1506 percent, 0.1339 percent, 0.1111 percent, 0.0596 percent, 0.0313 percent, 0.0241 percent and 0.0774 percent.

Long description

A table shows information of Factorization Records. The data listed for year and number of digits as follows:

Column 1: 1964, 1974, 1984, 1994, 1999, 2003, 2005 and 2009.

Column 2: 20, 45, 71, 129, 155, 174, 200 and 232.

Long description

The long message is an arbitrary message like
01101001... which pass through the hash function and
produces the output message digest of fixed length 256
bit- 11...10.

Long description

Initially, a value IV is fed as input in the first block f. A two-bit string M sub 0 is first fed into the block. The blocks are then fed one-by-one into f. The message M goes from M sub 0 to up to M sub left parenthesis n-1right parenthesis. The final output is the hash value H left parenthesis M right parenthesis.

Long description

r is taken as the rate and c is taken as the capacity. First a block o o is introduced. First o is the rate r and second o is the capacity c. Rate r occupies maximum part of the block. Then the message signal M subscript o is transferred to block f from where the message is transferred back to rate and capacity block and so on till message signal M subscript n minus 1. This is the absorbing part. The diagram is then separated by a dotted line after which the squeezing part starts. Here the outputs Z subscript o and so on are squeezed out of the rate and capacity and f blocks.

Long description

An illustration shows three blocks labeled as Data Record $k - 1$, Data Record k , and Data Record $k + 1$ and each block contains a data block, h left parenthesis right parenthesis, and a pointer. A vertical box with h left parenthesis right parenthesis and a pointer is shown at the upper top right. The vertical box is linked with the third block with the help of an arrow. Again the third block is linked with the second box and also the second box is linked with the first block with the help of an arrow.

Long description

An illustration shows a tree diagram. The first block of the tree diagram is $h \text{ left parenthesis right parenthesis}$. The first block is linked with the second block with a downward arrow and the second block has $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis}$. The second block is divided into two child blocks labeled as $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis right parenthesis}$ and $h \text{ left parenthesis right parenthesis right parenthesis }$. Again the block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis}$ is subdivided into two child blocks labeled as $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis right parenthesis}$ and $h \text{ left parenthesis right parenthesis right parenthesis right parenthesis}$. Also, the block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis}$ is subdivided into two child blocks labeled as $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis right parenthesis}$ and $h \text{ left parenthesis right parenthesis right parenthesis right parenthesis}$. The block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis}$ is subdivided into two child blocks labeled as R_0 and R_1 . The block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis right parenthesis}$ is subdivided into two child blocks labeled as R_2 and R_3 . The block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis}$ is subdivided into two child blocks labeled as R_4 and R_5 . The block $h \text{ left parenthesis right parenthesis } h \text{ left parenthesis right parenthesis right parenthesis}$ is subdivided into two child blocks labeled as R_6 and R_7 .

Long description

The illustration shows a basic Kerberos model with the participants Cliff, Serge, Trent and Grant, where Cliff is at the center which is linked to Trent, Grant and Serge. Cliff sends a message to Trent indicated by an arrow numbered 1. Trent replies the message to Cliff again indicated by an arrow numbered 2. Cliff sends a message to Grant indicated by an arrow numbered 3. Grant replies the message to Cliff again indicated by an arrow numbered 4. Cliff sends a final message to Serge indicated by an arrow numbered 5.

Long description

The illustration shows a block diagram of a Certification Hierarchy. The block “CA” abbreviated for Certification Authority, is characterized as Client, Client and “RA” abbreviated for Registration Authorities. The block “RA” is again sub-divided into three Clients.

Long description

The screenshot shows a CA's Certificate; General.

Line 1: This certificate has been verified for the following uses, colon

Line 2: Email Signer Certificate

Line 3: Email Recipient Certificate

Line 4: Status Responder Certificate

Line 5: Issued to, colon

Line 6: Organization left parenthesis O right parenthesis, colon, VeriSign, Inc

Line 7: Organizational Unit left parenthesis OU right parenthesis, colon, Class 1 Public Primary Certification Authority dash G2

Line 8: Serial Number

Line 9: Issued by, colon

Line 10: Organization left parenthesis O right parenthesis, colon, VeriSign, Inc

Line 11: Organizational Unit left parenthesis OU right parenthesis, colon, Class 1 Public Primary Certification Authority dash G2

Line 12: Validity, colon

Line 13: Issues on, colon, 05 forward slash 17 forward slash 98

Line 14: Expires on, colon, 05 forward slash 18 forward
slash 98

Line 15: Fingerprints, colon

Line 16: SHA1 Fingerprint, colon

Line 17: MD5 Fingerprint, colon

Long description

The screenshot shows a CA's Certificate; Details.

