Error-Correcting and Compression – Part 1

"How come a scratched CD can still play flawlessly?"

Information in a computer memory, or on a CD is stored in zeros and ones (see Figure 1). If the CD gets damaged, or a computer memory gets hit by cosmic radiation, or if there is a malfunction glitch, then some zeros may be misread for ones, or vice versa. How can we protect against this? We are interested in

- detecting errors
- correcting errors.

Let's start simply. Imagine the signal is:

$$1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots ;$$

after some data corrupting event, it becomes:

$$1 \quad 0 \quad 1 \quad \underline{0} \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots$$

This is impossible to detect here.

Instead of storing the original sequence, let us double every bit; we store now:

$$11 \quad 00 \quad 11 \quad 11 \quad 00 \quad 11 \quad 00 \quad 00 \quad 11 \quad 11 \quad 00 \quad \ldots$$

After corruption, some bits get changed:

$$11 \quad 0\underline{1} \quad 11 \quad 11 \quad 00 \quad 11 \quad 00 \quad 00 \quad 1\underline{0} \quad 11 \quad 00 \quad \ldots$$

Now we can see that errors were made: the string should consist only of pairs 00 or 11 (the two allowed *codewords*), but others have crept in (the pairs 01,10), indicating that a mistake must have been made. But we have no way to correct: a 01 could equally well be a 00 in which the second 0 was flipped as a 11 where the first 1 was flipped.

If we repeat each digit not just once, but twice, then we get

$$111 \quad 000 \quad 111 \quad 111 \quad 000 \quad 111 \quad 000 \quad 000 \quad 111 \quad 111 \quad 000 \quad \ldots$$

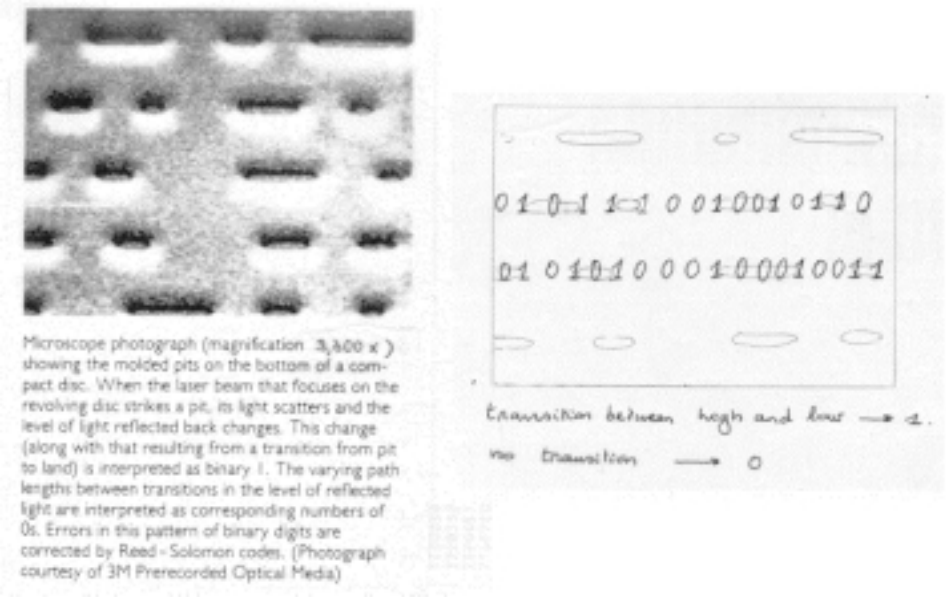


Figure 1: How signals are recorded (in binary form) on CDs.

After corruption, this may look like:

$$111 \quad \underline{\underline{1}}00 \quad 111 \quad 111 \quad 000 \quad \underline{\underline{1}}01 \quad 000 \quad 000 \quad 111 \quad \underline{\underline{0}}11 \quad 000 \quad \ldots$$

Now we can see not only **where** the errors happened, but we can also **correct** them!

- 100 is closer to 000 (1 flip) than to 111 (2 flips); this means that unless we are extremely unlucky, 100 must have come from 000
- 101 must have come from 111
- 011 must have come from 111

So by using three times as much memory, we can correct occasional flip-errors. But this is an enormous increase in the amount of memory needed; maybe we can try to achieve error correction with a different scheme that consumes less memory?

For example, let's do the following: we replace every two bits by a string or codeword of five bits, according to the following rule:

$$\begin{aligned} 00 &\to 00001 \\ 01 &\to 01010 \\ 10 &\to 10100 \\ 11 &\to 11111 \end{aligned}$$

For our original message

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

this gives

10100 11111 01010 00001 11111 ...

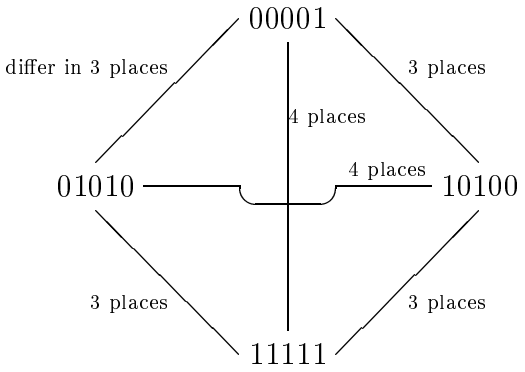
Can we correct here?

