3. Codes and Cryptos

In many contexts, especially where computers are involved, it's necessary to store and transmit information. Then a couple of problems arise: you have to choose a format to use when storing the information (computers only speak zeros and ones, a language you don't master yourself). You have to *code* your material in some way. An important aspect is that the coding mustn't be too prone to errors – it's a good thing if you are able to repair at least minor damages to the material. Furthermore, some material is secret; you don't want just anyone to be able to read it. In that case you have to *encrypt* it, in some way or another.

Both these problems, coding and encryption, are possible to tackle using mathematical methods – and on top of it using stuff that was earlier believed to be of purely theoretical interest, like group theory and the theory of prime numbers. Here we are going to study some versions of coding and encryption that show how theory and practice can be elegantly merged into something really useful. (Additionally, much of the history is exciting. We recommend for instance the book *Alan Turing: the Enigma* by Andrew Hodges, where you among other things can read about how the allied powers broke the crypto system used by the German forces.)

Highlights from this chapter

- A binary *error-correcting code* with the *distance* δ is a collection of code words with the property that every code word differs from every other code word in at least δ places. If a code word is corrupted so that less than $\delta/2$ bits are affected, the error can be corrected using the *nearest neighbour principle.*
- A *linear code* is a code where the sum of any two code words is a code word as well. Linear codes can be created using algebra. A binary linear code is always made up of 2^k code words for some integer k, which is called the *dimension* of the code. You can check whether a given word is a correct code word by multiplying it by the *check matrix* of the code.
- *Encryption* means making a message unreadable to outsiders. Authorised readers have to know how the message is to be *decrypted*, that is, made readable again. The tool used for encyption and decryption is called a *key.*
- Two in practical applications common methods for encryption are *RSA* and *DES.*

3.1 Encoding

What exactly is meant by **encoding**? The word may give you associations to stories about spies, but the technical definition is not at all that secretive. Encoding concerns packaging a message in a format that can be transmitted and unpacked by a recipient. This text is a code, where thoughts have been turned into words, which have been turned into letters, which are put back into words by the reader, and hopefully made into thoughts. On the way, it has made some stops in the computer, where it has been coded as sequences of zeros and ones. In a first stage, these have told what letters there should be; later the material has been recoded into a description of where the printer's ink should be placed on the paper.

Of primary interest when using codes is that the recipient should get the same message as the one that was sent. Unfortunately, there can occur trouble during transmission. In the following section, we will look at some kinds of errors that can occur, how you can detect them, and how, if possible, they can be corrected so that the message sent is recreated.

3.1.1 Error-Correcting Codes

In this text, we will focus on **binary codes**, where the **code words** are binary strings. Furthermore, we add the restriction that all the code words should have the same length, which is simply called the *length of the code.*

Exercise 3.1

(a) Do you know any binary code that isn't made up according to the principle above?

(b) Do you know any binary code that *is* made up according to the principle above?

Exercise 3.2 Assume that you have n different messages that you want to be able to transmit. How many **bits** (that is, binary digits) do you at least need in the code words?

When transmitting coded messages there can be **transmission errors**, where one or more components of the message are corrupted into something else.

Depending on the transmission channel, the errors may either occur singly or in bursts. If they occur just once in a while, you can assume that it's relatively unlikely that a short message will be the subject of more than one error, and that is the situation that we will study from now on. (**Burst errors** – such as lighting striking the hard drive – we will refrain from trying to analyse.)

Exercise 3.3 We are sending a message consisting of ten bits. The probability for a bit to be corrupted during transmission is 1 % for each bit, and the probabilities are independent.

(a) How great is the probability that the message will be transmitted without any errors?

(b) How great is the probability that the message will be transmitted with exactly one error?

(c) How great is the probability that we'll get exactly two errors?

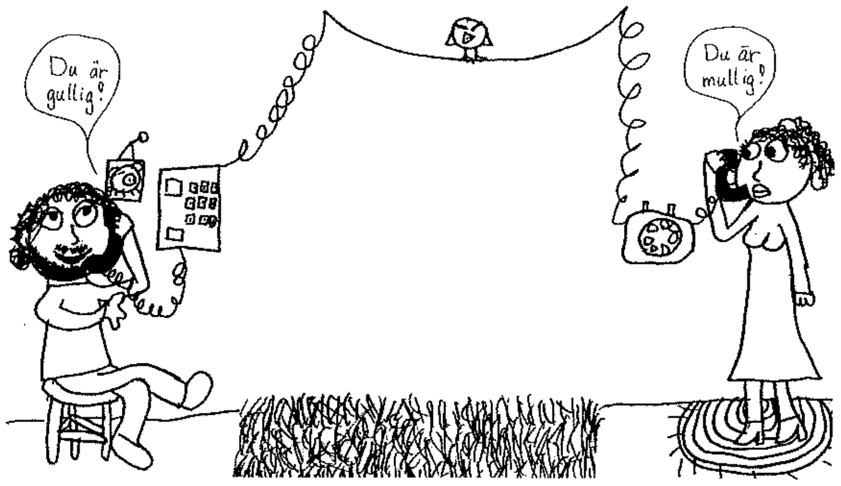

Figure 3.1: *Transmisssion error. (The message "you are cute" is turned into "you are plump".)*

Now let's assume that we have got a code consisting of the words {00, 01, 10, 11} and that we are affected by a transmission error when sending a message to the recipient. A transmission error means that a digit that was supposed to be one is turned into zero, or vice versa. Are we able to detect this? No, since if for instance the code word 10 during transmission is changed to 11 this is a completely valid code word, but with a different meaning. The problem seems to be that the difference between the code words is too small! But how do you measure differences between code words?

Definition 3.1: ***Distance*** By the **distance** between two code words is meant the number of places where they differ. The distance between the two code words $\mathbf{x}_1$ and $\mathbf{x}_2$ we denote by $\delta(\mathbf{x}_1, \mathbf{x}_2)$. By the **minimum distance** (or just distance) δ of a code C is meant the shortest distance between two words in the code,

$$\delta = \min\{\delta(\mathbf{x}_1, \mathbf{x}_2) \mid \mathbf{x}_1, \mathbf{x}_2 \in C\} \qquad \blacksquare$$

In the code above $\delta(00, 11) = 2$, since they differ at two places. Still, the code has the minimum distance 1, since there are code words (such as 00 and 01) that only differ at one place.

Exercise 3.4 What would it mean if a code had the minimum distance zero?

