CHAPTER ONE

Shannon's Free Lunch

1.1 The ISBN Code

Pick up a paperback book, any book which was published fairly recently, and on the back you will find a number—the ISBN or International Standard Book Number. (Hardback books are sometimes numbered inside the front cover.) The ISBN identifies the title among all titles published internationally. The ISBN sequence of this book is

0-691-11321-1.

(The hyphens are not important for our purposes but I left them in to make the number easier to read.) This number has a surprising property. In Table 1.1 the ISBN digits are written vertically in the first column. The numbers from 1 to 10 are written in the second column. The third column is formed by multiplying across the rows; for example, in the second row, the first two entries are 6 and 2, so the third entry is $6 \times 2 = 12$. Once the third column is formed, the 10 numbers that are obtained are added up and the total, 110, is written at the bottom.

The *surprising* thing is that this total, 110, turns out to be divisible by 11. The fact that the total turned out to be divisible by 11 is no coincidence: it would happen whichever book you used. Try it for yourself, with your favourite book. (If the ISBN has an X at the end, then treat it as the number 10.)

You might at first imagine that this is one of those mathematical magic tricks: whatever number I write down, the total will always end up divisible by 11. To see that this is *not* the case, imagine what would happen if I changed the last digit of

Table 1.1. The special property of an ISBN.

ISBN	Multiplier	Product
0	1	0
6	2	12
9	3	27
1	4	4
1	5	5
1	6	6
3	7	21
2	8	16
1	9	9
1	10	10
	Total	$110 = 11 \times 10$

the ISBN from 1 to 2. The bottom entry in the third column would change from 10 to 20 and the total would increase by 10, to 120, which is not divisible by 11. So the fact that the total *is* divisible by 11 depends upon the fact that the ISBN is a special number. All ISBNs are special numbers. The question is 'why'. Why would publishers choose only to use certain special numbers to identify their books?

Imagine that you are ordering a book from a publisher. You write the ISBN on the order form, or type it into a computer, but you make a small mistake; for example, you accidentally change the fifth digit from a 1 to a 2. What happens to the total in column three? The fifth entry in column three changes from 5 to 10. So the total increases by 5 and ends up not being divisible by 11. When the publisher receives your order, he or she (or the computer) can see that the number of the book you have ordered is not the right kind of special number: there is no book in existence whose number is the one you ordered. So the publisher knows that you have made a mistake, and instead of sending you the wrong book, which would be very costly since books are heavy, the publisher just asks you to reorder.

The ISBN is an example of an error-detecting code. The ISBN encodes information about the book (its publisher, title and

so on) but also, the ISBN can automatically detect errors: if you accidentally change one of the digits, any recipient of your message can see that there has been an error. The same thing happens if you accidentally swap two adjacent digits (which is a fairly common mistake for humans to make). Try the following problem before reading on.

Problem 1.1. Can you work out what feature of the code enables it to detect a swap?

The answer to this question lies in the second column of the table above. The ISBN code can detect swaps because of the different multipliers $1, 2, \ldots, 10$ that appear in the second column. Suppose you have two successive digits a and b in (say) the fourth and fifth places. Then the fourth and fifth rows of the table look as follows.

ISBN	Multiplier	Product
$\vdots$	$\vdots$	$\vdots$
a	4	$4a$
b	5	$5b$
$\vdots$	$\vdots$	$\vdots$

Now if you were to swap the a and b, then the corresponding rows would be as follows.

ISBN	Multiplier	Product
$\vdots$	$\vdots$	$\vdots$
b	4	$4b$
a	5	$5a$
$\vdots$	$\vdots$	$\vdots$

The total of the third column would thus be increased by a and decreased by b, since the a contribution changes from $4a$ to $5a$ and the b contribution from $5b$ to $4b$. So the total is changed

by $a - b$. The new total cannot be divisible by 11 unless $a - b$ is 0 (in which case a and b are the same digit and swapping them makes no difference).

The use of multipliers to distinguish the contributions of the different digits in the ISBN is the main reason that the code is based on divisibility by the number 11 rather than by the number 10, which might seem more natural.[1] Unlike 11, the number 10 is not a prime number. If we were to use divisibility by 10, then some of the digits would be 'booby-trapped'. Suppose that the fifth row is meant to read as follows.

ISBN	Multiplier	Product
⋮	⋮	⋮
3	5	15
⋮	⋮	⋮

If you accidentally change the digit 3 of the ISBN to 7, the number in the third column changes from 15 to 35—it changes by 20. Since 20 is a multiple of 10, the divisibility by 10 criterion would not detect the change. The problem here is that 10 is 5×2 so if you change the ISBN digit by 2, 4, 6 or 8, you will change the third column by a multiple of 10. With a prime number like 11, this cannot happen.