Line 1: Certificate Hierarchy

Line 2: Verisign Class 1 Public Primary Certification
Authority dash G2

Line 3: Certificate Fields

Line 4: Verisign Class 1 Public Primary Certification
Authority dash G2

Line 5: Certificate

Line 6: Version, colon, Version 1

Line 7: Serial Number, colon

Line 8: Certificate Signature Algorithms, colon, PKCS
hashtag 1 SHA dash 1 With RSA Encryption

Line 9: Issuer, colon, OU equals VeriSign Trust Network

Line 14: Validity

Line 15: Not before, colon, 05 forward slash 17 forward
slash 98

Line 16: Not after, colon, 05 forward slash 18 forward
slash 98

Line 17: Subject, colon, OU equals VeriSign Trust
Network

Line 22: Subject Public Key Info, colon, PKCS hashtag 1
RSA Encryption

Line 23: Subject's Public Key, colon

Line 24: A table of 9 rows and 16 columns is shown

Line 25: Certificate Signature Algorithm, colon, PKCS
hashtag 1 SHA dash 1 With RSA Encryption

Line 26: Certificate Signature Name, colon

Line 27: A table of 8 rows and 16 columns is shown.

Long description

The screenshot shows a Clients Certificate.

Line 1: Certificate Hierarchy

Line 2: Verisign Class 3 Public Primary CA

Line 3: Certificate Fields

Line 4: Verisign Class 3 Public Primary Certification Authority

Line 5: Certificate

Line 6: Version, colon, Version 3

Line 7: Serial Number, colon

Line 8: Certificate Signature Algorithms, colon, md5RSA

Line 9: Issuer, colon, OU equals www.verisign.com forward slash CPS Incorp.

Line 14: Validity

Line 15: Not before, colon, Sunday, September 21, 2003

Line 16: Not after, colon, Wednesday, September 21, 2005

Line 17: Subject, colon, CN equals online.wellsfargo.com

Line 22: Subject Public Key Info, colon, PKCS hashtag 1 RSA Encryption

Line 23: Subject's Public Key, colon, 30 81 89 02 81 81 00 a9

Line 24: Basic Constraints, colon, Subject Type equals
End Entity, Path Length Constraint equals None

Line 25: Subject's Key Usage, colon, Digital Signature,
Key Encipherment left parenthesis AO right parenthesis

Line 26: CRL Distribution Points, colon

Line 27: Certificate Signature Algorithm, colon, MD5
With RSA Encryption

Line 28: Certificate Signature Value, colon

Long description

A table represents Block ID colon 76 Create Coins. The table contains 4 rows and 3 columns. The Block ID colon 76 Create Coins is divided into three columns labeled as Trans ID with values 0, 1 and 2, Value with values 5, 2 and 10 and Recipient with values PK subscript BB, PK subscript Alice and PK subscript Bob.

Long description

A table represents Block ID colon 77 Consume Coins colon 41 left parentheses 2 right parentheses, 16 left parentheses 0 right parentheses, 31 left parentheses 1 right parentheses, Create Coins. The table contains 6 rows and 3 columns. The Block ID colon 77 Consume Coins colon 41 left parentheses 2 right parentheses, 16 left parentheses 0 right parentheses, 31 left parentheses 1 right parentheses, Create Coins is divided into three columns labeled as Trans ID with values 0, 1, 2 and 3, Value with values 15, 5, 4 and 11 and Recipient with values PK subscript Sarah Store, PK subscript Alice, PK subscript Bob and PK subscript Charles.

Long description

The block diagram of Bit coin's block chain and a Merkle tree of transactions three blocks in adjacent position.

Each block has four lines of codes. The third block connects to the second and the second block connects to the first.

Line 1: prev underscore hash tag, colon, h left parenthesis right parenthesis

Line 2: timestamp

Line 3: merkleroot, colon, h left parenthesis right parenthesis

Line 4: nonce

Another block is placed just below the second block, which is comprised of one sub block, with a line: h left parenthesis right parenthesis, indented, h left parenthesis right parenthesis. Below the first, there are two other sub-blocks with lines in each block as: h left parenthesis right parenthesis, indented, h left parenthesis right parenthesis, connected with two arrows with the first block. The above two sub-blocks are connected to four sub-blocks represented as TX.

Long description

The box is having a uniform thickness highlighted with black. Another solid rectangle is present inside the tunnel and below it a door is made.

Long description

Illustration shows the Tunnel Used in the Zero-Knowledge Protocol. The tunnel is represented by a rectangular box with an opening at the top. Inside it, another tunnel is made by rectangular box with an opening at the top. Both the boxes are having a uniform thickness highlighted with black. Another solid rectangle is present inside the tunnel and a Central Chamber is presented below it.

Long description

A schematic diagram of Huffman Encoding, showing outputs a, b, c and d, arranged vertically. Output c corresponding to 0.1 and d corresponding to 0.1 gives 0.2, with two choices 0 and 1. Again, 0.2 and b corresponding 0.3 gives 0.5, with two choices 0 and 1. Further, 0.5 and a corresponding 0.5 gives 1, with two choices 0 and 1.