Exercise 3.5 Determine the minimum distance of this code:

$$\{010101, 101010, 111000, 000111\}$$

We can divide the problem of error analysis into two subproblems: to *detect* that an error has occured, and to *correct* the error. The following theorem will, hopefully, feel fairly obvious:

Theorem 3.1 If the distance of the code is k you can detect all errors that affect at most $k-1$ bits. $\blacksquare$

If our code consists of the words $\{00, 11\}$ and a 00 that we have sent during transmission is turned into 10 we know that something has gone wrong, since no such message exists. On the other hand, it's not possible to determine what was sent.

Exercise 3.6 Prove theorem 3.1. *

Example 3.1: ***Personal Identity Numbers*** Swedish personal identity numbers consist of ten digits, where the tenth is a check digit. It's calculated from the other digits. The idea is that it's fairly unlikely that a mistyped or invented number will get the check digit right, and that makes it possible to detect many errors (both cased by carelessness and by fraud) that would otherwise have caused trouble. ■

Example 3.2: ***7-bit ASCII*** In the original American character encoding **ASCII** (American Standard Code for Information Interchange) every character was given eight bits of space (called one **byte**). But the Americans didn't have $2^8 = 256$ different characters that they wanted to encode, and settled for 128. That way, the first bit positition (the 128-digit) was left over, and it was frequently used for error control. The most common versions were **odd** and **even parity**, respectively, where it had been decided that all the code words should contain an odd (alternatively even) number of ones. If the word already satified this, the first bit, the **parity bit**, was set to zero, otherwise to one.

Nowadays all the bits are required for information, since we Europeans need to be able to code weird things like åäö as well. Unfortunately, there are a number of parallel standards available. One of the authors of this book was for instance affected by a collision between IBM-standard and Apple-standard which made the computer, without promting, turn a whole chapter from *Discrete Mathematics and Discrete Models* into 7-bit ASCII... ■

Detecting errors is a good thing, especially if you are able to request retransmission of the broken message. But even better is being able to recreate the message.

Assume that we have the code $\{000, 111\}$ and recieve the message 001. If it really is more likely that there is an error at one place than that there are errors at two places, then it's more likely that the message sent was 000, which only differs from the recieved one at one place, than that the message was 111, which differs at two places. Because of this, we can correct the messsage to 000. This way of reasoning is called the **nearest neighbour principle.**

This reasoning gives us another theorem:

Theorem 3.2 If the minimum distance of the code is $2k+1$ we can correctly correct errors affecting up to k bits. ■

Exercise 3.7 Prove the theorem. *

Exercise 3.8 Our code consists of the words $\{111000, 000111, 111111\}$. We have recieved 100110. Which code word is most likely the one sent by the sender?

Exercise 3.9 We have a code consisting of the words $\{00000, 00111, 11100, 11011\}$. Look at all the strings of length 5. Partition them into equivalence classes based on which code word they will be interpreted as according to the nearest neighbour principle.

3.1.2 Linear Codes

If you make up a code by choosing code words in an unsystematic way, you have to compare all the code words to each other to find the minimum distance. If there are n code words in the code, that means $\binom{n}{2}$ tests. In this section we'll see how it's possible, using group theory, to make codes in a systematic way that makes them more easy to analyse.

We are going to calculate with the set of binary strings of length n by adding component by component modulo 2. Then we have an Abelian group, which is denoted $(\mathbb{Z}_2)^n$.

> **Exercise 3.10** Check that this really is an Abelian group! *

> **Exercise 3.11** What do the inverses look like in this group?

Those who have studied linear algebra realise that $(\mathbb{Z}_2)^n$ can be seen as a vector space as well. The dimension of it is n since the smallest set that can be used to generate all the elements in $(\mathbb{Z}_2)^n$ contains n strings. $\{100\ldots0$, $010\ldots0$, $001\ldots0$, $\ldots$, $000\ldots1\}$ is an example of such a **basis** of the space.

> **Exercise 3.12** Check that it's true that it's possible to generate all the strings in $(\mathbb{Z}_2)^n$ using the indicated basis, while it isn't possible if we remove one of the strings from it. *

Now we choose our code as a subgroup of $(\mathbb{Z}_2)^n$. Such a code is called a **linear code**, since it has the group property that the sum of two code words is a code word as well. According to Lagrange's theorem, the number of elements in a subgroup is a divisor of the number of elements in the main group, and thus the number of code words that the code contains has to be a power of two, let's say 2^k words. This subgroup is isomorphic to $(\mathbb{Z}_2)^k$, so we say that the **dimension** of the code is k.

Now we'll see how we in a simple way are able to compute the minimum distance of a binary linear code. In a binary code, the difference between two words is the same thing as the sum. In a linear code, the sum of two code words is a code word as well. That means that the set of code words at the same time is a list of the places where the code words pairwise differ.

Every code word differs at zero places from itself, which corresponds to the neutral element of the group, the word consisting of zeros only. The code word with the least number of zeros, not counting the zero word, tells what the minimum distance between the code words is.

In a linear code we thus just have to count the number of ones in the n code words instead of the number of ones in the $\binom{n}{2}$ differences, and that's a lot less work.

> **Exercise 3.13** How can the set of $\binom{n}{2}$ pairwise differences between words at the same time be a set of n words?

> **Exercise 3.14** Try to find two linear codes with the dimension 1 from $(\mathbb{Z}_2)^4$: one that is as stupid as possible from an error analysis point of view, and one that is as clever as possible.

3.1.3 Design of Linear Codes

In exercise 2.1 in the previous chapter we established that the solution set of a homogeneous system of linear equations is a group. If we solve a system of equations modulo two, the solution set will be a group of binary strings, just like the codes we discussed above. We can thus use linear algebra as a tool to generate a group to use as a code.

Example 3.3 We start with this system of equations:

$$\begin{cases} x_1 + x_2 \qquad\quad + x_4 + x_5 = 0 \\ x_1 \qquad\quad + x_3 \qquad\quad + x_5 = 0 \\ x_1 \qquad\quad + x_3 + x_4 \qquad\quad = 0 \end{cases}$$

or on matrix form

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

x_1 represents the first bit of the code word, and so on. Since the system has five unknowns but only three equations we'll get (at least) two parameters in the solution, since we'll be able to solve the system for three of the unknowns, which will be expressed using the remaining two. We can tidy the system using Gaussian elimination, and then keep in mind that $+1 \equiv -1 \pmod 2$, so we don't have to bother writing down any minus signs. We get

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

which rewritten as a system of equations becomes

$$\begin{cases} x_1 \qquad\quad + x_3 \qquad\quad + x_5 = 0 \\ \qquad x_2 + x_3 \qquad\quad + x_5 = 0 \\ \qquad\qquad\qquad x_4 + x_5 = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} x_1 = x_3 + x_5 \\ x_2 = x_3 + x_5 \\ x_4 = x_5 \end{cases}$$