When I first came across error-detecting codes, I was absolutely delighted. Just by a careful choice of the 'language' in which you communicate information, you can automatically reduce the likelihood of misunderstanding. As you can imagine, this mathematical idea has numerous applications in our modern era of electronic information transfer. But in some ways, error-detecting codes are the oldest human idea of all. In a sense, all human languages are codes which surround the essential information with extra grammatical and syntactical devices that enable the listener to confirm the meaning of what

[1] Codes based on prime numbers have certain structural advantages in addition to the one described here, but these are rather more subtle than are needed for the ISBN.

is being said. This example helps to illustrate the point that error-detecting codes have nothing to do with cryptography—with codes that hide information from your enemies. The point of an error-detecting code is to transmit information accurately, not to transmit it secretly.

1.2 Binary Channels

When space-probes, such as the one pictured in Figure 1.1, send information (pictures of Mars or Jupiter) back to Earth, they have to encode the information in some way, usually as a sequence of 0s and 1s. The radio waves that carry the information to Earth will have to pass through atmospheric interference, and so the message that gets picked up on the ground will not be quite the same as the message that was sent. The *channel* by which the message is communicated is not perfectly accurate.

One way around this problem would be to replace each 1 by a long sequence of 1s (say ten of them),

1111111111,

and each 0 by ten 0s,

0000000000.

If your ground-station receives the sequence

1101110111,

you know that this was not the sequence that was sent, and you can be pretty certain that the true message was all 1s. So you can be pretty sure that the correct symbol was a 1 rather than a 0. This procedure of replacing each binary digit (or bit) by a long string of bits, constitutes a code, albeit a crude one. As with the ISBN code, certain messages are 'disallowed', and so if you receive such a message you can tell that there has been an error. However, this code does quite a bit more for you than the ISBN—it gives you a very good chance to *correct* the

Figure 1.1. Artist's impression of NASA's Mars Reconnaisance Orbiter, due for launch in 2005.

error, rather than merely *detect* it. That is just as well, because whereas a publisher can ask you to reorder a book, you cannot ask a satellite to re-photograph Mars after it has gone off towards Jupiter. This 'repetition' code, as we might call it, is a simple example of an error-*correcting* code.

The problem with the repetition code is that it is tremendously wasteful. In order to send each bit of information, the satellite has to send 10 bits of data. The transmission rate is only 10%. When you are trying to recover information from a space probe it is important to get a high rate of transmission, because the probe will only visit each planet for a short time and has only limited power. This looks like an example of the 'no free lunch' principle which turns up a lot in mathematics. If you want a high degree of accuracy, you must sacrifice speed of transmission.

But in 1948 the mathematician Claude Shannon discovered that the trade-off between speed and accuracy is not inevitable—if you eat at the right restaurant, the lunch is free. What Shannon discovered is that if you design your code very carefully, it is possible to achieve almost perfect accuracy and still transmit at a fixed rate (which depends only upon the quality of the channel). For example, if your transmission channel is just 90% accurate, you can achieve whatever accuracy you desire at a transmission rate of about 50% *provided you design your code carefully*. Shannon's discovery was extremely startling; it is hard to believe that you can achieve better and better accuracy, at the same rate of transmission, just by choosing your code correctly.[2]

However, there is a small catch. The problem with highly efficient codes is that they have to be complicated, and that means that it can be very time-consuming to decode the messages you receive. (The difficulty here is not to 'break' the code; you designed the code, so you know *how* to decode the messages. The problem is that for a complicated code, it just takes time to do the decoding.) In applications like space missions, that is not a big problem, because the messages can be recorded when they arrive on Earth, and then decoded at leisure. On the other hand, for information transmission from person to person, the recipient wants to hear the news quickly. So it is important to choose a code that matches your needs.

Shannon's Theorem tells you that efficient codes exist, but it does not tell you how to design them; much less does it tell you how to design codes that are easy to use. So, shortly after Shannon's ground-breaking work, people set about inventing good codes.