To make sure that we can always correct a single error in a codeword, we can do the following: for each codeword, make the list of the uncorrupted codeword and the corruptions that can be obtained if one bit is flipped. For our example we obtain:

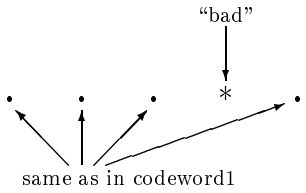
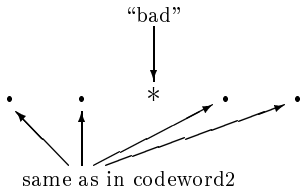
uncrypted codeword	00001	01010	10100	11111
possible 1-bit corruptions	10001	11010	00100	01111
	01001	00010	11100	10111
	00101	01110	10000	11011
	00011	01000	10110	11101
	00000	01011	10101	11110

No single five-bit string occurs twice here. That means that we can always trace one-flip errors back to the original codeword. This scheme is therefore an error correcting code that increases memory only by a factor 2.5, instead of the factor three we had before, and that still can correct single-flip errors. (Note that we have to assume that errors do not occur too closely together. In our first scheme, we can correct if there is at most 1 error in every codeword of 3 consecutive bits; in the scheme we are disscussing now, we can correct if there is at most 1 error in every codeword of 5 consecutive bits. Schemes like this would be used in practice when errors occur at much lower rates than this, such as in cosmic radiation disturbance for PC memories, so this is not so important.)

Another way to see that the code we have just given can correct single-flip errors is to see that every two codewords differ in at least three places:



Since the possible corruptions of the codewords can only differ from the codewords in one place, that means that corruptions of two different codewords must be different, as shown in more detail by the following argument:

> **codeword1 + corruption1** has at least four bits that are the same as in codeword1. For instance:
>
> 
>
>
> **codeword2 + corruption2** has again at least four bits that coincide with those in codeword 2. For instance:
>
> 
>
>
> If the two corruptions are identical, then all their bits are identical. Since the two corrupted codewords are identical to their original codewords in three shared places (in this case, in first, second, and fifth place), it follows that codeword1 and codeword2 must have three common bits. This is not possible if codeword1 differs from codeword2, since codewords always differ in 3 or more places.

However, a code like the one we just described would not be used in practice, because it is wasteful. What do we mean by this? Consider the total list of codewords and their possible one-flip errors. In our case, that total list has 24 entries: there are 4 codewords, which each consist of 5 bits, so that each codeword can lead to 5 different 1-bit corrupted codewords, leading to a total of $4\times(1+5) = 24$. On the other hand, there are $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$ possible five-bit strings. Our list of 24 codewords and their 1-bit errors therefore does not exhast all possible 5-bit strings. The eight missing strings could be reached by two-flip errors from some codewords (for example 00110 can be reached by two flips from 01010), but that doesn't help us, because the code would not be able to correct all two-flip errors (for instance, 11110 could be viewed as a two-flip error from 01010, or as a one-flip error from 11111).

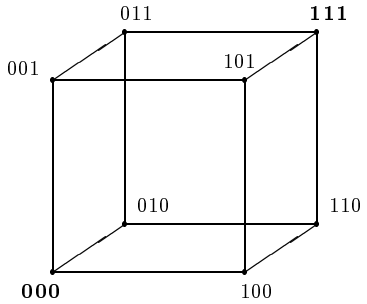
Let's check how "wasteful" the tripling code is. The encoding consists there of the correspondence

$$\begin{aligned} 0 &\rightarrow 000 \\ 1 &\rightarrow 111\ , \end{aligned}$$

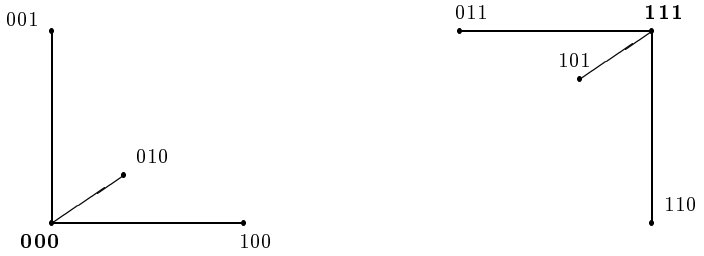
in this case we have the following list of codewords and one-flip corruptions of codewords:

000	111
001	110
010	101
100	011

This list of 8 entries **does** exhaust all possible strings of three bits (the total number of such possibilities $= 2^3 = 8 \rightarrow$ OK!). A code where the list of codewords and all possible corruptions fills the full available space is called a "perfect" code. The example with 5-bit codewords that we discussed earlier was not a perfect code. Perfect codes have a nice geometric interpretation. Let us look at our "tripling" codewords and their corruptions, and identify each with a corner on a cube in three dimensions:



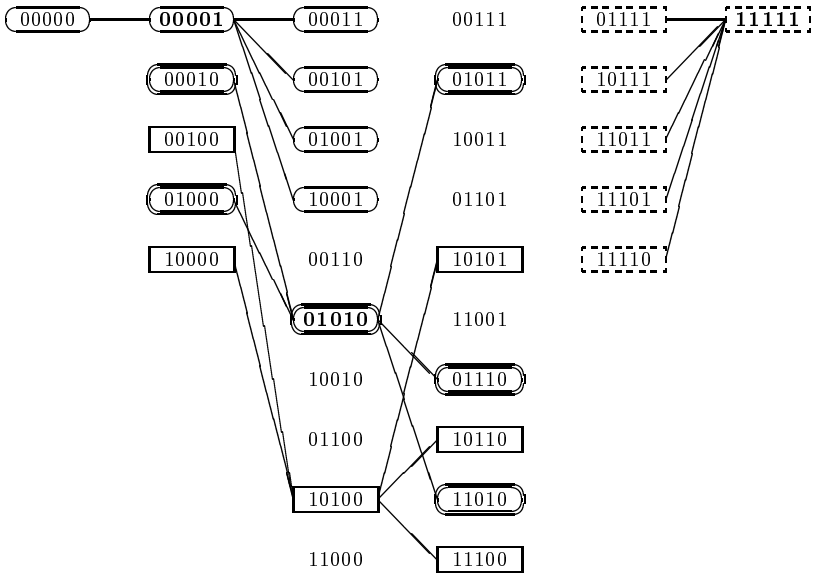
The two codewords are the vertices 000 and 111, and their corruptions together with the codeword form a kind of tripod:



These two tripods fit together and cover all the vertices of the cube.

For the code on page 2, the equivalent would be a cube in five dimensions (which we cannot draw); each codeword is at the center of a star with five branches, but the four different stars do not cover all the corners of the five-dimensional cube. We cannot draw this, but we can still list all the vertices, and identify the five-armed "stars," as in the table below.

We write all the possible 5-bit strings in 6 columns, depending on how many 1s they have (first column: no 1; second column: just one 1; third column: two 1s, etc.). We mark each codeword with a different type box (oval, double oval, rectangular solid, rectangular dashed), and then link to it all the corresponding 1-bit corrections, marked with a similar box. Clearly the 5 "stars" don't overlap, but they do not cover the whole table.



There are eight groups of five digits, that is eight corners on the five-dimensional cube, that are not covered!

The Hamming code is a code that improves on this situation. Here the correspondence replaces four-bit strings by seven-bit codewords:

Message		Code Word
0000	$\rightarrow$	0000000
0001	$\rightarrow$	0001011
0010	$\rightarrow$	0010111
0100	$\rightarrow$	0100101
1000	$\rightarrow$	1000110
1100	$\rightarrow$	1100011
1010	$\rightarrow$	1010001
1001	$\rightarrow$	1001101
0110	$\rightarrow$	0110010
0101	$\rightarrow$	0101110
0011	$\rightarrow$	0011100
1110	$\rightarrow$	1110100
1101	$\rightarrow$	1101000
1011	$\rightarrow$	1011010
0111	$\rightarrow$	0111001
1111	$\rightarrow$	1111111

(We follow the same convention as in "For All Practical Purposes" and in the video shown in class, where the rule for making codewords is explained with Venn diagrams.)

Again, all the codewords differ in at least three places, so we have a code that can correct one-bit-flip errors. This code now "lives" in seven dimensions, so that every "star" listing a codeword and its possible one-bit corruptions has eight elements (the codeword at its "center" and seven "branches" to the one-bit corruptions). We have 16 codewords in all, so the stars cover $16 \times 8 = 128$ different corners of the seven-dimensional cube. But this cube has exactly $2^7 = 128$ corners, so we have them all! The Hamming code is a **perfect** code.

It has other beautiful properties.

In the Hamming code, there is a simple way to compute the correspondence from four-bit string to seven-bit codeword. (We could, of course, always look it up in a table, but it is much easier if we don't have to do this. For 16 codewords this is not so bad, but larger codes, like the Reed-Solomon codes, may have 256 or even larger numbers of codewords ...).

This is how one computes the 7-bit codewords from the 4-bit strings $b_1 b_2 b_3 b_4$:

$b_1\ b_2\ b_3\ b_4$
$\rightarrow\ b_1\ b_2\ b_3\ b_4\ (b_1 \oplus b_2 \oplus b_3)\ (b_1 \oplus b_3 \oplus b_4)\ (b_2 \oplus b_3 \oplus b_4)$
(where $\oplus$ stands for parity-addition)

This makes the Hamming code a **linear** code: codewords are obtained from the uncoded data by a simple linear computation. It also means that for every codeword

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7$$

the following three constraints hold:

$$b_1 \oplus b_2 \oplus b_3 \oplus b_5 = 0$$
$$b_1 \oplus b_3 \oplus b_4 \oplus b_6 = 0$$
$$b_2 \oplus b_3 \oplus b_4 \oplus b_7 = 0$$

And this leads to a very simple algorithm for decoding. Imagine that

$$\widetilde{b}_1 \quad \widetilde{b}_2 \quad \widetilde{b}_3 \quad \widetilde{b}_4 \quad \widetilde{b}_5 \quad \widetilde{b}_6 \quad \widetilde{b}_7$$

is a possibly corrupted codeword. We then compute the three "parity-checks"

$$p_1 = \widetilde{b}_1 \oplus \widetilde{b}_2 \oplus \widetilde{b}_3 \oplus \widetilde{b}_5$$
$$p_2 = \widetilde{b}_1 \oplus \widetilde{b}_3 \oplus \widetilde{b}_4 \oplus \widetilde{b}_6$$
$$p_3 = \widetilde{b}_2 \oplus \widetilde{b}_3 \oplus \widetilde{b}_4 \oplus \widetilde{b}_7$$

Depending on the outcomes p_1, p_2, p_3, we have to do different things.

