Error-Correcting Codes

Prepared by Mark on May 9, 2025

Part 1: Error Detection

An ISBN[1] is a unique identifier publishers assign to their books.
It comes in two forms: ISBN-10 and ISBN-13. Naturally, ISBN-10s have ten digits, and ISBN-13s have thirteen. The final digit in both versions is a *check digit*.

Say we have a sequence of nine digits, forming a partial ISBN-10: $n_1 n_2 ... n_9$.
The final digit, n_{10}, is chosen from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ so that:

$$\sum_{i=1}^{10} (11 - i) n_i \equiv 0 \bmod 11$$

If n_{10} is equal to 10, it is written as `X`.

Problem 1:
Only one of the following ISBNs is valid. Which one is it?
Note: Dashes are meaningless.

- `0-134-54896-2`
- `0-895-77258-2`

[1] International Standard Book Number

Problem 2:
Take a valid ISBN-10 and change one digit.
Is it possible that you get another valid ISBN-10?
Provide an example or a proof.

Problem 3:
Take a valid ISBN-10 and swap two adjacent digits. This is called a *transposition error*.
When will the result be a valid ISBN-10?

Definition 4:
ISBN-13 error checking is slightly different.
Given a partial ISBN-13 with digits $n_1 n_2 n_3 ... n_{12}$, the final digit is given by

$$n_{13} = \left[\sum_{i=1}^{12} n_i \times (2 + (-1)^i) \right] \bmod 10$$

Problem 5:
What is the last digit of the following ISBN-13?
`978-030-7292-06*`

Problem 6:
Take a valid ISBN-13 and change one digit. Is it possible that you get another valid ISBN-13? If you can, provide an example; if you can't, provide a proof.

Problem 7:
Take a valid ISBN-13 and swap two adjacent digits. When will the result be a valid ISBN-13?
Hint: The answer here is more interesting than it was last time.

Problem 8:
`978-008-2066-466` was a valid ISBN until I changed a single digit.
Can you find the digit I changed? Can you recover the original ISBN?

Part 2: Error Correction

As we saw in **??**, the ISBN check-digit scheme does not allow us to correct errors.
QR codes feature a system that does.

Odds are, you've seen a QR code with an image in the center. Such codes aren't "special"—they're simply missing their central pixels. The error-correcting algorithm in the QR specification allows us to read the code despite this damage.

Definition 9: Repeating codes
The simplest possible error-correcting code is a *repeating code*. It works just as you'd expect:
Instead of sending data once, it sends multiple copies of each bit.
If a few bits are damaged, they can be both detected and repaired.
For example, consider the following three-repeat code encoding the binary string 101:

111 000 111

If we flip any one bit, we can easily find and fix the error.

Problem 10:
How many repeated digits do you need to...
- detect a transposition error?
- correct a transposition error?

Definition 11: Code Efficiency
The efficiency of an error-correcting code is calculated as follows:

$$\frac{\text{number of data bits}}{\text{total bits sent}}$$

For example, the efficiency of the three-repeat code above is $\frac{3}{9} = \frac{1}{3} \approx 0.33$

Problem 12:
What is the efficiency of a k-repeat code?

Repeat codes are not efficient. We need to inflate our message by *three times* if we want to correct even a single error. We need a better system.

Part 3: Hamming Codes

Say we have an n-bit message, for example `1011 0101 1101 1001`.
We will number its bits in binary, from left to right:

Bit	1	0	1	1	0	1	0	1
Index	0000	0001	0010	0011	0100	0101	0110	01

and so on...

Problem 13:
If we number the bits of a 16-bit message as described above,
how many message bits have an index with a one as the last digit?
(i.e, an index that looks like `***1`)

Problem 14:
Now consider a 32-bit message.
How many message bits have an index with a one as the n^{th} digit?

Now, let's come up with a way to detect errors in our 16-bit message. To do this, we'll replace a few data bits with parity bits. This will reduce the amount of information we can send, but will allow the receiver to detect errors in the received message.