1.3 The Hunt for Good Codes

What does it mean to 'design a code'? What decisions are involved in inventing a code? When you design a code, you have

[2] The proof of Shannon's result is quite straightforward by modern standards, but it is a bit beyond the scope of this book.

to select your 'alphabet'; for example, are your messages built out of 0s and 1s or out of decimal digits like the ISBN code? But the important question is which strings of symbols will be admissible and which will not. For example, in the repetition code mentioned above, the message is broken into strings of 10 binary digits. The admissible strings are

1111111111

and

0000000000,

while all the other possible strings of 10 bits are inadmissible. The strings into which you break the message are called 'words'. You have to choose how long each 'word' is going to be and you must decide which words are 'allowed' and which ones are 'disallowed'. (Eventually, when you come to use a code for a specific purpose, you also have to decide how to express *your* information in terms of the admissible codewords. This is usually just a matter of doing whatever takes your fancy, and has little to do with the design of the code.)

Let us build a simple code to try and understand what issues we have to address. Suppose we have decided to build a binary code[3] of length three: each codeword will then be one of the eight triples:

000 001 010 011 100 101 110 111.

We have to choose which ones to use as codewords and which ones to omit. The repetition code consists of choosing the first and last as codewords and rejecting the other six. But we want to try to improve upon the repetition code by including a few more codewords without sacrificing too much reliability.

To maintain reliability we want to choose codewords that are 'as different as possible' so that it is unlikely that one codeword will be accidentally converted into a different one, by the inaccuracy of our communication channel. On the other hand we want to choose a lot of words, so that we can send more

[3] A code using the digits 0 and 1.

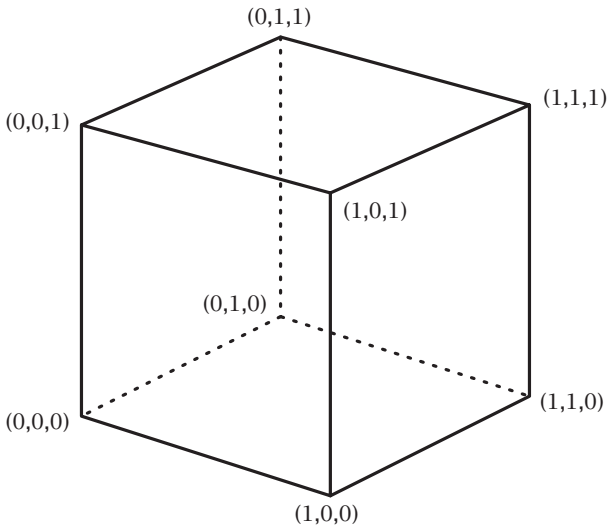

Figure 1.2. Representing strings as points in 3-space.

information using the same number of bits. We can get a better picture of what we are doing by representing the possible triples as the corners of a cube, just by regarding each string of 0s and 1s as a point in three-dimensional space. Thus 000 becomes the point $(0, 0, 0)$ and so on (see Figure 1.2).

The requirement that our codewords should be 'as different as possible' can be interpreted geometrically: it says that the corners of the cube that we select should be well separated from one another, i.e. that it takes several steps to get from one to another. The repetition code consists of choosing two opposite corners, $(0, 0, 0)$ and $(1, 1, 1)$, which are three steps apart. The resulting simple code is called a 3-separated code. What else can we do? It is not hard to see that there is a way to choose four corners, no two of which are adjacent, in which case it takes two steps to get from one to another. Such a choice of four corners is illustrated in Figure 1.3. This choice corresponds to the selection of codewords:

000
011
101
110.

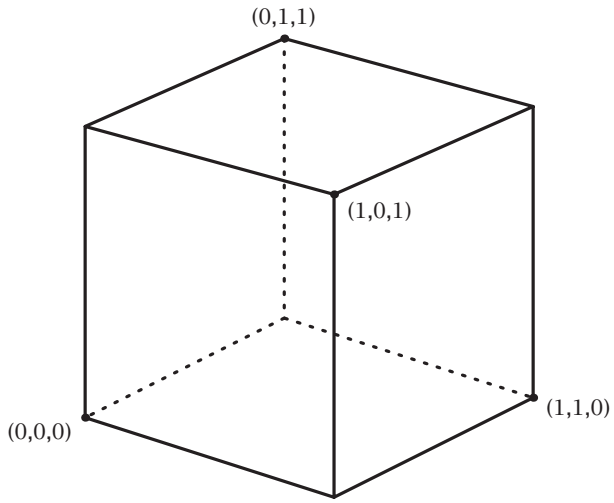

Figure 1.3. A 2-separated code.

This code is 2-separated; so it is not as reliable as the repetition code, but we have twice as many codewords. Let us calculate the transmission rate of this code?

For the tenfold repetition code the rate was 10% because the satellite needed 10 bits of data to send each bit of information. The number of bits' worth of information depends upon the number of different words in your code. If you have four words in the code, you have a choice of four different messages that you can send, just as if you were using every possible combination of 2 bits:

$$00 \quad 01 \quad 10 \quad 11.$$

So four codewords provide you with 2 bits' worth of information. Similarly eight codewords provide 3 bits' worth. In general, if you have a choice of 2^m possible codewords, you have m bits' worth of information.