1. If $p_1 = p_2 = p_3 = 0$, then $\widetilde{b}_1\ \widetilde{b}_2\ \widetilde{b}_3\ \widetilde{b}_4\ \widetilde{b}_5\ \widetilde{b}_6\ \widetilde{b}_7$ was a correct codeword, and we can go back to the four-bit string by just dropping $\widetilde{b}_5, \widetilde{b}_6, \widetilde{b}_7$.

2. If one of the p_j equals 1, but the other two are zero: look for the $\widetilde{b}_\ell$ which comes up in p_j but not in the other and flip it in order to obtain a correct codeword.

   Example: $p_2 = 1 \quad p_1 = p_3 = 0$. The only $\widetilde{b}$ which occurs in p_2 and not in p_1 or p_3 is $\widetilde{b}_6 \Rightarrow$ flip $\widetilde{b}_6$ to get the correct codeword. Then drop $\widetilde{b}_5, \widetilde{b}_6, \widetilde{b}_7$ to go back to the four-bit string. (In fact, you can go back straight away in this case, because only $\widetilde{b}_5$, $\widetilde{b}_6$, or $\widetilde{b}_7$ will need to be flipped, and they get dropped anyway.)

3. If only one of the p_j equals 0, and the other two are 1: look for the $\widetilde{b}$-bit that occurs in the two "wrong" parity-bits, but not in the other, and flip it.

   Example: $p_1 = p_2 = 1 \quad p_3 = 0$. The only $\widetilde{b}$ which p_1, p_2 have in common but that does not occur in p_3 is $\widetilde{b}_1 \Rightarrow$ flip $\widetilde{b}_1$ to get the correct codeword.

4. If $p_1 = p_2 = p_3 = 1$, then flip $\widetilde{b}_3$.

So correcting the errors is very simple.

Note that in this Hamming code a 4-bit string is encoded by a 7-bit codeword: this code multiplies the amount of memory needed by $\frac{7}{4} = 1.75$. In the earlier more "wasteful" example, which also appended three bits to every string that was encoded, and that turned 2-bit strings into 5-bit codewords, the memory-expansion factor is $\frac{5}{2} = 2.5$: it is indeed more wasteful!

What would happen if we add four bits? Can we make a perfect code? How large would it be?

There are indeed "larger" Hamming codes, which are still perfect, corresponding with appending 4 bits to make codewords, or 5 bits (for a yet larger code), etc. We shall not discuss them in detail here. But we can easily compute how many bits there will be in the codewords and in the un-encoded strings, for each case. Let's look at the case where we add 4 bits to an un-encoded string so as to turn it into a codeword. Imagine starting from strings that are n bits long. There are 2^n such strings, so we will need 2^n codewords. The codewords are $n+4$ bits long (since we add four bits), so they "live" in a world where 2^{n+4} strings are possible. Around each codeword, we will have a "star" (codeword and one-bit-flips) of $1+(n+4) = n+5$ elements. In order for the code to be perfect, we need

$$\begin{array}{ccccc} (n+5) & \times & 2^n & = & 2^{n+4} \\ \uparrow & & \uparrow & & \uparrow \\ \text{size of} & & \text{number of} & & \text{total number of} \\ \text{every star} & & \text{codewords} & & \text{possible } (n+4)\text{-strings.} \end{array}$$

or $(n+5) = 2^4$. This is fulfilled for $n = 2^4 - 5 = 16 - 5 = 11$. So this would lead to a perfect code that replaces 11-bit strings by 15-bit codewords, so that the amount of memory needed is multiplied by $\frac{15}{11} \simeq 1.4$—better again! Of course, we can now afford only one error every 15 bits after encoding, instead of one every seven bits before.

There exists again a corresponding nice linear code that can be easily decoded; this is the 11-15 Hamming code. There is a whole hierarchy of Hamming codes that can be constructed in this way; they are widely used to protect data in computer systems.

One can, of course, also construct codes that can correct **two** errors in one codeword. In that case, every two codewords should differ in at least **five** places.

The codes used on CDs

Many other error correcting codes were developed after Hamming's work. About ten years later, Irving Reed and Gus Solomon developed what are now known as the Reed-Solomon codes, used in the coding for CDs and in the transmission of information from outer space (such as the Voyager images, for instance).

The key to the Reed-Solomon codes is to use a different **alphabet**.

In the Hamming codes, we were working with only two symbols, 0 and 1. We could also have written them as a and b, so that a string to be encoded could be, e.g.

$$abbababaabaabbbaabaababbbbbab\ldots.$$

Let us now change our point of view, and take not the individual a and b as building blocks,

but instead the four possible pairs aa, ab, ba, and bb. The string then becomes

$$ab\ ba\ ba\ ba\ ab\ aa\ bb\ ba\ ab\ aa\ ba\ bb\ bb\ ab\ \ldots.$$

We can now choose to use an enlarged alphabet, writing A for every pair aa, B for ab, C for ba and D for bb; now this string becomes

$$B\ C\ C\ C\ B\ A\ D\ D\ C\ B\ A\ C\ D\ D\ B\ \ldots.$$

By taking larger chunks as building blocks, we would enlarge the alphabet even more.

What makes the Hamming code especially nice is that the codewords could be easily computed from the uncoded string by simple sums, using the "parity add" (more commonly called "exclusive or" or "xor") operation. In order to make these larger alphabet codes practical to work with, an analog of the xor operation is used, again to **compute** codewords, and to compute the error correction (instead of having to look it up in a gigantic table). We thus have to work in a finite collection of symbols, in which special operations are defined; mathematicians call this a **finite field**. (We have already seen examples of such finite fields, when we looked at modular arithmetic—when you work with numbers "modulo 7," you have an "alphabet" of seven "letters," namely the numbers 0 though 6, and we saw how to make sense of the operations "addition" and "multiplication" on this alphabet.)