We have, as stated, been able to express three of the unknowns (one from each equation) using the remaining two: x_3 and x_5. If we assign all possible values to these two parameters we get all the solutions of the system. Since there are only two different values in $\mathbb{Z}_2$, there are in total $2^2 = 4$ combinations of values:

x_3	x_5	x_1	x_2	x_4	code word
0	0	0	0	0	00000
0	1	1	1	1	11011
1	0	1	1	0	11100
1	1	0	0	1	00111

We have aquired a linear code with the dimension 2. The least number of ones in a code word is 3, so the minimum distance of the code is 3. That means that we are able to correct all singe bit errors, and detect all errors affecting two bits. ■

Exercise 3.15 Check the calculations in the example. Do the whole Gaussian elimination by hand, and check that you get the same answer. Check as well that the set of code words obtained really is a group, and that is solves the system of equations. *

Exercise 3.16 Figure out some other method for finding a subgroup of $(\mathbb{Z}_2)^n$.

One point of the particular method for designing codes described here is that correct code words are solutions to a system of equations. If a code word is damaged in transmission it is no longer a solution. To check whether a recieved message is a code word we can thus simply check whether it satisfies the system of equations. That is done by multiplying by the coefficient matrix and check whether the result is a zero vector, because if so it was a correct word (and otherwise not). Because of this, the coefficient matrix is called the **check matrix** of the code.

Exercise 3.17 Determine whether the strings below belong to the code with the check matrix

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(a) 0010101.

(b) 1110111.

Here comes another feature! An incorrect message can be partitioned into the original message and an **error vector**. The error vector has ones in the places that have been corrupted, and zeros in the others. If for instance 101010 is corrupted to 101011 the error vector is 000001, since it's the last bit that is wrong.

Now we'll use some matrix algebra, where we assume that the check matrix is named H, the message $\mathbf{x}$, and the error $\mathbf{e}$:

$$\mathsf{H}(\mathbf{x}+\mathbf{e}) = \mathsf{H}\mathbf{x} + \mathsf{H}\mathbf{e} = \mathbf{0} + \mathsf{H}\mathbf{e} = \mathsf{H}\mathbf{e}$$

The most common error type is the one where just a single bit is wrong. In that case the error vector $\mathbf{e}$ contains just a single one (in the position where the error has occured) and zeros for the rest. If you multiply a vector with a one in position i and zeros for the rest by a matrix, the result will be column number i in the matrix. If $\mathsf{H}\mathbf{e}$ is identical to column i it's thus bit i in the recieved message that has to be changed. (Errors in more than one bit are somewhat more complicated to handle, but this elegant method works in most cases.)

Exercise 3.18 Determine what the faulty code word in the previous exercise ought to have been.

A particularly frequently used check matrix is the one we get by using the binary code for one as column one, the binary code for 2 as column 2, and so on. Then the errors tell directly where they have occured, without any comparisons to the matrix being necessary! A code like this is called a **Hamming code**.

3.1.4 Design of Error-Correcting Linear Codes

Above we have seen what linear codes are. The most interesting parameters we have been discussing are:

- n, the *length* of the code, which tells how many bits there are in each code word
- k, the *dimension* of the code, which tells how many code words we have (namely 2^k)

- δ, the *distance* of the code, which determines how many errors we can correct

How should these parameters be chosen to get a good code? Short code words are from many aspects (particularily storage requirements) to prefere to long ones, but at the same time long code words increase the opportunities to get a large distance. In exercise 3.14 we designed codes with the same n and k but different δ, so the relationship between the parameters is apparently not trivial. Resonable questions to pose are thus firstly which length n that is demanded in theory to achieve a certain distance δ, and secondly how to actually achieve the said distance.

Example 3.4: ***Design of a code with distance $\delta = 3$*** Let's here aim at achieving the distance $\delta = 3$, which according to theorem 3.2 is enough to ensure that it's possible to correct all singel-bit errors. What should the matrix look like if we want this distance?

The minimum distance of the code is equal to the least number of ones occuring in non-zero code words. Because of this, for a start we wish that the code shouldn't contain any words with just a single one and zeros for the rest. If the homogenous system were to have such a solution, what would the matrix then look like? We've just noted that if we multiply the matrix by a vector with a single one we get one of the columns of the matrix as the result. If the vector in question is a solution of the homogenous system the result will consist of zeros only. A matrix that allows for code words with a single one will thus have a column consisting of zeros only!

When we are aiming for a code with the minimum distance 3 we can't allow words with exactly two ones either. A word with two ones can be regarded as the sum of two words with exactly one one each, and if we multiply by the matrix we get

$$\begin{aligned}\mathsf{H}\mathbf{x}_{\text{two ones}} &= \mathsf{H}(\mathbf{x}_{\text{one one}} + \mathbf{x}_{\text{other one}}) \\ &= \mathsf{H}\mathbf{x}_{\text{one one}} + \mathsf{H}\mathbf{x}_{\text{other one}} \\ &= \text{one column} + \text{other column}\end{aligned}$$

If $\mathbf{x}_{\text{two ones}}$ is a solution to the system of equations then

$$\text{one column} + \text{other column} = \mathbf{0},$$

that is, the columns are identical.

A matrix that generates a code with the minimum distance 3 must thus not include any columns with only zeros, nor two columns that are alike. Besides, this is sufficient; if the matrix satisfies these two conditions you get a code with the desired distance. ■

Exercise 3.19 We want to design a check matrix for a code with the minimum distance 3. The matrix is to have four rows. What is the largest possible number of columns we can use?

Now we know how to achieve a certain distance. The second question concerned the relationship between the length and the dimension of the code.

If you solve a homogeneous system of equations with m equations and n unknowns you will get an answer with $n - r$ parameters, where r is the number of linearly independent rows in the matrix (the **rank** of the matrix). The number of parameters equals the dimension of the zero space. And the zero space, that is the set of solutions to the homogeneous system, that is, our code.

A matrix will always have the same number of independent columns as of rows. If you want all the m rows to be independent you can achieve this by using m independent columns, for instance $\{100\ldots0,\ 010\ldots0,\ 001\ldots0,\ \ldots,\ 000\ldots1\}$ (but written vertically). If you want a code of length n and dimension k you can get this from a matrix with $m = n - k$ rows, if you just ensure that it has m independent columns, let's say the ones previously listed and another k ones.

Example 3.5 We want to make up a code with the dimension 3 (that is, 8 code words), the length 5 and maximum distance.

