5

Error-Correcting Codes: Mistakes That Fix Themselves

> It is one thing to show a man that he is in an error, and another to put him in possession of truth.
>
> —John Locke, *Essay Concerning Human Understanding* (1690)

These days, we're used to accessing computers whenever we need them. Richard Hamming, a researcher working at the Bell Telephone Company labs in the 1940s, was not so lucky: the company computer he needed was used by other departments and available to him only on weekends. You can imagine his frustration, therefore, at the crashes that kept recurring due to the computer's errors in reading its own data. Here is what Hamming himself had to say on the matter:

> Two weekends in a row I came in and found that all my stuff had been dumped and nothing was done. I was really aroused and annoyed because I wanted those answers and two weekends had been lost. And so I said, "Dammit, if the machine can detect an error, why can't it locate the position of the error and correct it?"

There can be few more clear-cut cases of necessity being the mother of invention. Hamming had soon created the first ever *error-correcting code*: a seemingly magical algorithm that detects and corrects errors in computer data. Without these codes, our computers and communication systems would be drastically slower, less powerful, and less reliable than they are today.

THE NEED FOR ERROR DETECTION AND CORRECTION

Computers have three fundamental jobs. The most important job is to perform computations. That is, given some input data, the computer must transform the data in some way to produce a useful

answer. But the ability to compute answers would be essentially useless without the other two very important jobs that computers perform: storing data and transmitting data. (Computers mostly store data in their memory and on disk drives. And they typically transmit data over the internet.) To emphasize this point, imagine a computer that could neither store nor transmit information. It would, of course, be almost useless. Yes, you could do some complex computations (for example, preparing an intricate financial spreadsheet detailing the budget for a company), but then you would be unable to send the results to a colleague or even to save the results so you could come back and work on them later. Therefore, transmission and storage of data are truly essential for modern computers.

But there is a huge challenge associated with transmitting and storing data: the data must be *exactly right*—because in many cases, even one tiny mistake can render the data useless. As humans, we are also familiar with the need to store and transmit information without any errors. For example, if you write down someone's phone number, it is essential that every digit is recorded correctly and in the right order. If there is even one mistake in one of the digits, the phone number is probably useless to you or anyone else. And in some cases, errors in data can actually be *worse* than useless. For example, an error in the file that stores a computer program can make that program crash or do things it was not intended to. (It might even delete some important files or crash before you get a chance to save your work.) And an error in some computerized financial records could result in actual monetary loss (if, say, you thought you were buying a stock priced at $5.34 per share but the actual cost was $8.34).

But, as humans, the amount of error-free information we need to store is relatively small, and it's not too hard to avoid mistakes just by checking carefully whenever you know some information is important—things like bank account numbers, passwords, e-mail addresses, and the like. On the other hand, the amount of information that computers need to store and transmit without making any errors is absolutely immense. To get some idea of the scale, consider this. Suppose you have some kind of computing device with a storage capacity of 100 gigabytes. (At the time of writing, this is the typical capacity of a low-cost laptop.) This 100 gigabytes is equivalent to about 15 million pages of text. So even if this computer's storage system makes just one error per million pages, there would still be (on average) 15 mistakes on the device when filled to capacity. And the same lesson applies to transmission of data too: if you download a 20-megabyte software program, and your computer misinterprets just one in every million characters it receives, there will

probably still be over 20 errors in your downloaded program—every one of which could cause a potentially costly crash when you least expect it.

The moral of the story is that, for a computer, being accurate 99.9999% of the time is not even close to good enough. Computers must be able to store and transmit literally billions of pieces of information without making a single mistake. But computers have to deal with communication problems just like other devices. Telephones are a good example here: it's obvious that they don't transmit information perfectly, because phone conversations often suffer from distortions, static, or other types of noise. But telephones are not alone in their suffering: electrical wires are subject to all sorts of fluctuations; wireless communications suffer interference all the time; and physical media such as hard disks, CDs, and DVDs can be scratched, damaged, or simply misread because of dust or other physical interference. How on earth can we hope to achieve an error rate of less than one in many billions, in the face of such obvious communication errors? This chapter will reveal the ideas behind the ingenious computer science that makes this magic happen. It turns out that if you use the right tricks, even extremely unreliable communication channels can be used to transmit data with incredibly low error rates—so low that in practice, errors can be completely eliminated.