In practice, the Reed-Solomon codes use finite fields with 256 elements, that is they have an "alphabet" with 256 "letters."

If you have a RS code that corrects for **ONE** wrong letter, then that is really the same as a binary code that corrects for **EIGHT** bit-errors, because you could imagine writing every one of the 256 letters as an 8-bit string. (Strictly speaking, you have to be more careful here: in the binary representation of RS words, eight consecutive errors need not fall exactly on eight bits corresponding to *one* RS letter: they could correspond to two adjacent RS letters, straddling their two 8-bit representations. More precisely: $8N$ consecutive binary errors can correspond to $N + 1$ consecutive wrong RS letters.)

To encode sound on an audio CD several different strategies are used.

1. The RS code used is one that that can correct for several consecutive wrong letters. Say that we can correct 40 consecutive wrong letters from the RS alphabet.

2. Moreover, information is spread out by interleaving. Suppose the string of codewords is

   $$\ldots\ c1\ c2\ c3\ c4\ c5\ c6\ c7\ c8\ c9\ c10\ c11\ c12\ c13\ c14\ c15\ c16\ c17\ c18\ c19\ \ldots$$

   Instead of writing them down in this original order, let's interleave them as follows:

   $$\ldots\ c1\ c11\ c2\ c12\ c3\ c13\ c4\ c14\ c5\ c15\ c6\ c16\ c7\ c17\ c8\ c18\ c9\ c19\ c10\ \ldots$$

If, in this new order, two consecutive letters are misread, that really corresponds to two errors that are 10 symbols apart in the original sequence! Similarly, four consecutive errors in the interleaved sequence correspond to two smaller bursts of two consecutive errors each, but spaced 10 symbols apart in the original sequence. If you interleave more, then you can reduce the effective "burst-length" even more.

In the CD disk application, the interleaving is with a factor 5, and it is very widely spread out. The result is that if you make N consecutive errors in this interleaved representation, then you have really made only 5 distant groups of errors of length $N/5$ in the original representation.

3. Finally, every letter in the RS alphabet is encoded not just with 8 bits (corresponding to the 256 possibilities), but with 10 bits, with a little error correcting code thrown in at this individual level as well (but it is a different scheme).

The big scratch we made on the CD in class was about 0.25 mm wide. Since 0.01 mm correspond to 8 bits, we destroyed about 200 consecutive bits. By unwrapping the 8-10 "outer" code, this means that we destroyed about $160 = 8/10 \times (200)$ worth of bits in the RS language, or (since every RS word takes 8 bits) about 20 RS words. Because of the interleaving, this corresponds to 5 distant groups of errors, each of about 4 RS letters. This is a breeze for the RS correcting code to handle—it was designed to take care of much longer errors than this!

Detecting errors in barcodes, airline ticket numbers and elsewhere

In the applications we discussed before (computer memories, CDs, transmission of images from outer space), we were dealing with extremely long data strings, and we wanted to be able to not only **detect** but also to **correct** errors.

In a completely different set of applications, you want to read or register a fairly short number (say ten digits) and detect whether it is correct or not—usually you don't care to correct it automatically; if you detect an error, you just ask for confirmation (credit card number for phone ordering) or you have to resort to manual handling based on other information (typing in the UPC code at a cash register in the supermarket, correcting the ZIP code based on the address).

For many of these applications, some error detection capacity is built into the numbers. Let's look at a few schemes, and their respective merits.

American Express Traveler's Checks

Check number 448721117 has at its bottom the ID number 4487211171:

$$\begin{array}{ll} 448721117 & 1 \\ & \uparrow \\ & \text{this extra digit is a check digit.} \end{array}$$

What is the relation of the check digit to the others? Add all other digits, and take the remainder of this sum after dividing by 9.

$$4+4+8+7+2+1+1+1+7 = 35 \equiv 8 \pmod 9$$

Then compute $9-8=1$ (1 is the check digit).

Or, in other words

$$4+4+8+7+2+1+1+1+7+1 \equiv 0 \pmod 9$$

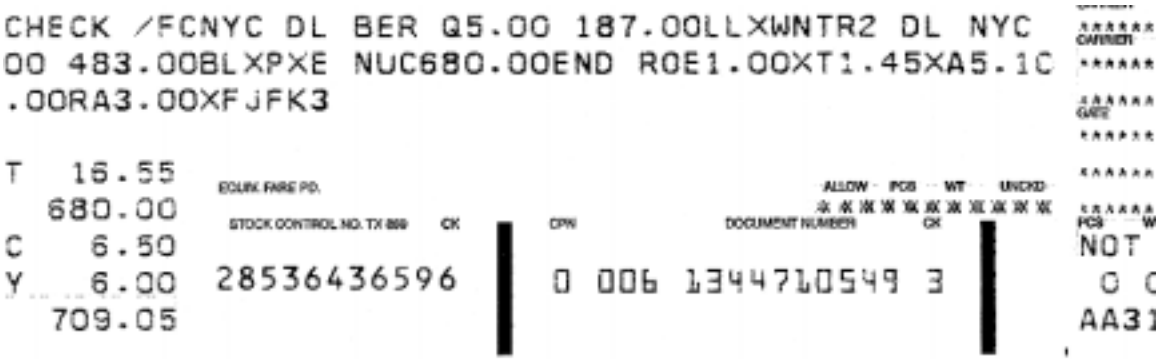
Sum of all digits + check digit = multiple of 9

Check digit can take values $0, 1, 2, \ldots, 8$. What kind of errors can we detect?