The length 5 gives us five unknowns in the system, the dimension 3 that we need two equations, so that we can express two of the unknowns using the remaining three.

Since we have to solve the system ourselves we might as well make it easy to solve. When you use Gaussian elimination on a matrix, the goal is usually to get a first column with a one on top and zeros for the rest, a second column with a one in the second row and zeros for the rest, and so on. We may just as well give the matrix this structure from start, and thereby avoid the whole Gaussian elimination!

$$\begin{pmatrix}1 & 0 & \cdots\\ 0 & 1 & \cdots\end{pmatrix}$$

Then we want maximum distance. To get distance $\delta = 3$ we firstly can't have any columns consisting of only zeros, and secondly not two identical columns. Unfortunately this can't be done here, since we need five columns in the matrix and there are only $2^2 = 4$ ways that a column with two elements can look (and one of these is the column with only zeros). Since distance 2 at least is better than distance 1, we choose to use some of the columns twice.

$$\begin{pmatrix}1 & 0 & 1 & 1 & 0\\ 0 & 1 & 1 & 0 & 1\end{pmatrix}$$

The associated system of equations has the solution

$$\begin{cases}x_1 = x_3 + x_4\\ x_2 = x_3 + x_5\end{cases}$$

x_3	x_4	x_5	x_1	x_2	code word
0	0	0	0	0	00000
0	0	1	0	1	01001
0	1	0	1	0	10010
0	1	1	1	1	11011
1	0	0	1	1	11100
1	0	1	1	0	10101
1	1	0	0	1	01110
1	1	1	0	0	00111

The code has as stated the minimum distance 2, which we can detect from the second and third code word, which have two ones each. Besides, we can note that the placement of the ones indicate which of the columns it is that are alike. This code can't correct any errors, but at least we can detect all singel-bit ones.

■

Exercise 3.20: ***Important!*** The reason why we couldn't get the minimum distance 3 in the example above was obviously that we were too stingy with the length. If you want this many this short code word you can't get any error correction as well.

(a) Which length would the code words need to have if we are to get single-bit error correction, if we really want 8 code words?

(b) Design a matrix that fix this!

(c) If we absolutely want code words of this length, how many words can we get if we want single-bit error correction?

(d) Design a matrix that fix this!

(e) Alternatively, if we want words of this length and feel happy with this distance, how many code words can we get?

(f) Design a matrix that fix this!

(g) Lastly: assume that we completely abandon the whole error-control idea, and decide to take as many code words as we can get. How many are they and what does the matrix look like?

Exercise 3.21 Design matrices for the two codes that were designed in exercise 3.14, and check that they seem to match the theory we have been deducing.

Exercise 3.22: ***Important!*** We have now focused on codes that correct singel-bit errors. But if the channel for transmission is bad that might not be enough. How should a matrix that generates a code where two-bit errors as well can be corrected be designed?

Exercise 3.23 Is it possible to design a linear code that corrects *all* errors?

A theroretical upper limit on the number of errors that can be corrected in a code of length n and dimension k is given by the so-called **sphere-packing theorem**:

Theorem 3.3: ***Sphere-Packing theorem*** A code of length n, dimension k, and distance $\delta = 2d+1$ (where d is the number of errors that can be corrected) has to satisfy the inequality

$$2^{n-k} \geq 1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{d}$$

Proof. In the space $(\mathbb{Z}_2)^n$ consisting of all possible words, we correct each incorrect word to the correct code word that is closest. Given certain correct code word $\mathbf{x}$, we will correct all words that differ from $\mathbf{x}$ at at most d places to $\mathbf{x}$. There are of course $\binom{n}{i}$ words that differ from $\mathbf{x}$ at exactly i places, so in total that is

$$1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{d}$$

words that are corrected to $\mathbf{x}$. We can imagine that these words are placed in a sphere with the radius d around $\mathbf{x}$. Since there are 2^k real code words that means that there are

$$2^k \cdot (1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{d})$$

words that can be corrected in this way. That number can at most be equal to 2^n, the total number of words that exist in the space. The theorem follows. ■

People working with code theory do among other things spend their time on finding smart error-correcting codes that are as close to this theoretical upper limit as possible. There exist a rich litterature on error-correcting codes, for instance *Introduction to the Theory of Error-Correcting Codes* by Vera Pless.

3.2 Encryption

So the point of coding is to get a message transmitted correctly. Encryption also concerns transmission of a message, but with the constraint that nobody else should be able to understand the message during transmission. (Now we are entering the realm of spies!) The sender and the recipient have to have some sort of agreement, a **key** to how the message is to be scrambled and recreated, repectively. These operations are called **encryption** and **decryption** or **ciphering** and **deciphering**.

> **Exercise 3.24** If we need encryption because we have to send top secret messages through an unsafe channel, can you think of any problem about the agreement in question?

The simplest form of encryption consists of changing every letter into some other symbol. But longer messages encrypted in this way tend to be fairly simple to decipher, since the different letters occur with different frequencies in the language, besides the fact that there is only a limited number of very short words. If you for instance in an ciphered message in English find a word consisting of just one letter you can suspect that the symbol represents *a* or *I*, the only one-letter words in the language. Probably *a* which is the more common of these two words. One entertaining description of cipher breaking is found in Edgar Allan Poe's *The Gold Bug*.

A special case of encryption by changing every letter into some other symbol is called a **Caesar cipher**, after the most famous user. The system is based on shifting the alphabet a number of steps, for instance changing *a* to *b*, *b* to *c*, and so on. When you reach the end you restart from the beginning, so in this instance *z* is changed to *a*. Caesar cipher is an application of modular arithmetic!

> **Exercise 3.25** You have intercepted the message "Xs fi sv rsx xs fi, xlex mw xli uyiwxmsr", which you suspect is encrypted using Caesar cipher. How should you go about it to break the cipher, and what is the message?

> **Exercise 3.26** In English-speaking environments it's common to use ROT 13, which is a Caesar cipher shifted 13 steps. What can be the point of that?

One of the first things you should demand from a *good* encryption scheme is that every character in the encrypted message should depend on all other characters in the original message. That makes code-breaking methods of this kind impossible, and has the further advantage that messages that closely resemble each other are encrypted into completely different things. Several of the largest users of encryption (banking and military, among others) send lots of messages with very standardised appearances (often the only difference is the date), and if they are encrypted into largely the same thing it can give a spy many clues to what is happening.

We will look at some of these more advanced encryption schemes. But then we'll need to calculate powers in modular arithmetic, so we start with a section covering that.

3.2.1 Calculation of Powers

In several contexts, one of them encryption, you need to calculate *large* powers of numbers in modular arithmetic. ("Large" means exponents in the range of several hundred decimal digits!) Here, you have every reason to check whether there is any way to speed up the calculations.