In the case of the cube code, we have 2 bits' worth of information, out of 3 bits of data. So the rate is $2/3$: the cube code is a 2-separated code with rate $2/3$. That compares well with the 2-bit repetition code

$$00 \quad 11,$$

which is also 2-separated but only has rate $1/2$.

We have thus built a code which genuinely improves upon the simplest repetition code, by using some geometric intuition. In order to design really useful codes we have to combine several different techniques—both geometric and algebraic. There are many methods that have been invented for this purpose and the codes they produce have widely differing uses. One such method is described in the next section.

1.4 Parity-Check Construction

If you look at the cube code we found earlier

$$\begin{array}{c} 000 \\ 011 \\ 101 \\ 110, \end{array}$$

you can see that there is a simple way to describe the codewords: they are the strings that contain an even number of 1s. With this description it is immediately obvious why they differ in at least two places. In mathematics we refer to the distinction, 'even versus odd', as *parity*. For the code we are looking at, you can test whether a word is admissible by testing the parity of the sum of its digits. A similar idea underlies the construction of the ISBN code. In order to test whether a sequence is admissible you add up certain multiples of the digits and then check whether the sum is divisible by 11.

This idea of checking parity can be used to build some quite useful codes. Let me explain how to construct a 3-separated code on 7 bits that has rate $4/7$; much better than the 3-separated repetition code, which has rate $1/3$. My codewords will be strings of 7 bits:

$$b_1, b_2, b_3, \ldots, b_7.$$

The admissible strings will be those which satisfy the following three 'parity rules':

$$\begin{aligned} b_1 + \quad b_3 + \quad b_5 + \quad b_7 &\text{ is even,} \\ b_2 + b_3 + \quad b_6 + b_7 &\text{ is even,} \\ b_4 + b_5 + b_6 + b_7 &\text{ is even.} \end{aligned}$$

I claimed that this code has rate $4/7$—in other words, that there are 4 bits' worth of information out of every 7. That is the same as saying that there are $2^4 = 16$ codewords. Here is one way to see why. The bits numbered 1, 2 and 4 each appear in just one of the parity rules. That means that you can build an admissible codeword as follows. Begin by selecting the bits b_3, b_5, b_6 and b_7 in any way you wish. For example, you might pick

$$b_3 = 1, \quad b_5 = 1, \quad b_6 = 0, \quad b_7 = 1.$$

Now fill in the blanks, b_1, b_2 and b_4, in just the right way to make each of the parity rules hold true. Look at the first rule: it says that $b_1 + b_3 + b_5 + b_7$ is even. So far, you have selected b_3, b_5 and b_7 all to be 1 and so their sum is odd. To make the total come out even, you need to choose b_1 to be 1 as well. That will take care of the first rule. What about the second? This time you need to take into account the fact that b_3, b_6 and b_7 have already been chosen and their sum is even. So you need to pick b_2 equal to 0 in order to satisfy the second rule. Finally, you can check that you pick $b_4 = 0$ to arrive at the admissible codeword

$$1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1.$$

The same method of filling in the values b_1, b_2 and b_4 can be used to build a codeword whichever values we select for the bits b_3, b_5, b_6 and b_7. Thus all 16 possible choices of the latter bits give rise to codewords. So we certainly have at least 16 different codewords. Using the same idea, you can check that there are no others. In fact it is not too hard to list all 16, but when you move on to more sophisticated codes, that would not be so easy. The 16 words of this code are given in Table 1.2.

I stated before that this code is 3-separated. You can check this by looking at the table of codewords: any two of them differ in at least three places. (As you might hope, there is a more 'theoretical' way of checking, which is useful when the codes get bigger; this is explained in the solutions section at the end of the chapter.) This 7-bit code is an example of what is called a Hamming code after the mathematician R. W. Hamming. The

Table 1.2. A 3-separated code.

1	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0
3	1	0	0	1	1	0	0
4	0	1	1	1	1	0	0
5	0	1	0	1	0	1	0
6	1	0	1	1	0	1	0
7	1	1	0	0	1	1	0
8	0	0	1	0	1	1	0
9	1	1	0	1	0	0	1
10	0	0	1	1	0	0	1
11	0	1	0	0	1	0	1
12	1	0	1	0	1	0	1
13	1	0	0	0	0	1	1
14	0	1	1	0	0	1	1
15	0	0	0	1	1	1	1
16	1	1	1	1	1	1	1

3-separated repetition code

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

is also a Hamming code. Indeed, for each number of the form $2^n - 1$ (the numbers 3, 7, 15, 31, 63, 127 and so on), there is a Hamming code of this length, and they are all 3-separated.