- Substitution of 0 by 9 or vice versa in other digits than the check digit: **NO**
- Other substitutions of a single digit by another: **YES**
- Inversions (that is, $\ldots 53 \ldots$ instead of $\ldots 35 \ldots$): **NO**

The same method is used on US Postal Service money orders.

Airline tickets



Stock control number Tx 889	CK
2853643659	6
	↑
	Check digit is remainder after dividing the long 10 digit number by 7.

Check digit can take values $0, 1, \ldots, 6$. Detection of wrong digit?

- Substitution of 0 for 7 (or vice versa), 1 for 8 (or vice versa), 2 for 9 (or vice versa), are undetectable, except in check digit.
- Other substitutions: detectable.
- Inversions?

correct:	$\ldots 34 \ldots C$
incorrect:	$\ldots 43 \ldots C$

  Can this be detected? In other words, would the check digit for the second number really be C?

  The first number (without its check digit) = (multiple of 7) + C.

  The second number (also dropping the last digit or the would-be check digit) = first number +90000 = multiple of $7 + C + 1$ (because the remainder of 90000 after you divide by 7 is 1); this means that the check digit of second number should be different from C $\rightarrow$ error detected.

  Not all inversions can be detected:

correct:	$\ldots 18 \ldots C$
incorrect:	$\ldots 81 \ldots C$

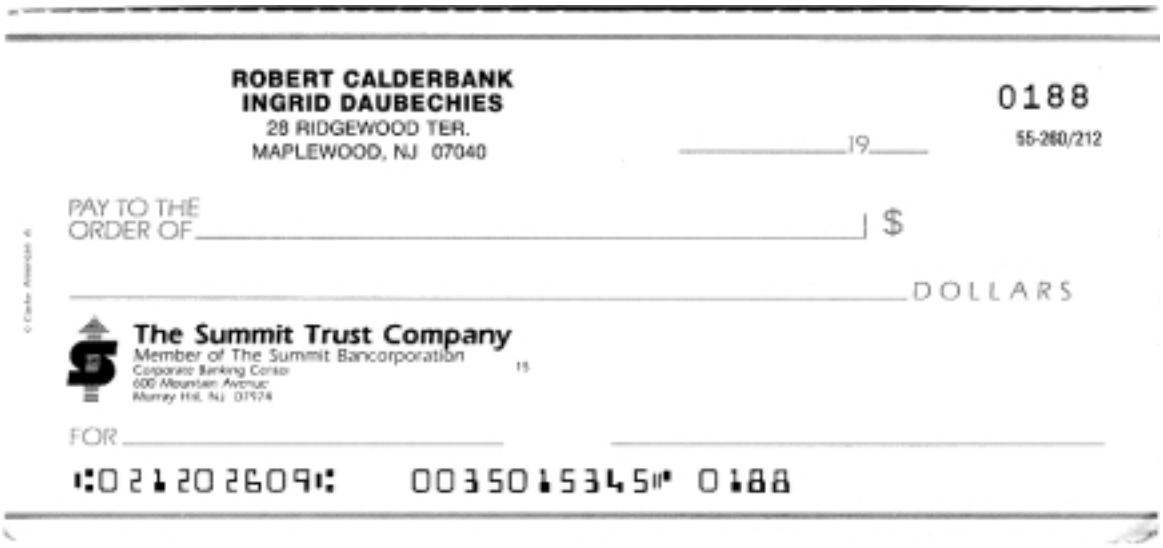
First number without check digit = multiple of $7 + C$

Second number without check digit = first number $+63000000$ = multiple of $7 + C$.

This inversion is not detected, because it does not affect the remainder after dividing by 7, so the check digits are the same.

This scheme is also used by FedEx, UPS, AVIS, and National car rental agencies.

American banking system



0	2	1	2	0	2	6	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	check digit

Compute

$7a_1$	+	$3a_2$	+	$9a_3$	+	$7a_4$	+	$3a_5$	+	$9a_6$	+	$7a_7$	+	$3a_8$
0	+	6	+	9	+	14	+	0	+	18	+	42	+	0

$= 89 \equiv 9 \pmod{10}$

$\Rightarrow$ check digit is 9 .

Detectability of errors?

- Single digit mistake in check digit: always detected
- Single digit elsewhere?

in first position:

$$\begin{array}{lcccccccc} \text{correct:} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{mistake:} & a_1' & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

Then $7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C$.

$7a_1' + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C + 7(a_1' - a_1)$.

The error will be undetected only if $7(a_1' - a_1) =$ a multiple of 10. But $a_1' - a_1$ ranges from -9 to 9 (with the exception of 0, since $a_1' \neq a_1$: a mistake was made!), and multiplying this by 7 gives the numbers

$$-63, -56, -49, -42, -35, -28, -21, -14, -7, 7, 14, 21, 28, 35, 42, 49, 56, 63$$

Not a single multiple of 10! All errors in first place will be detected. Same argument applies for fourth or seventh place.

What for an error in second place?

$$\begin{array}{lcccccccc} \text{correct:} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{mistake:} & a_1 & a_2' & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

$7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C$.