Method 3.1: ***Periodicity*** Calculate

$$7^{317} \pmod 9$$

One method is to write 317 7:s and sit down and start to multiply. (You get 316 times signs, and the same number of multiplications.) We start like this, and note that the calculations become a lot easier if we reduce modulo 9 in every step instead of saving that operation for last. We write down all the intermediate results.

$$\begin{aligned}
&7^1 = 7\\
&7^2 = 7\cdot 7^1 = 7\cdot 7 = 49 = 5\cdot 9 + 4 \equiv 4\\
&7^3 = 7\cdot 7^2 \equiv 7\cdot 4 = 28 = 3\cdot 9 + 1 \equiv 1\\
&7^4 = 7\cdot 7^3 \equiv 7\cdot 1 = 7\\
&7^5 = 7\cdot 7^4 \equiv 7\cdot 7 = 49 = 5\cdot 9 + 4 \equiv 4\\
&7^6 = 7\cdot 7^5 \equiv 7\cdot 4 = 28 = 3\cdot 9 + 1 \equiv 1 \qquad\qquad \pmod 9\\
&\vdots
\end{aligned}$$

Here something interesting has happened! The fourth row contain the same end result as the first one, from which follows that the table becomes periodic. Every third number is 7, every third 4, and every third 1. We get 7 when the exponent modulo 3 is equal to 1, 4 when it's equal to 2, and 1 when it's equal to 0.

Since $317 = 105\cdot 3 + 2 \equiv 2 \pmod 3$ we see that

$$7^{317} \equiv 7^2 \equiv 4 \pmod 9 \qquad \blacksquare$$

The answers becoming periodic must always occur sooner or later, since there is only a limited number of numbers in $\mathbb{Z}_n$, while the number of possible exponents is unlimited. (An application of the pigeonhole principle!)

But if the modular base is large as well, it may take quite a lot of time before the periodicity starts showing, and then it's fairly inefficient to save all intemediate results and compare every new result to the list, to see if it has appeared before.

Exercise 3.27

(a) Why do we say that we have to save the *whole* list? Won't saving the first element be enough?

(b) *When* will saving the first element be enough?

There exists another method, which is useful when performing non-modular calculations as well:

Method 3.2: ***Exponentiation by repeated squaring*** We take the same problem as before: calculate

$$7^{317} \pmod 9$$

We start by converting the exponent into binary:

$$\begin{aligned}
317 &= 2\cdot 158 + 1\\
158 &= 2\cdot 79 \ \ + 0\\
79 &= 2\cdot 39 \ \ + 1\\
39 &= 2\cdot 19 \ \ + 1\\
19 &= 2\cdot 9 \ \ \ + 1\\
9 &= 2\cdot 4 \ \ \ + 1\\
4 &= 2\cdot 2 \ \ \ + 0\\
2 &= 2\cdot 1 \ \ \ + 0\\
1 &= 2\cdot 0 \ \ \ + 1
\end{aligned}$$

From this we se that

$$\begin{aligned}
&317 =\\
&= 1\cdot 2^8 + 0\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 =\\
&\qquad\qquad\qquad\qquad = 256 + 32 + 16 + 8 + 4 + 1
\end{aligned}$$

This partitioning of 317 into powers of two we use in the following way:

$$7^{317} = 7^{256+32+16+8+4+1} = 7^{256}7^{32}7^{16}7^{8}7^{4}7$$

Now we calculate the different powers of 7:

$$\begin{aligned}
7^1 &= 7\\
7^2 &= 49 \equiv 4\\
7^4 &= (7^2)^2 \equiv 4^2 = 16 \equiv -2\\
7^8 &= (7^4)^2 \equiv (-2)^2 = 4\\
7^{16} &= (7^8)^2 \equiv 4^2 = 16 \equiv -2\\
7^{32} &= (7^{16})^2 \equiv (-2)^2 = 4\\
7^{64} &= (7^{32})^2 \equiv 4^2 = 16 \equiv -2\\
7^{128} &= (7^{64})^2 \equiv (-2)^2 = 4\\
7^{256} &= (7^{128})^2 \equiv 4^2 = 16 \equiv -2 \qquad\qquad \pmod 9
\end{aligned}$$

These values now give that

$$\begin{aligned}
7^{317} &\equiv (-2)\cdot 4\cdot(-2)\cdot 4\cdot(-2)\cdot 7 =\\
&\qquad = (-8)\cdot(-8)\cdot(-14) \equiv 1\cdot 1\cdot 4 = 4 \qquad \pmod 9.
\end{aligned}$$

In this calculation we thus succeded in reducing the number of multiplications from 316 to just 13, out of which eight were when squaring and five when we put it all together. ■

Exercise 3.28 How many multiplications are needed if the exponent is n?

Exercise 3.29 Calculate $17^{255} \pmod{33}$.

Exercise 3.30 Implement exponentiation by repeated squaring. Write a function that exponentiates by repeated multiplication as well. Make test runs for values in different ranges, until you find some where the difference in time is noticable. Calculate the same thing using the pre-programmed exponentiation that is included in the programming language. Which of the two calculation methods does the pre-programmed function seem to be using? *

3.2.2 RSA-Cryptography

In a group consisting of n persons who want to send encrypted messages to each other, in the normal case $\binom{n}{2}$ agreements are needed, one for each pair of persons. That's quite a lot to keep track of and all of it has to be kept secret, otherwise some unauthorised person can unpack the messages.

There is another way of organising cryptography, which is to encrypt using a **public key**. Each person has publicly declared: "If you want to send me a message, encrypt it like this." For this to be secure, a requirement is that knowledge about the encryption key doesn't automatically give knowledge about the decryption.

Every user of the system has an encryption key that is public and a decryption key that only she knows. A person who wants to send a secret message through an unsafe channel encrypts the message using the recipients public key. The message, which now looks like pure nonsense, is sent. The recipient decrypts the message using her secret key and gets the message in plaintext. Anyone not having the decryption key is unable to decipher the message.

If this is to work we have to find two operations, encryption and decryption, that firstly cancel out, secondly give a completely weird intermediate result, and thirdly don't give any information about each other.

The most commonly used method of this kind was invented in 1977 by Ronald L. Rivest, Adi Shamir, and Leonard Adleman, and is called RSA-encryption, after the inventors. It's based on prime number theory and modular arithmetic. They have not succeded completely in making knowledge about the encryption key not giving knowledge about the decryption key, but they have succeded in making the work of generating the decryption key so time consuming that it isn't worth starting.