Let's arrange our message in a grid. We'll make the first bit (currently empty, marked `X`) a parity bit. Its value will depend on the content of the message: if our message has an even number of ones, it will be zero; if our message has an odd number of ones, it will be one.
This first bit ensures that there is an even number of ones in the whole message.

Bit Numbering

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

Sample Message

X	0	1	1
0	1	0	1
1	1	0	1
1	0	0	1

Problem 15:
What is the value of the parity bit in the message above?

Problem 16:
Can this coding scheme detect a transposition error?
Can this coding scheme detect two single-bit errors?
Can this coding scheme correct a single-bit error?

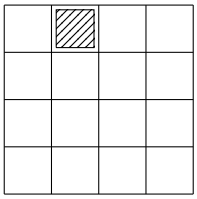
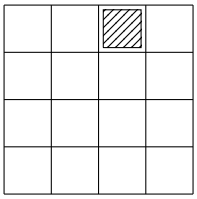
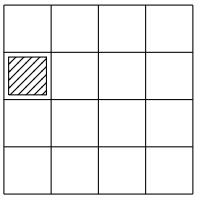
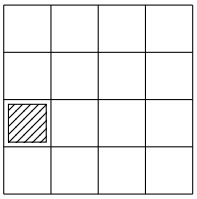
We'll now add four more parity bits, in positions `0001`, `0010`, `0100`, and `1000`:
This error-detection scheme is called the *Hamming code*.

X	X	X	1
X	1	0	1
X	1	0	1
1	0	0	1

Bit `0001` will count the parity of all bits with a one in the first digit of their index.
Bit `0010` will count the parity of all bits with a one in the second digit of their index.
Bits `0100` and `1000` work in the same way.
Hint: In `0001`, `1` is the first digit. In `0010`, `1` is the second digit.
When counting bits in binary numbers, go from right to left.

Problem 17:
Which message bits does each parity bit "cover"?
In other words, which message bits affect the value of each parity bit?

Four diagrams are shown below. In each grid, fill in the bits that affect the shaded parity bit.

Problem 18:
Compute all parity bits in the message above.

Problem 19:
Analyze this coding scheme.

- Can we detect one single-bit errors?
- Can we detect two single-bit errors?
- What errors can we correct?

Problem 20:
Each of the following messages has either one, two, or no errors.
Find the errors and correct them if possible.
Hint: Bit 0000 should tell you how many errors you have.

0	1	1	1
0	1	1	1
0	0	1	1
1	1	1	0

1	1	0	1
1	0	1	0
0	1	1	0
1	1	0	1

0	1	1	1
1	0	1	1
1	0	1	1
1	0	0	0

Problem 21:
How many parity bits does each message bit affect?
Does this correlate with that message bit's index?

Problem 22:
Say we have a message with exactly one single-bit error.
If we know which parity bits are inconsistent, how can we find where the error is?

Problem 23:
Generalize this system for messages of 4, 64, or 256 bits.

- How does the resilience of this scheme change if we use a larger message size?
- How does the efficiency of this scheme change if we send larger messages?

Definition 24:
A *deletion* error occurs when one bit of the message is deleted.
Likewise, an *insertion* error consists of a random inserted bit.

Definition 25:
A *message stream* is an infinite string of binary digits.

Problem 26:
Show that Hamming codes do not reliably detect bit deletions:
Hint: Create a 17-bit message whose first 16 bits are a valid Hamming-coded message, and which is still valid when a bit (chosen by you; not the 17^{th}) is deleted.

Problem 27:
Convince yourself that Hamming codes cannot correct insertions.
Then, create a 16-bit message that:

- is itself a valid Hamming code, and
- incorrectly "corrects" a single bit error when it encounters an insertion error. You may choose where the insertion occurs.

As we have seen, Hamming codes effectively handle substitutions, but cannot reliably detect (or correct) insertions and deletions. Correcting those errors is a bit more difficult: if the number of bits we receive is variable, how can we split a stream into a series of messages?
This is a rhetorical question, which we'll discuss another day.