When we started hunting for codes, we were interested in finding codes that would improve accuracy without being too complicated to be useful. Hamming codes, like other codes that are constructed by means of parity checks, have a big advantage in this respect. They have a lot of structure which makes them easy to decode.

1.5 Decoding a Hamming Code

Let us see how to decode the Hamming code described in the last section. We have to be a bit more precise about what we mean by 'decoding'. Each time we receive a message we break it into strings of 7 bits and then we examine each string to see

if it is an admissible codeword. If it is, then we accept it, i.e. we assume that it was sent correctly. What do we do if we come across a string which is *not* one of the 16 codewords? We know immediately that it was not the string that was originally sent, i.e. that it got corrupted along the way. What we would like to do is to try to guess which codeword *was* originally sent. Given a corrupted string we want to try to find the 'nearest' codeword, i.e. the codeword that is most similar to the thing we received, since this will be our best guess as to what was sent.

To get an idea of what to do, let us look first at the simple repetition code:

$$0\ 0\ 0$$
$$1\ 1\ 1.$$

If, instead of receiving one of these codewords, we get one of the other six strings of 3 bits,

$$0\ 0\ 1$$
$$0\ 1\ 0$$
$$1\ 0\ 0$$
$$0\ 1\ 1$$
$$1\ 0\ 1$$
$$1\ 1\ 0,$$

we have no difficulty in deciding which codeword to guess. If the string has two 0s and one 1, we guess that the original was 000. If we receive a string with one 0 and two 1s we guess 111. The situation here is rather simple because each string is either itself a codeword, or is one step away from a codeword (and it cannot be one step away from each of two different codewords). In other words, each codeword is surrounded by a little 'cloud' of neighbours, which are only one step away, and these neighbourhoods include all the possible strings of 3 bits. This is illustrated in Figure 1.4.

So, in order to decode a string, we find out which cloud it belongs to, and in the 'centre' of that cloud will be the codeword we want. The fact that the code is 3-separated plays an important role in this process. If we had a string which was adjacent to two different codewords, then we would not know

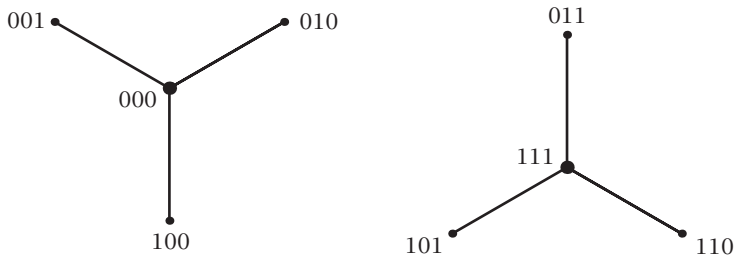

Figure 1.4. Clouds for the 3-bit repetition code.

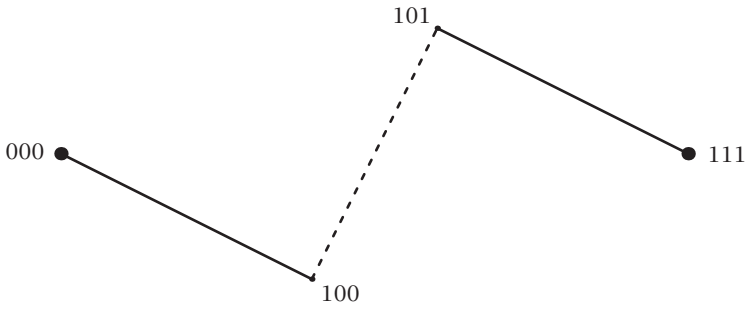

Figure 1.5. Clouds cannot overlap.

how to decode it. But if codewords are three steps apart, this cannot happen (see Figure 1.5).

Now let us see if we can do the same thing with the 7-bit Hamming code. This code is also 3-separated; so if each string is adjacent to a codeword, there will be no ambiguity about which one. The problem is that as far as we know, some strings might be a long way from any codeword. It does not seem like a good idea to check all 128 possible strings to find out. You could try checking a few strings written down at random. For example, is the string

1 1 1 1 0 0 1

adjacent to one of the codewords listed in Table 1.2? Once you have checked a few strings, you will probably be convinced that they *are* all within one step of a codeword. Is there a quick way to check this?