Mathematical Background To cover RSA-encryption we need some lemmas:

Theorem 3.4: *Fermat's little theorem* If p is prime and this prime isn't a factor in a then

$$a^{p-1} \equiv 1 \pmod p$$

Proof. We start by calculating $(a+b)^p \pmod p$:

$$(a+b)^p = \binom{p}{0}a^p + \binom{p}{1}a^{p-1}b + \binom{p}{2}a^{p-2}b^2 + \cdots + \\ + \binom{p}{p-2}a^2b^{p-2} + \binom{p}{p-1}ab^{p-1} + \binom{p}{p}b^p$$

All the binomial coefficients except for the first one has the prime number p in the denominator. None of them except for the last one has p in the numerator. That means that all the binomial coefficients except for the first and last ones will be multiples of p, and then they are equivalent to zero when calculating modulo p. Remaining are the first and the last terms, so we have

$$(a+b)^p \equiv a^p + b^p \pmod p.$$

From this follows by induction that $a^p \equiv a$ for all integers $a \geq 0$, since this statement is trivially true in the base case $a = 0$, and if it is true that $(a-1)^p \equiv (a-1)$ we can derive that

$$a^p = \big((a-1)+1\big)^p \equiv (a-1)^p + 1^p \equiv (a-1) + 1 = a.$$

So now we know that $a^p \equiv a$ for all positive integers a. If a besides isn't a multiple of p we can find a number a^{-1}, and if we multiply both sides of the equation by that we get

$$a^p \equiv a \quad \Rightarrow \quad a^{-1}a^p \equiv a^{-1}a \quad \Rightarrow \quad a^{p-1} \equiv 1 \pmod p \qquad \blacksquare$$

We need a generalisation of this theorem as well:

Theorem 3.5

$$a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

provided that p and q are *different* prime numbers and that a doesn't include either of them as a factor.

Proof. By applying the just proved Fermat's little theorem and the normal calculation rules for powers we get

$$\begin{aligned}
a^{(p-1)(q-1)} &= (a^{p-1})^{q-1} \equiv 1^{q-1} = 1 \pmod{p} \\
&\qquad \Rightarrow \quad a^{(p-1)(q-1)} = rp + 1 \\
a^{(p-1)(q-1)} &= (a^{q-1})^{p-1} \equiv 1^{p-1} = 1 \pmod{q} \\
&\qquad \Rightarrow \quad a^{(p-1)(q-1)} = sq + 1
\end{aligned}$$

From this we see that rp is the same number as sq, and since p and q are *different* primes, the number is divisible by both p and q *at the same time.* So we have that

$$a^{(q-1)(p-1)} = tpq + 1 \quad \Rightarrow \quad a^{(q-1)(p-1)} \equiv 1 \pmod{pq} \qquad \blacksquare$$

Designing the Encryptor With the mathematical background as support, we are now able to design the encryptor. First we choose the two different prime numbers, p and q. Using them we calculate the two numbers $n = pq$ and $m = (p-1)(q-1)$.

Then we pick two numbers, e and d, that have to have the property that $ed \equiv 1 \pmod{m}$, that is to say, $ed = km + 1$. This is possible, because you can always find some number e that is coprime to m, and then you can work out d using the Euclidian algorithm. (See the example below.)

Now we have that $ed = k(p-1)(q-1)+1$. We take our secret message a, and start by rising it to e, after which we rise the result to d:

$$\begin{aligned}
(a^e)^d = a^{ed} = a^{km+1} = a^{k(p-1)(q-1)+1} = a \cdot a^{k(p-1)(q-1)} = \\
= a \cdot (a^{(p-1)(q-1)})^k \equiv a \cdot 1^k = a \cdot 1 = a \pmod{pq}
\end{aligned}$$

We get back what we had at start! But the intermediate result $a^e \pmod{pq}$ will probably not have any resemblance to a. This is the way the encryption is carried out.

The number e is the public encryption key, the number d is the secret decryption key. The number n is public as well, while it's prime factorisation *isn't.* The file that is to be encrypted is divided into pieces of a suitable size (somewhat smaller than n). Each piece is interpreted as a number in binary and run through the first part of the calculation. The encrypted pieces are then sent, and the reciever runs them through the second part of the calculation and puts the file back together.

Example 3.6 Here we are going to show how keys can be designed and how encryption and decryption is carried out. (The numbers we are going to use are unrealistically small, so that they will be readable. Real keys are in the order of size 1024 binary digits, corresponding to a little more than 300 decimal ones.)

First we need two primes, and we choose $p = 47$ and $q = 11$. That gives us $n = pq = 47 \cdot 11 = 517$, and $m = (p-1)(q-1) = 46 \cdot 10 = 460$ as well.

Then we need an encryption key, and that has to be a number coprime to m. If we pick a prime number here as well we ought to be safe. We choose $e = 97$. The associated decryption key d must have the property $d \cdot e = 1 \mod m$, that is, $e = d^{-1} \pmod{m}$. The inverse of e we can find using the Euclidian algorthm, by searching for $\gcd(460, 97)$ and expressing it as a linear combination of the numbers:

$$\begin{aligned}
460 &= 4 \cdot 97 + 72 & 1 &= 22 - 7 \cdot 3 & &= 22 - 7(25 - 22) \\
97 &= 1 \cdot 72 + 25 & &= 8 \cdot 22 - 7 \cdot 25 & &= 8(72 - 2 \cdot 25) - 7 \cdot 25 \\
72 &= 2 \cdot 25 + 22 & &= 8 \cdot 72 - 23 \cdot 25 & &= 8 \cdot 72 - 23(97 - 72) \\
25 &= 1 \cdot 22 + 3 & &= 31 \cdot 72 - 23 \cdot 97 & &= 31(460 - 4 \cdot 97) - 23 \cdot 97 \\
22 &= 7 \cdot 3 + 1 & &= 31 \cdot 460 - 147 \cdot 97 & &
\end{aligned}$$

$$97(-147) = 1 - 31 \cdot 460 \qquad \Rightarrow \qquad 97^{-1} = -147 \equiv 313 \pmod{460}$$

Thus the decryption key is 313.

An acquaintance now wants to send us the secret message 50. She knows that $n = 517$ and $e = 97 = 64 + 32 + 1$. She calculates $50^{97} \pmod{517}$ using exponentiation by repeated squaring:

$$\begin{aligned}
50^1 &= 50 \\
50^2 &= 2500 \equiv -85 \\
50^4 &\equiv (-85)^2 \equiv -13 \\
50^8 &\equiv (-13)^2 \equiv 169 \\
50^{16} &\equiv 169^2 \equiv 126 \\
50^{32} &\equiv 126^2 \equiv -151 \\
50^{64} &\equiv (-151)^2 \equiv 53
\end{aligned}$$

$$50^{97} = 50^{64+32+1} \equiv 53 \cdot (-151) \cdot 50 \equiv 8 \pmod{517}$$

The encrypted message became 8 which is sent to us. We unpack it using the decryption key $d = 313 = 256 + 32 + 16 + 8 + 1$.

$$\begin{aligned}
8^1 &= 8 \\
8^2 &= 64 \\
8^4 &= 64^2 \equiv -40 \\
8^8 &\equiv (-40)^2 \equiv 49 \\
8^{16} &\equiv 49^2 \equiv 184 \\
8^{32} &\equiv 184^2 \equiv 251 \\
8^{64} &\equiv 251^2 \equiv -73 \\
8^{128} &\equiv (-73)^2 \equiv -159 \\
8^{256} &\equiv (-159)^2 \equiv 52
\end{aligned}$$

$$8^{313} = 8^{256+32+16+8+1} \equiv 52 \cdot 251 \cdot 184 \cdot 49 \cdot 8 \equiv 50 \pmod{517}$$

We may note that the decryption took more time than the encryption, since the decryption key was larger. Often keys are chosen deliberately with this in mind, since the same message is often distributed to a number of different persons, each with their own encryption key. To make this reasonable, the encryption keys have to be simple. ■

Exercise 3.31 Encrypt and decrypt 88 using the same set of keys. (Calculator recommended.) *

Exercise 3.32 Implement RSA-encryption of numbers. Use the function for exponentiation that you wrote in exercise 3.30. *

Exercise 3.33 When we are working with numbers this small, can you invent some method of breaking the crypto, that isn't based on prime factorisation of n?

The safety of the encryption scheme is coupled to the fact that there doesn't exist any (known) fast method for prime-factorising of numbers, while it on the other hand is possible to calculate large powers in modular arithmetic. That makes it possible to choose p and q (and thereby n) large enough to make it impossible to prime-factorise n in a reasonable amount time. (If you have succeded in doing the factorisation you can find m, and using m and e find d in the same way as the designer of the cipher did, and then you have broken the code.) On the other hand, nobody has yet *proved* that there doesn't exist some other way of breaking the crypto that doesn't require that you factorise n – but the belief is that no such method exists.

Exercise 3.34 You have intercepted the encrypted message 444, from a person whose public key is $e = 797$, $n = 1961$. You have also succeded in spying out that $q = 53$. Decipher the message!

To keep up to date with the latest development on the prime-factorisation front, the company that market RSA-encryption runs a competition in factorising large composite numbers. The prizes are in the range of several thousand dollars. When this text was written, the largest completed number consisted of 512 bits (155 decimal digits), and the computation (which was distributed over a number of machines) took about 7.4 months and was finished 22 August 1999. The development of algorithms for factorisation has by the way been a lot faster than anticipated – in 1985 it was stated that the factorisation of a 100-digit number would take 74 years of computations.

The encryption method is also dependent on the feasibility of finding large prime numbers, otherwise there wouldn't be anything to base it on. The method used is to generate a random number in the right range and check whether it by chance is prime. If not one generates a different number, and keep on like that until success. There are fail-proof primality-checking algorithms that don't take too much time. And in these circumstances it's possible to manage using numbers that with great but not a hundred percent probability are prime. These test are a lot faster.

Most primality tests are too theoretically complicated to be covered here, but we can look at one of them, the **Fermat test**, that is based on Fermat's little theorem. That says that

$$a^{p-1} \equiv 1 \pmod p$$

if p is prime and a doesn't include p as a factor. If we have a number q that we wonder whether it is prime we can insert it into the theorem. If

$$a^{q-1} \not\equiv 1 \pmod q$$

we can at least be a hundred percent sure that q *isn't* prime. If it passes the test it's probably (but not certainly) prime. A composite number that sneaks through a test like this is called a **pseudoprime**. If we test using several different values of a the probability that the number really is prime increases. Still there exist sneaky composite numbers that pass the Fermat test for all numbers a except the ones containing one of the factors of the number! Such numbers are called **Carmichael numbers**. The smallest of them is 561, that can be factorised as $3 \cdot 11 \cdot 17$.

Exercise 3.35 Check whether 91 seems to be prime by running the Fermat test, first using the base 3 and then the base 2.

Exercise 3.36 Do you know of, or are you able to invent, some other primality tests?

In some cases one isn't just anxious that nobody should be able to intercept data – one is also anxious that nobody should be able to fake a message. If the local office of the bank gets a message telling them to transfer ten billions to an account in Irkutsk they probably want to feel sure that the order really originated at the main office.

In this situation it is possible to use the fact that not only $(a^e)^d = a$ but $(a^d)^e = a$ as well. If the sender starts by running the message through her own decryptor before encrypting it using the key of the intended recipient, the recipient can unpack the message by firstly decrypting it in the normal way, and then encrypting it using key of the purported sender. If you get an intelligible message it's with high probability really the stated person who sent the message. This is called **electronic signature**.

3.2.3 DES-Encryption

RSA-encryption is, as shown, mathematically elegant, but it has a drawback: encryption and decryption take time. (That they take time is perhaps unavoidable, but they take a *long* time.) An alternative system is **DES-encryption** (Data Encryption Standard)[1]. This system is based on secret keys. The two participants have access to the same key, which is used both for encryption and decryption. The key is usually encrypted using RSA or some other public-key system and transmitted before the message. That makes it possible to change the key often, which increases the security.

The system has a rather amusing backstory. In the USA there is an organisation, NSA (National Security Association), that among other things works with cryptography. They are for reasons of principle opposed to anyone else being able to encrypt anything, since that complicates their activities. Still, there was a need for working cryptography in the USA (it's among other things a necessity in computerised banking), and in the 1970s the American National Standards Institute announced a competition to invent a crypto-system that met certain criteria. The second time around they got hold of something useful, and a design from IBM was used as a base for the system. Then NSA went through the algorithm and changed a number of details. But they refused to explain why, so the public opinion was that they had added a backdoor making it possible for them to identify the encryption key and decrypt anything they wanted. As a matter of fact, they had enhanced the system so that it became impervious to a crypto-breaking method that they alone know about – but that wasn't anything they wanted to explain publicly.