Long description

The illustration shows Shannon's Experiment on the Entropy of English. There are six sentences “there is no reverse”, “on a motorcycle a”, “friend of mine found”, “this out rather”, “dramatically the” and “other day”. Below each letter, information obtained is displayed.

Long description

The x-axis ranges from negative 1 to 3, in increments of 1 and the y-axis ranges from negative 4 to 4, increments of 2. The first curve begins at the first quadrant, passes decreasing through point 1 of the x-axis, then ends decreasing at the fourth quadrant making a peak point at point 1 of the x-axis.

Long description

The x-axis ranges from negative 4 to 8, in increments of 2 and the y-axis ranges from negative 20 to 20, in increments of 10. The curve starts from the first quadrant passing through the point 10 of the y-axis, then passes through the second quadrant decreasing through the point negative 4 of the x-axis intersecting the point negative 10 and ends decreasing at the fourth quadrant.

Long description

The graph shows a curve, a straight line and another doted straight line that is parallel to the y-axis and it is plotted in the first and second quadrant. The curve begins at the first quadrant passes decreasing through the y-axis, enters the second quadrant passes through the x-axis, enters the third quadrant passes through the y-axis and ends decreasing at the fourth quadrant.

Long description

The illustration shows a straight line segment and two vectors. The first vector labeled as v_1 lies along the line segment pointing toward the right direction. The second vector labeled as v_2 starts from the beginning point of the first vector, pointing upwards and slightly deflected towards right from the angle 90 degrees. The second line lies between two dotted lines perpendicular to the line segment.

Long description

The illustration shows a straight line segment and two vectors. The first vector labeled as v_1 lies along the line segment pointing toward the right direction. The second vector labeled as v_2 starts from the beginning point of the first vector, pointing upwards and deflexed at angle 60 degrees towards the right. The second vector lies between two dotted lines perpendicular to the line segment and the vector crosses the perpendicular line at the very right.

Long description

The illustration shows an input message which is fed to the encoder, encoder output is fed into the noisy channel through codewords. Noisy channel output is fed to decoder and decoder output is fed to message.

Long description

The x-axis ranges from 0.1 to 0.5, in increments of 0.1 and the y-axis labeled as code rate ranges from 0.2 to 1, in increments of 0.2. The graph shows a non-linear decreasing curve starting from point 1 of the y-axis and ends at point 0.5 on the x-axis.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical black edge is shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical edge labeled as Polaroid C and a black vertical edge is are shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical edge labeled as Polaroid B is shown.

The Polaroid B is inclined with a gray rectangular bar with a vertical bar labeled as Polaroid C which is inclined with a grey rectangular bar with a vertical black edge is shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The image shows a graph of y versus x. The x axis ranges from 0 to 7, in increments of 1 and the y-axis ranges from 0 to 1, in increments of 0.2. Following coordinates (0, 0.15), (1, 0.15), (2, 0.35), (3, 0.85), (4, 0.35), (5, 0.85), (6, 0.35) and (7, 0.15) are marked.

Long description

The image shows a graph of y versus x. The x axis ranges from 0 to 14, in increments of 2 and the y-axis ranges from 0 to 1, in increments of 0.2. Following coordinates (0, 1), (1, 0.2), (2, 0.3), (3, 0.9), (4, 0), (5, 0.2), (6, 0.65), (7, 0.3), (8, 0), (9, 0.3), (10, 0.65), (11, 0.2), (12, 0), (13, 0.9), (14, 0.25) and (15, 0.2) are marked.

Long description

The image shows a graph of y versus x . The x axis ranges from 0 to 500, in increments of 100 and the y -axis ranges from 0 to 0.2, in increments of 0.05. The graph starts from y equals 0.01, is comprised of four peaks with an altitude of y equals 0.02. The graph is comprised of numerous dots, whose density decreases towards the peaks.

Long description

A graph ranges negative 4 to 4 in increment of 2 in vertical axis and negative 1 to 3 in increment of 1 in horizontal axis. The graph plots an oval shaped curve at point o on vertical axis. Another elliptic curve is plotted that starts from 3 on horizontal axis, vertex lies at $(1, 0)$ and extends to $(3, 5)$.

Long description

A y versus x graph ranges from negative 4 to 4 in increment of 2 in y axis and negative 1 to 3 in increment of 1 in x axis. The graph plots an oval shaped curve whose vertex lies at the origin and (-1, 0). Another elliptic curve is plotted which is symmetric about the x axis and vertex lies at (1, 0).

Long description

A graph ranges negative 5 to 5 in increment of 1 in vertical axis and negative 1 to 3 in increment of 0.5 in horizontal axis. The graph plots an oval shaped curve at point 0 on vertical axis. Another elliptic curve is plotted that starts from 3 on horizontal axis, vertex lies at (1, 0) and extends to (3, 5). An equation is shown above the graph depicting $y^2 - x \times (x - 1) \times (x + 1) = 0$.