Suppose that instead of looking at each string which is *not* a codeword and trying to decide whether it is *adjacent* to a

codeword, we look at each codeword and ask which strings are adjacent to it. For example, what about the simplest codeword

$$0\quad 0\quad 0\quad 0\quad 0\quad 0\quad 0?$$

Which strings are one step away from this one? There are seven of them, obtained by switching each of the seven bits. So this codeword has a neighbourhood consisting of itself and seven other strings:

$$\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1.
\end{array}$$

In the same way, each codeword is at the centre of a cloud of eight strings—itself and seven others that are not codewords. Now there are 16 of these clouds, one for each codeword. What is more, the clouds cannot overlap because no string can be adjacent to two different codewords. So we have 16 non-overlapping neighbourhoods, each of eight strings. So they cover all $128 = 16 \times 8$ strings.

Now we know that the 7-bit Hamming code has a perfect decoding procedure. For each corrupted string that you receive, there is one and only one codeword which is one step away. That codeword is the best guess you can make as to what was originally sent. However, that does not tell us any very efficient way to work out which codeword is the right one. If you receive the string

$$1\quad 1\quad 0\quad 1\quad 1\quad 0\quad 1,$$

you can easily check that it is not a codeword because it violates the first and third parity rules. But it is very tedious to go down the list of codewords to find which one it is adjacent to. It would be even more tedious with a bigger code containing thousands of admissible words. Fortunately, there is a shortcut which depends upon the fact that the code is defined using parity rules.

In order to understand the shortcut, it helps to represent the parity rules in a matrix: each rule corresponds to a row of the matrix and each bit position to a column. Each entry in the matrix is 1 or 0 depending upon whether the corresponding bit features in the relevant rule. The matrix for the 7-bit code is as follows.

	b_1	b_2	b_3	b_4	b_5	b_6	b_7
Rule 1	1	0	1	0	1	0	1
Rule 2	0	1	1	0	0	1	1
Rule 3	0	0	0	1	1	1	1

Thus, for example, there is a 1 in the second row and third column because b_3 appears in the second parity rule.

Let us go back to our corrupted string

$$1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1$$

and recall that it violates the first and third parity rules (but satisfies the second). We want to find an admissible codeword which is only one step away. Our aim is to switch just one bit in our string so that we end up with a codeword, i.e. a string that satisfies all the parity rules. Now, whichever bit we are hunting, it corresponds to a column of the parity-check matrix. We want to find a column of this matrix with the property that changing the corresponding bit will alter the parity in rows 1 and 3 but will not affect row 2. That means that we want the column to be

$$\begin{matrix} 1 \\ 0 \\ 1. \end{matrix}$$

The question is, does this column appear in the parity-check matrix? Indeed it does: it is column number 5. Thus, if we change bit number 5 in our string, we will alter the parity of rules 1 and 3 because bit number 5 appears in these rules, but we will not alter rule 2 because bit 5 is absent from this rule. Changing bit number 5 from 1 to 0 in our string gives us

$$1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1$$

and sure enough this is a codeword—the ninth on the list in Table 1.2.

The procedure worked fine for this particular corrupted string, but how do we know that it will always work? Was it just luck that the column

$$\begin{matrix}1\\0\\1\end{matrix}$$

appeared in the parity-check matrix or was that bound to happen? Look again at the columns of the matrix. They are the binary representations of the numbers 1–7 (written vertically upwards). For example, the binary representation of 4 is 100 and sure enough the fourth column of the matrix is

$$\begin{matrix}0\\0\\1.\end{matrix}$$

Thus, the columns of the matrix constitute all possible sequences of three bits except 000. So whichever set of parity rules we wanted to alter, we could do it with the appropriate column.

Problem 1.2. Try to use the above procedure to decode the string

$$0\quad 1\quad 1\quad 1\quad 1\quad 0\quad 1.$$

The 7-bit Hamming code has a perfect decoding procedure that is very simple to implement. For each corrupted string that you receive, there is one and only one codeword which is one step away, and you can find out very quickly which it is by testing your string against the parity rules. As was mentioned earlier, there are Hamming codes of lengths $3, 7, 15, 31, 63, \ldots$, all of which are 3-separated and they are all easy to decode. But for most practical purposes we need codes with greater separation distances. The construction of good codes has become a branch (or sub-branch) of mathematics in its own right, the essentials of which would fill at least a book. Some such books

are mentioned in the Further Reading section at the end of the chapter. To finish this section, here are some problems involving Hamming codes.

Problem 1.3. Can you construct a 4-separated code on 7 bits which has eight codewords? You could try to write down four parity rules or else find the eight codewords by (educated) trial and error. Alternatively, you could think carefully about the 7-bit Hamming code and extract from it a 4-separated code.

Problem 1.4. Can you generalize the 7-bit Hamming code to find an analogous 3-separated code of length 15 (or more generally $2^n - 1$)? How many words are there in this code?

∗ **Problem 1.5.** How would you check that the generalized Hamming code of the previous problem is 3-separated?

Shannon's discovery of the free lunch led to the hunt for codes that could be used to send information accurately and quickly. His remarkable work also led to the development of another sub-branch of mathematics: information theory. The connection is briefly described in the next section.

1.6 The Free Lunch Made Precise

Shannon's Theorem guarantees that with a suitable code it is possible to transmit with arbitrary accuracy at a fixed rate which depends only upon the quality of the channel. He was able to describe the dependence precisely. If the channel has a probability p of transmitting each bit correctly, and a probability $1 - p$ of corrupting it, then the optimum rate of information transfer is given by the formula

$$1 + p \log_2 p + (1 - p) \log_2 (1 - p).$$

The graph in Figure 1.6 shows this rate plotted as a function of p. From the graph you can see that as the accuracy, p, approaches 1, the transmission rate also approaches 1: for a perfectly accurate channel you do not have to waste any space trying to protect your message from corruption.

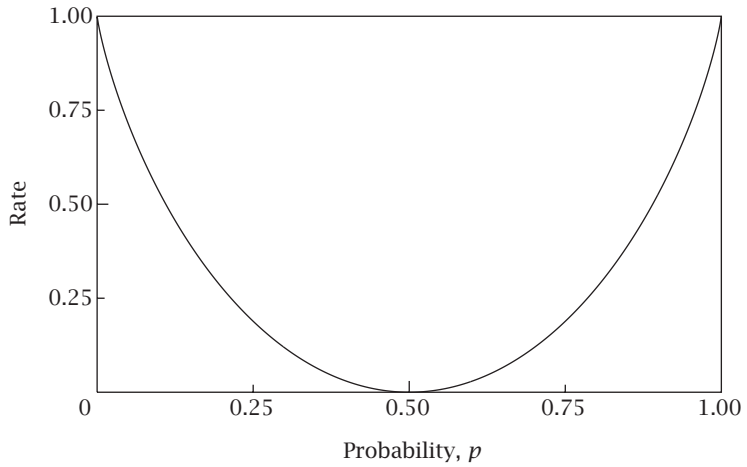

Figure 1.6. Optimal transmission rate.

You might at first be surprised that as p approaches 0 it is also possible to transmit at a very high rate. Surely, as the channel becomes less and less accurate it should be harder to transmit, not easier. The point here is that p does not measure accuracy in the colloquial sense: it measures the probability that the channel will switch your bit from 0 to 1 or vice versa. If $p = 0$, then the channel is guaranteed to change each piece of information to its opposite. As long as you are aware of what the channel is doing, you can recover the original information perfectly, by switching everything back—just as if you were printing a photograph from a negative. The worst value of p is $1/2$: right in the middle. When $p = 1/2$, the channel is just as likely to send 0 or 1, whatever the original message. Each bit that is received is completely random, regardless of what was sent. Clearly, in this situation it is impossible ever to send any information. That is reflected in the fact that the graph touches the horizontal axis at $p = 1/2$; for this value of p the transmission rate is 0.

The formula for the transmission rate is more usually written $1 - E(p)$, where

$$E(p) = -p \log_2 p - (1 - p) \log_2 (1 - p).$$

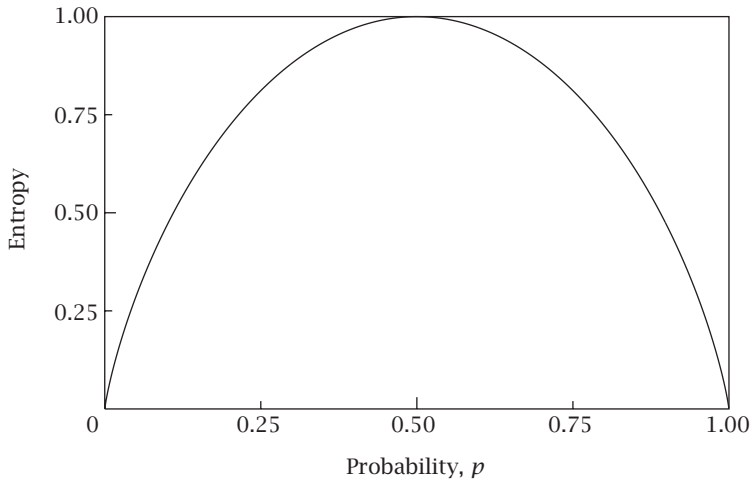

Figure 1.7. The entropy curve.

If you plot the quantity $E(p)$ against p, you get the previous graph turned upside down as in Figure 1.7. The quantity $E(p)$ is known as the *entropy* of the channel and measures the uncertainty of the channel's operation. It is 0 if p is 0 or 1, and is maximum when $p = 1/2$. Probabilities 0 or 1 give complete certainty: whatever you receive, you know for sure what was sent. Probability $1/2$ gives you maximum *un*certainty: the message you receive gives you no idea what was sent. Entropy is one of the fundamental concepts in the branch of mathematics known as information theory. It will reappear in Chapter 7, where the idea of uncertainty is explored in more depth.

1.7 Further Reading

There are several excellent text books on coding theory. I personally like the one by van Lint, which is referenced below. Most of them assume some knowledge of university-level mathematics; in particular linear algebra and group theory. At a more recreational level, there is a video lecture by Peter Cameron, which is available from the London Mathematical Society at http://www.lms.ac.uk.

As was explained in the text, the codes discussed in this chapter have nothing to do with cryptography (the art of communicating information secretly). Cryptography is a flourishing branch of mathematics with numerous applications to the security of financial transactions as well as the more traditional military uses. A popular book on the subject is the one by Piper and Murphy, referenced below.

Shannon's work in the late 1940s gave rise to two new branches or sub-branches of mathematics: coding theory and information theory. Shannon's own articles are highly readable by the standards of research mathematics, but the casual reader might not consider that a very strong recommendation.

Chapter Bibliography

Piper, F. and Murphy, S. 2002 *Cryptography: a very short introduction*. Oxford University Press.

van Lint, J. H. 1999 *Introduction to coding theory*, 3rd edn. Graduate Texts in Mathematics, vol. 86. Springer.

1.8 SOLUTIONS

Solution 1.1. A solution is given in the text. □

Solution 1.2. The string

$$0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1$$

violates all three parity rules of the 7-bit Hamming code, so the bit that needs to be changed is bit number 7. The nearest code word is thus

$$0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0,$$

which is the fourth entry in Table 1.2. □

Solution 1.3. Among the 16 codewords of the 7-bit Hamming code, 8 of them have an even number of 1s in them and the other 8 have an odd number of 1s. Any two of the 'even' words must differ in an even number of places. We already know that they differ in at least three places. So any two *even* words must

differ in at least four places. We can thus extract a 4-separated code from the Hamming code by using the even words:

$$\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1.
\end{array}$$

This code can be described using a parity-check matrix just by taking the matrix for the Hamming code and adding an extra row (shown here at the top) which checks that the words have an even number of 1s:

b_1	b_2	b_3	b_4	b_5	b_6	b_7
1	1	1	1	1	1	1
1	0	1	0	1	0	1
0	1	1	0	0	1	1
0	0	0	1	1	1	1

□

Solution 1.4. The 15-bit Hamming code is constructed using the parity matrix

1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	**1**	1	0	0	1	1	0	0	1	1	0	0	1	1
0	0	0	**1**	1	1	1	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	**1**	1	1	1	1	1	1	1

in which the 15 columns are the binary representations of the numbers from 1 to 15. (The analogous construction works for each number of the form $2^n - 1$.) Of the 2^{15} possible strings,

$2^{11} = 2048$ are codewords. You can check this using the same argument as for the 7-bit code. This time the crucial bits are b_1, b_2, b_4 and b_8, each of which appears in only one row of the parity-check matrix (in which they are highlighted). □

Solution 1.5. To check that the general Hamming code is 3-separated, we need a new argument since we do not want to check all possible pairs of codewords. (There are over two million such pairs in the case of the 15-bit Hamming code.) We want to check that any two codewords differ in at least three places. To do so, let us eliminate, in turn, the other possibilities: that two codewords could differ in just one place or in just two places.

If you start with a codeword and change just one bit, then the resulting string will violate any parity rule that involves the bit you have changed. So it cannot be a codeword.

If we could get from one codeword to another in *two* steps, then there would be a string in the middle, which would be just one step away from each of the *two* codewords. The offending string would be decodable in two different ways. However, the decoding procedure in Section 1.5 shows that this cannot happen. We saw that each string which is not a codeword can be converted into a codeword by changing a single bit; but the argument actually showed that we have no choice about which bit to change. Once you have identified which rules are violated by the string, you know which column to look for in the parity-check matrix. Since there is only one way to decode each string, there cannot be two different codewords adjacent to the same string. □