Then $7a_1 + 3a_2' + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C + 3(a_2' - a_2)$.

Here $a_2' - a_2$ ranges again from -9 to 9, with the exception of 0, and multiplying by 3 gives a collection of possible values for $3(a_2' - a_2)$, none of which is a multiple of 10. All errors in second (and fifth and eighth) position are detected.

The same argument, but now with $9(a_3' - a_3)$ (multiplying $a_3' - a_3$ by 9, where $a_3' - a_3 \neq 0$ and between -9 and 9, never gives a multiple of 10) shows that all errors in third or sixth place will be detected.

Inversion errors? Example:

$$\begin{array}{lcccccccc} \text{correct:} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{mistake:} & a_2 & a_1 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

$7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C$

$7a_2 + 3a_1 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 =$ multiple of $10 + C + 4a_2 - 4a_1$.

This will go **undetected** if $4(a_2 - a_1)$ is a multiple of 10, for instance if $a_2 - a_1 = -5$ or 5. (These are the only two cases.)

Examples:

$$a_1 = 2 \quad a_2 = 7$$

or

$$a_1 = 9 \quad a_2 = 4\,.$$

Inversions of other consecutive digits?

$$\begin{array}{l} \cdots a_2 a_3 \cdots\cdots \\ \cdots a_3 a_2 \cdots\cdots \end{array}$$

is undetectable if

$$3a_2 + 9a_3 - 3a_3 - 9a_2 = 6(a_3 - a_2)$$

is a multiple of 10. Again, if $a_3 - a_2 = 5$ or -5.

Inversion of third and fourth digit:

$$\begin{array}{l} \cdots a_3 a_4 \cdots\cdots \\ \cdots a_4 a_3 \cdots\cdots \end{array}$$

This will be undetectable if

$$9a_3 + 7a_4 - 9a_4 - 7a_3 = 2(a_3 - a_4)$$

is a multiple of 10. Again, if $a_3 - a_4 = 5$ or -5.

Most inversions are detected in this scheme, but not all. Reason:

$$\left.\begin{array}{l} 7-3=4 \\ 3-9=-6 \\ 9-7=2 \end{array}\right\}$$ have a common factor 2 with 10, the basis of the modular arithmetic

differences of consecutive weights

Remark: We can analyze the earlier airline ticket scheme in the same way. After all, dividing 6483839328 by 7 is the same as dividing

$$\begin{array}{l} 6 \times 1000000000 + 4 \times 100000000 + 8 \times 10000000 + 3 \times 1000000 \\ +8 \times 100000 + 3 \times 10000 + 9 \times 1000 + 3 \times 100 + 2 \times 10 + 8 \end{array}$$

by 7. Since

$$\begin{array}{rcl} 10 & = & \text{multiple of } 7 + \underline{3} \\ 100 & = & \text{multiple of } 7 + \underline{2} \\ 1000 & = & \text{multiple of } 7 + \underline{6} \\ 10000 & = & \text{multiple of } 7 + \underline{4} \\ 100000 & = & \text{multiple of } 7 + \underline{5} \\ 1000000 & = & \text{multiple of } 7 + \underline{1} \\ 10000000 & = & \text{multiple of } 7 + \underline{3} \\ 100000000 & = & \text{multiple of } 7 + \underline{2} \\ 1000000000 & = & \text{multiple of } 7 + \underline{6}\,, \end{array}$$

this means that we could also have looked at the remainder, mod 7, of the combination

$$6 \times \underline{6} + 4 \times \underline{2} + 8 \times \underline{3} + 3 \times \underline{1} + 8 \times \underline{5} + 3 \times \underline{4} + 9 \times \underline{6} + 3 \times \underline{2} + 2 \times \underline{3} + 8 \times \underline{1} .$$

More generally, if the number is

$$a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ a_9 \ a_{10} ,$$

then the check digit is equal, mod 7, to the combination

$$6a_1 + 2a_2 + 3a_3 + a_4 + 5a_5 + 4a_6 + 6a_7 + 2a_8 + 3a_9 + a_{10} .$$

Let us now do the same reasoning to see whether a wrong digit in, say, sixth place will be detectable. Changing

$$\cdots\cdots a_6 \cdots$$
$$\cdots\cdots a_6' \cdots$$

leads to the same check digit if $4(a_6' - a_6)$ is a multiple of 7. Here 4 has no common factor with 7, so we are okay there, but $a_6' - a_6$ can range from -9 to 9 (excepting 0), so that $a_6' - a_6 = 7$ or -7 is possible .... Undetectable error if we substitute 0 for 7 (or vice versa), 1 for 8 (or vice versa), 2 for 9 (or vice versa).

The UPC System

(bar codes on grocery products)

Two different parts to UPC-bar codes:

- reading the bar code itself
- check digit

Bar Code: The bar code is a way of printing, via thick and thin lines and narrow and wide spaces, binary representations for numbers.

In this representation, every decimal digit is encoded by 7 binary digits.

How should we read the barcode:

- thin black line = 1
- slightly thicker black line = 11
- thick black line = 111

- very thick black line = 1111

- thin blank space = 0
- slightly wider blank space = 00
- wide blank space = 000
- very wide blank space = 0000

For instance, in the example above, we have

left side
|| 0001101 0100011 0110111 0110001 0001101 0001101 || 1110010 1110010 1100110 1000010 1110100 1011100 ||
right side
separator separator separator

The code for reading this as standard digits is the following:

	left	right
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100.