THE REPETITION TRICK

The most fundamental trick for communicating reliably over an unreliable channel is one that we are all familiar with: to make sure that some information has been communicated correctly, you just need to repeat it a few times. If someone dictates a phone number or bank account number to you over a bad telephone connection, you will probably ask them to repeat it at least once to make sure there were no mistakes.

Computers can do exactly the same thing. Let's suppose a computer at your bank is trying to transmit your account balance to you over the internet. Your account balance is actually $5213.75, but unfortunately the network is not very reliable and every single digit has a 20% chance being changed to something else. So the first time your balance is transmitted, it might arrive as $5293.75. Obviously, you have no way of knowing whether or not this is correct. All of the digits *might* be right, but one or more of them might be wrong and you have no way of telling. But by using the repetition trick, you can make a very good guess as to the true balance. Imagine that you

ask for your balance to be transmitted five times, and receive the following responses:

transmission 1:	$	5	2	9	3	.	7	5
transmission 2:	$	5	2	1	3	.	7	5
transmission 3:	$	5	2	1	3	.	1	1
transmission 4:	$	5	4	4	3	.	7	5
transmission 5:	$	7	2	1	8	.	7	5

Notice that some of the transmissions have more than one digit wrong, and there's even one transmission (number 2) with no errors at all. The crucial point is that you have no way of knowing where the errors are, so there is no way you can pick out transmission 2 as being the correct one. Instead, what you can do is examine each digit separately, looking at all transmissions of that one digit, and pick the value that occurs most often. Here are the results again, with the most common digits listed at the end:

transmission 1:	$	5	2	9	3	.	7	5
transmission 2:	$	5	2	1	3	.	7	5
transmission 3:	$	5	2	1	3	.	1	1
transmission 4:	$	5	4	4	3	.	7	5
transmission 5:	$	7	2	1	8	.	7	5
most common digit:	$	5	2	1	3	.	7	5

Let's look at some examples to make the idea absolutely clear. Examining the first digit in the transmission, we see that in transmissions 1–4, the first digit was a 5, whereas in transmission 5, the first digit was a 7. In other words, four of the transmissions said "5" and only one said "7." So although you can't be absolutely sure, the *most likely* value for the first digit of your bank balance is 5. Moving on to the second digit, we see that 2 occurred four times, and 4 only once, so 2 is the most likely second digit. The third digit is a bit more interesting, because there are three possibilities: 1 occurs three times, 9 occurs once, and 4 occurs once. But the same principle applies, and 1 is the most likely true value. By doing this for all the digits, you can arrive at a final guess for your complete bank balance: $5213.75, which in this case is indeed correct.

Well, that was easy. Have we solved the problem already? In some ways, the answer is yes. But you might be a little dissatisfied because of two things. First, the error rate for this communication channel was only 20%, and in some cases computers might need to communicate over channels that are much worse than that. Second, and perhaps more seriously, the final answer happened to be correct in

the above example, but there is no guarantee that the answer will always be right: it is just a guess, based on what we think is most likely to be the true bank balance. Luckily, both of these objections can be addressed very easily: we just increase the number of retransmissions until the reliability is as high as we want.

For example, suppose the error rate was 50% instead of the 20% in the last example. Well, you could ask the bank to transmit your balance 1000 times instead of just 5. Let's concentrate on just the first digit, since the others work out the same way. Since the error rate is 50%, about half of them will be transmitted correctly, as a 5, and the other half will be changed to some other random values. So there will be about 500 occurrences of 5, and only about 50 each of the other digits (0–4 and 6–9). Mathematicians can calculate the chances of one of the other digits coming up more often than the 5: it turns out that even if we transmitted a new bank balance every second using this method, we would have to wait many trillions of years before we expect to make a wrong guess for the bank balance. The moral of the story is that by repeating an unreliable message often enough, you can make it as reliable as you want. (In these examples, we assumed the errors occur *at random*. If, on the other hand, a malicious entity is deliberately interfering with the transmission and choosing which errors to create, the repetition trick is much more vulnerable. Some of the codes introduced later work well even against this type of malicious attack.)

So, by using the repetition trick, the problem of unreliable communication can be solved, and the chance of a mistake essentially eliminated. Unfortunately, the repetition trick is not good enough for modern computer systems. When transmitting a small piece of data like a bank balance, it is not too costly to retransmit 1000 times, but it would obviously be completely impractical to transmit 1000 copies of a large (say, 200-megabyte) software download. Clearly, computers need to use something more sophisticated than the repetition trick.

THE REDUNDANCY TRICK

Even though computers don't use the repetition trick as it was described above, we covered it first so that we could see the most basic principle of reliable communication in action. This basic principle is that you can't just send the original message; you need to send something extra to increase the reliability. In the case of the repetition trick, the extra thing you send is just more copies of the original message. But it turns out there are many other types of extra stuff

you can send to improve the reliability. Computer scientists call the extra stuff "redundancy." Sometimes, the redundancy is added on to the original message. We'll see this "adding on" technique when we look at the next trick (the checksum trick). But first, we will look at another way of adding redundancy, which actually transforms the original message into a longer "redundant" one—the original message is deleted and replaced by a different, longer one. When you receive the longer message, you can then transform it back into the original, even if it has been corrupted by a poor communication channel. We'll call this simply the *redundancy trick*.

An example will make this clear. Recall that we were trying to transmit your bank balance of $5213.75 over an unreliable communication channel that randomly altered 20% of the digits. Instead of trying to transmit just "$5213.75," let's transform this into a longer (and therefore "redundant") message that contains the same information. In this case, we'll simply spell out the balance in English words, like this:

```
five two one three point seven five
```

Let's again suppose that about 20% of the characters in this message get flipped randomly to something else due to a poor communication channel. The message might end up looking something like this:

```
fiqe kwo one thrxp point sivpn fivq
```

Although it's a little annoying to read, I think you will agree that anyone who knows English can guess that this corrupted message represents the true bank balance of $5213.75.

The key point is that because we used a *redundant* message, it is possible to reliably detect and correct any single change to the message. If I tell you that the characters "fiqe" represent a number in English and that only one character has been altered, you can be absolutely certain that the original message was "five," because there is no other English number that can be obtained from "fiqe" by altering only one character. In stark contrast, if I tell you that the digits "367" represent a number but one of the digits has been altered, you have no way of knowing what the original number was, because there is no redundancy in this message.

Although we haven't yet explored exactly how redundancy works, we have already seen that it has something to do with making the message *longer*, and that each part of the message should conform to some kind of well-known *pattern*. In this way, any single change can be first identified (because it does not fit in with a known pattern) and then corrected (by changing the error to fit with the pattern).

Computer scientists call these known patterns "code words." In our example, the code words are just numbers written in English, like "one," "two," "three," and so on.

Now it's time to explain exactly how the redundancy trick works. Messages are made up of what computer scientists call "symbols." In our simple example, the symbols are the numeric digits 0–9 (we'll ignore the dollar sign and decimal point to make things even easier). Each symbol is assigned a code word. In our example, the symbol 1 is assigned the code word "one," 2 is assigned "two," and so on.

To transmit a message, you first take each symbol and translate it into its corresponding code word. Then you send the transformed message over the unreliable communication channel. When the message is received, you look at each part of a message and check whether it is a valid code word. If it is valid (e.g., "five"), you just transform it back into the corresponding symbol (e.g., 5). If it is not a valid code word (e.g., "fiqe"), you find out which code word it matches most closely (in this case, "five"), and transform *that* into the corresponding symbol (in this case, 5). Examples of using this code are shown in the figure above.

Encoding

1	→	one
2	→	two
3	→	three
4	→	four
5	→	five

Decoding

five	→	5	(exact match)
fiqe	→	5	(closest match)
twe	→	2	(closest match)

A code using English words for digits.

That's really all there is to it. Computers actually use this redundancy trick all the time to store and transmit information. Mathematicians have worked out fancier codewords than the English-language ones we were using as an example, but otherwise the workings of reliable computer communication are the same. The figure on the facing page gives a real example. This is the code computer scientists call the $(7,4)$ Hamming code, and it is one of the codes discovered by Richard Hamming at Bell Labs in 1947, in response to the weekend computer crashes described earlier. (Because of Bell's requirement that he patent the codes, Hamming did not publish them until three years later, in 1950.) The most obvious difference to our previous code is that everything is done in terms of zeros and ones. Because every piece of data stored or transmitted by a

computer is converted into strings of zeros and ones, any code used in real life is restricted to just these two digits.

Encoding

0000 → 0000000
0001 → 0001011
0010 → 0010111
0011 → 0011100
0100 → 0100110

Decoding

0010111 → 0010 (exact match)
0010110 → 0010 (closest match)
1011100 → 0011 (closest match)

A real code used by computers. Computer scientists call this code the $(7, 4)$ Hamming code. Note that the "Encoding" box lists only five of the 16 possible 4-digit inputs. The remaining inputs also have corresponding code words, but they are omitted here.

But apart from that, everything works exactly the same as before. When encoding, each group of four digits has redundancy added to it, generating a code word of seven digits. When decoding, you first look for an exact match for the seven digits you received, and if that fails, you take the closest match. You might be worried that, now we are working with only ones and zeros, there might be more than one equally close match and you could end up choosing the wrong decoding. However, this particular code has been designed cunningly so that any single error in a 7-digit codeword can be corrected unambiguously. There is some beautiful mathematics behind the design of codes with this property, but we won't be pursuing the details here.

It's worth emphasizing why the redundancy trick is preferred to the repetition trick in practice. The main reason is the relative *cost* of the two tricks. Computer scientists measure the cost of error-correction systems in terms of "overhead." Overhead is just the amount of extra information that needs to be sent to make sure a message is received correctly. The overhead of the repetition trick is enormous, since you have to send several entire copies of the message. The overhead of the redundancy trick depends on the exact set of code words that you use. In the example above that used English words, the redundant message was 35 characters long, whereas the original message consisted of only 6 numeric digits, so the overhead of this particular application of the redundancy trick is also quite large. But mathematicians have worked out sets of code words that have much lower redundancy, yet still get incredibly high performance in terms of the chance of an error going undetected. The low

overhead of these code words is the reason that computers use the redundancy trick instead of the repetition trick.

The discussion so far has used examples of *transmitting* information using codes, but everything we have discussed applies equally well to the task of *storing* information. CDs, DVDs, and computer hard drives all rely heavily on error-correcting codes to achieve the superb reliability we observe in practice.

THE CHECKSUM TRICK

So far, we've looked at ways to simultaneously *detect* and *correct* errors in data. The repetition trick and the redundancy trick are both ways of doing this. But there's another possible approach to this whole problem: we can forget about *correcting* errors and concentrate only on *detecting* them. (The 17th-century philosopher John Locke was clearly aware of the distinction between error detection and error correction—as you can see from the opening quotation of this chapter.) For many applications, merely detecting an error is sufficient, because if you detect an error, you just request another copy of the data. And you can keep on requesting new copies, until you get one that has no errors in it. This is a very frequently used strategy. For example, almost all internet connections use this technique. We'll call it the "checksum trick," for reasons that will become clear very soon.

To understand the checksum trick, it will be more convenient to pretend that all of our messages consist of numbers only. This is a very realistic assumption, since computers store all information in the form of numbers and only translate the numbers into text or images when presenting that information to humans. But, in any case, it is important to understand that any particular choice of symbols for the messages does not affect the techniques described in this chapter. Sometimes it is more convenient to use numeric symbols (the digits 0–9), and sometimes it is more convenient to use alphabetic symbols (the characters a–z). But in either case, we can agree on some translation between these sets of symbols. For example, one obvious translation from alphabetic to numeric symbols would be $a \rightarrow 01$, $b \rightarrow 02, \ldots, z \rightarrow 26$. So it really doesn't matter whether we investigate a technique for transmitting numeric messages or alphabetic messages; the technique can later be applied to any type of message by first doing a simple, fixed translation of the symbols.

At this point, we have to learn what a checksum actually is. There are many different types of checksums, but for now we will stick with the least complicated type, which we'll call a "simple checksum."

Computing the simple checksum of a numeric message is really, really easy: you just take the digits of the message, add them all up, throw away everything in the result except for the last digit, and the remaining digit is your simple checksum. Here's an example: suppose the message is

4 6 7 5 6

Then the sum all the digits is $4 + 6 + 7 + 5 + 6 = 28$, but we keep only the last digit, so the simple checksum of this message is 8.

But how are checksums used? That's easy: you just append the checksum of your original message to the end of the message before you send it. Then, when the message is received by someone else, they can calculate the checksum again, compare it with the one you sent, and see if it is correct. In other words, they "check" the "sum" of the message—hence the terminology "checksum." Let's stick with the above example. The simple checksum of the message "46756" is 8, so we transmit the message and its checksum as

4 6 7 5 6 8

Now, the person receiving the message has to know that you are using the checksum trick. Assuming they do know, they can immediately recognize that the last digit, the 8, is not part of the original message, so they put it to one side and compute the checksum of everything else. If there were no errors in the transmission of the message, they will compute $4 + 6 + 7 + 5 + 6 = 28$, keep the last digit (which is 8), check that it is equal to the checksum they put aside earlier (which it is), and therefore conclude that the message was transmitted correctly. On the other hand, what happens if there *was* an error in transmitting the message? Suppose the 7 was randomly changed to a 3. Then you would receive the message

4 6 3 5 6 8

You would set aside the 8 for later comparison and compute the checksum as $4 + 6 + 3 + 5 + 6 = 24$, keeping only the last digit (4). This is *not* equal to the 8 that was set aside earlier, so you would be sure that the message was corrupted during transmission. At this point, you request that the message is retransmitted, wait until you receive a new copy, then again compute and compare the checksum. And you can keep doing this until you get a message whose checksum is correct.

All of this seems almost too good to be true. Recall that the "overhead" of an error-correcting system is the amount of extra information you have to send in addition to the message itself. Well, here

we seem to have the ultimate low-overhead system, since no matter how long the message is, we only have to add one extra digit (the checksum) to detect an error!

Alas, it turns out that this system of simple checksums *is* too good to be true. Here is the problem: the simple checksum described above can detect at most *one* error in the message. If there are two or more errors, the simple checksum might detect them, but then again it might not. Let's look at some examples of this:

						checksum
original message	4	6	7	5	6	8
message with one error	1	6	7	5	6	5
message with two errors	1	5	7	5	6	4
message with two (different) errors	2	8	7	5	6	8

The original message (46756) is the same as before, and so is its checksum (8). In the next line is a message with one error (the first digit is a 1 instead of a 4), and the checksum turns out to be 5. In fact, you can probably convince yourself that changing any *single* digit results in a checksum that differs from 8, so you are guaranteed to detect any single mistake in the message. It's not hard to prove that this is always true: if there is only one error, a simple checksum is absolutely guaranteed to detect it.

In the next line of the table, we see a message with two errors: each of the first two digits has been altered. In this case, the checksum happens to be 4. And since 4 is different from the original checksum, which was 8, the person receiving this message would in fact detect that an error had been made. However, the crunch comes in the last line of the table. Here is another message with two errors, again in the first two digits. But the values are different, and it so happens that the checksum for this two-error message is 8—the same as the original! So a person receiving this message would fail to detect that there are errors in the message.

Luckily, it turns out that we can get around this problem by adding a few more tweaks to our checksum trick. The first step is to define a new type of checksum. Let's call it a "staircase" checksum because it helps to think of climbing a staircase while computing it. Imagine

you are at the bottom of a staircase with the stairs numbered 1, 2, 3, and so on. To calculate a staircase checksum, you add up the digits just like before, but each digit gets multiplied by the number of the stair you are on, and you have to move up one step for each digit. At the end, you throw away everything except the last digit, just as with the simple checksum. So if the message is

4 6 7 5 6

like before, then the staircase checksum is calculated by first calculating the staircase sum

$$\begin{aligned}(1 \times 4) + (2 \times 6) + (3 \times 7) + (4 \times 5) + (5 \times 6) \\ &= 4 + 12 + 21 + 20 + 30 \\ &= 87\end{aligned}$$

Then throw away everything except the last digit, which is 7. So the staircase checksum of "46756" is 7.

What is the point of all this? Well, it turns out that if you include *both* the simple and staircase checksums, then you are guaranteed to detect any two errors in any message. So our new checksum trick is to transmit the original message, then two extra digits: the simple checksum first, then the staircase checksum. For example, the message "46756" would now be transmitted as

4 6 7 5 6 8 7

When you receive the message, you again have to know by prior agreement exactly what trick has been applied. But assuming you do know, it is easy to check for errors just as with the simple checksum trick. In this case you first set aside the last two digits (the 8, which is the simple checksum, and the 7, which is the staircase checksum). You then compute the simple checksum of the rest of the message (46756, which comes to 8), and you compute the staircase checksum too (which comes to 7). If *both* the computed checksum values match the ones that were sent (and in this case they do), you are guaranteed that the message is either correct, or has at least three errors.

The next table shows this in practice. It is identical to the previous table except that the staircase checksum has been added to each row, and a new row has been added as an extra example. When there is one error, we find that both the simple and staircase checksums differ from the original message (5 instead of 8, and 4 instead of 7). When there are two errors, it is possible for both checksums to differ, as in the third row of the table where we see 4 instead of 8, and 2 instead of 7. But as we already found out, sometimes the simple checksum

will not change when there are two errors. The fourth row shows an example, where the simple checksum is still 8. But because the staircase checksum differs from the original (9 instead of 7), we still know that this message has errors. And in the last row, we see that it can work out the other way around too: here is an example of two errors that results in a different simple checksum (9 instead of 8), but the same staircase checksum (7). But, again, the point is that we can still detect the error because at least one of the two checksums differs from the original. And although it would take some slightly technical math to prove it, this is no accident: it turns out that you will always be able to detect the errors if there are no more than two of them.

						simple and staircase checksums
original message	4	6	7	5	6	8 7
message with one error	1	6	7	5	6	5 4
message with two errors	1	5	7	5	6	4 2
message with two (different) errors	2	8	7	5	6	8 9
message with two (again different) errors	6	5	7	5	6	9 7

Now that we have a grasp of the fundamental approach, we need to be aware that the checksum trick just described is guaranteed to work only for relatively short messages (fewer than 10 digits). But very similar ideas can be applied to longer messages. It is possible to define checksums by certain sequences of simple operations like adding up the digits, multiplying the digits by "staircases" of various shapes, and swapping some of the digits around according to a fixed pattern. Although that might sound complicated, computers can compute these checksums blindingly fast and it turns out to be an extremely useful, practical way of detecting errors in a message.

The checksum trick described above produces only two checksum digits (the simple digit and the staircase digit), but real checksums usually produce many more digits than that—sometimes as many

as 150 digits. (Throughout the remainder of this chapter, I am talking about the ten *decimal* digits, 0–9, not the two *binary* digits, 0 and 1, which are more commonly used in computer communication.) The important point is that the number of digits in the checksum (whether 2, as in the example above, or about 150, as for some checksums used in practice) is *fixed*. But although the length of the checksums produced by any given checksum algorithm is fixed, you can compute checksums of messages that are as long as you want. So for very long messages, even a relatively large checksum like 150 digits ends up being minuscule in proportion to the message itself. For example, suppose you use a 100-digit checksum to verify the correctness of a 20-megabyte software package downloaded from the web. The checksum is less than one-thousandth of 1% of the size of the software package. I'm sure you would agree this is an acceptable level of overhead! And a mathematician will tell you that the chance of failing to detect an error when using a checksum of this length is so incredibly tiny that it is for all practical purposes impossible.

As usual, there are a few important technical details here. It's not true that any 100-digit checksum system has this incredibly high resistance to failure. It requires a certain type of checksum that computer scientists call a *cryptographic hash function*—especially if the changes to the message might be made by a malicious opponent, instead of the random vagaries of a poor communication channel. This is a very real issue, because it is possible that an evil hacker might try to alter that 20-megabyte software package in such a way that it has the same 100-digit checksum, but is actually a different piece of software that will take control of your computer! The use of cryptographic hash functions eliminates this possibility.

THE PINPOINT TRICK

Now that we know about checksums, we can go back to the original problem of both detecting *and* correcting communication errors. We already know how to do this, either inefficiently using the repetition trick or efficiently using the redundancy trick. But let's return to this now, because we never really found out how to create the code words that form the key ingredient in this trick. We did have the example of using English words to describe numerals, but this particular set of code words is less efficient than the ones computers actually use. And we also saw the real example of a Hamming code, but without any explanation of how the code words were produced in the first place.

So now we will learn about another possible set of code words that can be used to perform the redundancy trick. Because this is a very special case of the redundancy trick that allows you to quickly pinpoint an error, we'll call this the "pinpoint trick."

Just as we did with the checksum trick, we will work entirely with numerical messages consisting of the digits 0–9, but you should keep in mind that this is just for convenience. It is very simple to take an alphabetical message and translate it into numbers, so the technique described here can be applied to any message whatsoever.

To keep things simple, we'll assume that the message is exactly 16 digits long, but, again, this doesn't limit the technique in practice. If you have a long message, just break it into 16-digit chunks and work with each chunk separately. If the message is shorter than 16 digits, fill it up with zeroes, until it is 16 digits long.

The first step in the pinpoint trick is to rearrange the 16 digits of the message into a square that reads left to right, top to bottom. So if the actual message is

4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7

it gets rearranged into

4	8	3	7
5	4	3	6
2	2	5	6
3	9	9	7

Next, we compute a simple checksum of each row and add it to the right-hand side of the row:

4	8	3	7	2
5	4	3	6	8
2	2	5	6	5
3	9	9	7	8

These simple checksums are computed just like before. For example, to get the second row checksum you compute $5 + 4 + 3 + 6 = 18$ and then take the last digit, which is 8.

The next step in the pinpoint trick is to compute simple checksums for each column and add these in a new row at the bottom:

4	8	3	7	2
5	4	3	6	8
2	2	5	6	5
3	9	9	7	8
4	3	0	6	

Again, there's nothing mysterious about the simple checksums. For example, the third column is computed from $3+3+5+9 = 20$, which becomes 0 when we take the last digit.

The next step in the pinpoint trick is to reorder everything so it can be stored or transmitted one digit at a time. You do this in the obvious way, reading digits from left to right, top to bottom. So we end up with the following 24-digit message:

4 8 3 7 2 5 4 3 6 8 2 2 5 6 5 3 9 9 7 8 4 3 0 6

Now imagine you have received a message that has been transmitted using the pinpoint trick. What steps do you follow to work out the original message and correct any communication errors? Let's work through an example. The original 16-digit message will be the same as the one above, but to make things interesting, suppose there was a communication error and one of the digits was altered. Don't worry about which is the altered digit yet—we will be using the pinpoint trick to determine that very shortly.

So let's suppose the 24-digit message you *received* is

4 8 3 7 2 5 4 3 6 8 2 7 5 6 5 3 9 9 7 8 4 3 0 6

Your first step will be to lay the digits out in a 5-by-5 square, recognizing that the last column and last row correspond to checksum digits that were sent with the original message:

4	8	3	7	2
5	4	3	6	8
2	7	5	6	5
3	9	9	7	8
4	3	0	6	

Next, compute simple checksums of the first four digits in each row and column, recording the results in a newly created row and column next to the checksum values that you received:

4	8	3	7	2	2
5	4	3	6	8	8
2	7	5	6	5	0
3	9	9	7	8	8
4	3	0	6		
4	8	0	6		

It is crucial to bear in mind that there are two sets of checksum values here: the ones you were *sent*, and the ones you *calculated*. Mostly, the two sets of values will be the same. In fact, if they are all identical, you

can conclude that the message is very likely correct. But if there was a communication error, some of the calculated checksum values will differ from the sent ones. Notice that in the current example, there are two such differences: the 5 and 0 in the third row differ, and so do the 3 and 8 in the second column. The offending checksums are highlighted in boxes:

4	8	3	7	2	2
5	4	3	6	8	8
2	7	5	6	5	[0]
3	9	9	7	8	8
4	3	0	6		
4	[8]	0	6		

Here is the key insight: the location of these differences tells you exactly where the communication error occurred! It *must* be in the third row (because every other row had the correct checksum), and it *must* be in the second column (because every other column had the correct checksum). And as you can see from the following diagram, this narrows it down to exactly one possibility—the 7 highlighted in a solid box:

4	8	3	7	2	2
5	4	3	6	8	8
2	[7]	5	6	5	[0]
3	9	9	7	8	8
4	3	0	6		
4	[8]	0	6		

But that's not all—we have located the error, but not yet corrected it. Fortunately, this is easy: we just have to replace the erroneous 7 with a number that will make both of the checksums correct. We can see that the second column was meant to have a checksum of 3, but it came out to 8 instead—in other words, the checksum needs to be reduced by 5. So let's reduce the erroneous 7 by 5, which leaves 2:

4	8	3	7	2	2
5	4	3	6	8	8
2	[2]	5	6	5	[5]
3	9	9	7	8	8
4	3	0	6		
4	[3]	0	6		

You can even double-check this change, by examining the third row—it now has a checksum of 5, which agrees with the received checksum. The error has been both located and corrected! The final obvious step is to extract the corrected original 16-digit message from the 5-by-5 square, by reading top to bottom, left to right (and ignoring the final row and column, of course). This gives

4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7

which really is the same message that we started with.

In computer science, the pinpoint trick goes by the name of "two-dimensional parity." The word *parity* means the same thing as a simple checksum, when working with the binary numbers computers normally use. And the parity is described as *two-dimensional* because the message gets laid out in a grid with two dimensions (rows and columns). Two-dimensional parity has been used in some real computer systems, but it is not as effective as certain other redundancy tricks. I chose to explain it here because it is very easy to visualize and conveys the flavor of how one can both find and correct errors without requiring the sophisticated math behind the codes popular in today's computer systems.

ERROR CORRECTION AND DETECTION IN THE REAL WORLD

Error-correcting codes sprang into existence in the 1940s, rather soon after the birth of the electronic computer itself. In retrospect, it's not hard to see why: early computers were rather unreliable, and their components frequently produced errors. But the true roots of error-correcting codes lie even earlier, in communication systems such as telegraphs and telephones. So it is not altogether surprising that the two major events triggering the creation of error-correcting codes both occurred in the research laboratories of the Bell Telephone Company. The two heroes of our story, Claude Shannon and Richard Hamming, were both researchers at Bell Labs. Hamming we have met already: it was his annoyance at the weekend crashes of a company computer that led directly to his invention of the first error-correcting codes, now known as Hamming codes.

However, error-correcting codes are just one part of a larger discipline called *information theory*, and most computer scientists trace the birth of the field of information theory to a 1948 paper by Claude Shannon. This extraordinary paper, entitled "The Mathematical Theory of Communication," is described in one biography of Shannon as "the Magna Carta of the information age." Irving Reed

(co-inventor of the Reed–Solomon codes mentioned below) said of the same paper: "Few other works of this century have had greater impact on science and engineering. By this landmark paper ... he has altered most profoundly all aspects of communication theory and practice." Why the high praise? Shannon demonstrated through mathematics that it was possible, in principle, to achieve surprisingly high rates of error-free communication over a noisy, error-prone link. It was not until many decades later that scientists came close to achieving Shannon's theoretical maximum communication rate in practice.

Incidentally, Shannon was apparently a man of extremely diverse interests. As one of the four main organizers of the 1956 Dartmouth AI conference (discussed at the end of chapter 6), he was intimately involved in the founding of another field: artificial intelligence. But it doesn't stop there. He also rode unicycles and built an improbable-sounding unicycle with an elliptical (i.e., noncircular) wheel, meaning that the rider moved up and down as the unicycle moved forward!

Shannon's work placed Hamming codes in a larger theoretical context and set the stage for many further advances. Hamming codes were thus used in some of the earliest computers and are still widely used in certain types of memory systems. Another important family of codes is known as the *Reed–Solomon* codes. These codes can be adapted to correct for a large number of errors per codeword. (Contrast this with the (7, 4) Hamming code in the figure on page 67, which can correct only one error in each 7-digit code word.) Reed–Solomon codes are based on a branch of mathematics called finite field algebra, but you can think of them, very roughly, as combining the features of the staircase checksum and the two-dimensional pinpoint trick. They are used in CDs, DVDs, and computer hard drives.

Checksums are also widely used in practice, typically for detecting rather than correcting errors. Perhaps the most pervasive example is Ethernet, the networking protocol used by almost every computer on the planet these days. Ethernet employs a checksum called CRC-32 to detect errors. The most common internet protocol, called TCP (for Transmission Control Protocol), also uses checksums for each chunk, or *packet*, of data that it sends. Packets whose checksums are incorrect are simply discarded, because TCP is designed to automatically retransmit them later if necessary. Software packages published on the internet are often verified using checksums; popular ones include a checksum called MD5, and another called SHA-1. Both are intended to be cryptographic hash functions, providing protection against malicious alteration of the software as well as random communication errors. MD5 checksums have about 40 digits,

SHA-1 produces about 50 digits, and there are some even more error-resistant checksums in the same family, such as SHA-256 (about 75 digits) and SHA-512 (about 150 digits).

The science of error-correcting and error-detecting codes continues to expand. Since the 1990s, an approach known as *low-density parity-check codes* has received considerable attention. These codes are now used in applications ranging from satellite TV to communication with deep space probes. So the next time you enjoy some high-definition satellite TV on the weekend, spare a thought for this delicious irony: it was the frustration of Richard Hamming's weekend battle with an early computer that led to our own weekend entertainment today.