The Standards Institute then as usual made public how an encypter was to be made to confirm to the DES standard, which NSA hadn't intended at all. Their plan was that the encryptor should be distributed in the form of ready-made chips to be plugged into the computers. But not everyone is that keen on buying a black box that the country's spying authorities asssure is completely reliable, so that the material was made public probably contibuted a lot to the popularity of the system. Still, the most common thing is that the encryptor is installed in form of a ready-made chip, since custom-made

[1]By now mostly replaced by **AES-encryption**, Advanced Encryption Standard, which uses longer keys.

hardware works a lot faster than software. The code method is furthermore optimised to be easily implemented in hardware, and uses operations that are very time consuming on an ordinary processor.

How DES Works DES is not at all as mathematically elegant as RSA, but we will give a short sketch of the way it works. For a start you have a key consisting of 64 bits, out of which 8 are parity bits, used for error control. There are thus $2^{64-8} = 2^{56} \approx 72 \cdot 10^{15}$ possible keys. To break an encrypted message by simply test decrypt it using all existing keys, and choosing the result that seems reasonable, is thus rather time-consuming (but by now entirely feasible, which it wasn't when the standard was introduced).

The document that is to be decrypted is divided into 64-bit blocks. Each plaintext block is transformed into an ciphertext block of the same length. The block passes through the encryptor sixteen times, and each time some of the bits are moved and some of the bits are changed, based on the way the key looks. The original key is used to make 16 different subkeys, and each pass uses one of them. The decryption is done in the same way, except that the subkeys are used in the opposite order, which means that you move back and change back everything you changed during encryption.

To make it even more difficult for spies, each block is added to the encrypted version of the previous block before encryption. That means that one and the same plaintext will be encrypted into completely different things depending on where in the document it appears.

DES-encryption is between a hundred and a thousand times faster than RSA-encryptions, and thanks to the existence of RSA-encryption the handling of keys is simple.

Still, there exists a fairly simple method of breaking this: A spy who already knows that she want to intercept messages sent to a certain person can in advance test-encrypt all the 2^{56} possible DES-keys using the recipient's public key, and store the results. Then she intercepts the encrypted key when it's sent, and compares it to the database. That gives her the key in plaintext, and then she can peacefully decrypt the intercepted message. The American authorities have already decided to switch to longer keys, so that the database over encrypted keys becomes unreasonably large, to prevent this. If you switch to for instance 256-bit keys, the database would need to contain more posts than there are atoms in the Milky Way, and it doesn't seem likely that technology will ever bring about efficient computers with memory capacity on this scale.

A from a mathematical point of view interesting question that remained unanswered for a long time is whether DES-encryption is a group operation. One way of extra-specially encrypt something is to run it twice through the encryption, using two different keys. But the question is whether the whole thing gets any more secure by this. If the operation is a group operation the set is closed, so the combination of two operations is an operation. In that case, encryption using two keys consecutively would be equivalent to encryption using a third key, and in that case the only thing gained in the process would be that it has taken twice as much time. Now it has been found that DES isn't a group (but there exist other encryption systems that are).

3.2.4 Crypto-Breaking

As shown by the description of the systems, encryption is used when one wants to keep things secret. Secrets are things that some people are interested in. How thoroughly you have to cipher depends on how secret things are, and even more on for how *long* they will remain secret. Lots of messages, like "attack at dawn" and "sell, damn it, Ericsson is going burst" will be out-of-date the next day, and then it doesn't matter much if the crypto can be broken in a week. Other kinds of material, like the secret archive of MI-6, may be of interest even if you have to spend several years unpacking it after stealing it. The RSA-company recommends for instance 2048-bit modulos for things that you really don't want to be known ever, while 1024-bits are enough for every-day applications.

Most crypto-breaking is by the way not done using mathematical methods. A crowbar, pistol, or a couple of thousand dollars in cash are often more efficient means. Looking through garbage or making use of people's tendency to choose super simple passwords or writing them on a piece of paper tacked to the computer is efficient as well.

If you really want to be mathematical you can use the fact that codebreaking is an activity that is eminently fit to be **parallelised**, that is to say, to divide into several separate calculations that can run on separate computers. If you can distribute the calculations to a thousand machines it will take a thousand of the time, and the difference between one week and 19 years is from a practical point of view rather large. Adding the calculation to screen-savers works fine, so that the computers work on this while not used for something else. In that way it isn't disturbing, which increases the chance that people will let you borrow their computers for this purpose. One special version of this is not asking for permission, but writing a small virus that goes around, installs itself on the computers and multiply, and runs its part of the calculation and when it's finished sends in the results and uninstalls itself. (How common this is is hard to say, but it's an interesting possibility.)

Anyone who wants to read more about encryption and code-breaking is warmly recommended *The Code Book* by Simon Singh.

3.3 More Exercises

Encoding

Exercise 3.37 We have the code

$$\{0000000000, 1111111111, 0000011111, 1111100000\}$$

(a) Verify that the code is linear.

(b) Which is the minimum distance of the code, and how many errors will it correct?

(c) Design a check matrix for the code.

(d) You recieve 0001100111. How many errors does this word include (at least) and to what would you correct it? And how does this answer match the answer on (b)?

(e) Is it possible to design a code with this length and dimension that corrects three errors?

(f) What does the sphere packing theorem say about the question in (e)?

Exercise 3.38 Assume that we want to send 256 different messages using a binary code that corrects single-bit errors.

(a) What is the least possible length of such a code?

(b) Write a suitable check matrix for such a code of minimum length.

Exercise 3.39 Here you are going to analyse a non-binary code with error detection:

Each published book has a so-called ISBN-code. The ISBN-code[2] is a string $a_1a_2\dots a_{10}$ of integers where $0 \le a_i \le 9$ if $1 \le i \le 9$, while $0 \le a_{10} \le 10$. (But instead of 10 the letter X is used.) a_{10} is a check digit that is calculated according to the formula

$$a_{10} = \sum_{i=1}^{9} ia_i \pmod{11}.$$

Show that if two different digits a_j and a_k exchange places or if a digit a_j is changed the check digit will be wrong. (Hint! It's easier to study $\sum_{i=1}^{10} ia_i$ (mod 11).) *

Encryption

Exercise 3.40 Assume that the professors A and B participate in a system for RSA-cryptography and have the following public keys:

$$n_A = 91, \quad e_A = 5, \quad n_B = 35, \quad e_B = 7$$

Unfortunately 91 and 35 are small enough numbers to make it possible for a proficient person to break the crypto.

(a) Determine the decryption keys d_A and d_B.

[2]This system has been replaced with one with longer codes, ISBN-13, since the number of published book started to exceed the number of possible codes.

(b) B wants to send the message 23 to A coded using electronic signature. What should B send?

Exercise 3.41 One big problem when using cryptography is that by encrypting something you have at the same time shown that you have something to hide. Ponder upon whether you know of any methods to get around that problem!