(This is not binary coding of the digits! The groups of zeros and ones are chosen so that very special properties hold; see below.)

For the example above, we get 048500 00139 4. The number encoded in the UPC bar code is a $6+6$ digit number, or rather

1	+	5	+	5	+	1		digits
↑		↑		↑				
identifies kind of product		code for manufacturer		code for product		check digit (see below) not always printed in decimal numbers, but always present in bar codes		

Note:

- on left, every group of 7 zeros and ones has an **odd** number of ones; on right, every group of 7 has an **even** number of ones. Automatic reading system can decide when it reads things backwards!
- on left: start = 0, end = 1 for all groups. On right: start = 1, end = 0 for all groups (in order to separate things nicely).

Check digit

Take the first 11 digits: $a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}$, and compute

$$3a_1 + a_2 + 3a_3 + a_4 + 3a_5 + a_6 + 3a_7 + a_8 + 3a_9 + a_{10} + 3a_{11} \equiv A \pmod{10}\,.$$

then the check digit C, or a_{12} is $a_{12} = C = 10 - A \pmod{10}$.

In our example:

$$\begin{aligned} &3a_1 + a_2 + 3a_3 + a_4 + 3a_5 + a_6 + 3a_7 + a_8 + 3a_9 + a_{10} + 3a_{11} \\ &= 4 + 24 + 5 + 3 + 3 + 27 = 66 \Rightarrow A = 6 \Rightarrow a_{12} = C = 4\,. \end{aligned}$$

ISBN

(International Standard Book Number)

ISBN is the smartest of these error detection schemes: it detects **all** single errors and **all** inversions.

ISBN number:

$$\begin{array}{cc} a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9 & a_{10} \\ & \uparrow \\ & = C \end{array}$$

How find C from $a_1 - a_9$? Compute

$$10a_1 + 9a_2 + 8a_3 + 7a_4 + 6a_5 + 5a_6 + 4a_7 + 3a_8 + 2a_9 \equiv A \pmod{11}$$

then $a_{10} = C = 11 - A \pmod{11}$.

Example: 0-7167-2378-6

$$10 \times 0 + 9 \times 7 + 8 \times 1 + 7 \times 6 + 6 \times 7 + 5 \times 2 + 4 \times 3 + 3 \times 7 + 2 \times 8$$
$$= 214 \Rightarrow A = 5 \Rightarrow a_{10} = 6 \text{ OK!}$$

Because 11 is prime, and all the weights $(10, 9, 8, \ldots, 2)$ are therefore relatively prime with respect to 11, this detects all single errors.

Detect transpositions? The difference between two consecutive weights is always 1.

Fail to detect transposition between nth place and $(n+1)$th places if $(a_n - a_{n+1})$ is multiple of $11 \Rightarrow$ can't be $\Rightarrow$ all transpositions are detected.

Only inconvenience: sometimes $a_{10} = 10 \ldots \Rightarrow$ replace by X.

Example: 0-273-00218-X.

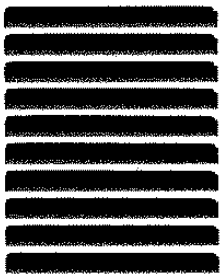
Another bar code: the ZIP bar code

This is the code found on many bulk-mail envelopes (business reply cards, for instance). Every ZIP+4 number is represented by

$$1 + 10 \times 5 + 1 = 52$$

long and short bars at the bottom of the letter. For instance:

No Postage
Necessary
if Mailed
in the
United States



BUSINESS REPLY MAIL

First Class Permit No. 10207 Washington, D.C.

Postage will be paid by addressee

ISSUES IN
SCIENCE AND TECHNOLOGY
National Academy of Sciences
2101 Constitution Avenue, N.W.
Washington, D.C. 20077-5576

The dictionary to read these is the following:

- initial and final long bars are just guard lines
- every group of five lines encodes one digit, according to the following correspondence:

Decimal Digit	Bar Code	Binary Code
1		00011
2		00101
3		00110
4		01001
5		01010
6		01100
7		10001
8		10010
9		10100
0		11000

The Postnet bar code

The number above is therefore 2007755761. This correspondes to the ZIP+4 code 20077-5576; 1 is a check digit. How does one construct the check digit?

$$a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10}$$
$$\uparrow$$
check digit

should be an exact multiple of 10.

In example:

$$\underbrace{2+7+7+5+5+7+6}_{\text{total is } 39}+1 \quad \rightarrow 40\text{: OK}$$

But now we have some error correction possibility too! Every group of five bars representing a digit has **exactly** two long and three short bars. Misreading happens when a short bar is mistaken for a long one or vice versa. We then not only know that we are making an error, but we know **where**! (We have a parity check within every group of five representing a digit—remember the parity checks within codewords earlier!) We can then use the check digit to make everything right again.

Example:

Here one group of five has four short bars and only one long bar. We therefore can only read the other digits, resulting in

$$2007?\ 55761\ .$$

But ? is easily found:

$$\underbrace{2+0+0+7+5+5+7+6+1}_{\text{total is } 33}+?$$

should be evenly divisible by 10 $\rightarrow$?=7 here.

More recent bar code for ZIP codes uses 12 digits:

$$\underbrace{\text{ZIP}+4}+\text{last 2 digits of address} + \text{check digit.}\ \ \text{total of 9}$$

again: sum of all digits should be divisible by 10.

Example: