Today (Friday the 13th!):

Error-detecting and error-correcting codes.

Next week:

Cryptography
From last time to this time...

Compression takes advantage of redundancy in the data to store it in less space and take less time to transmit:

But: transmission and storage media are physical devices and may corrupt the data with errors which show up as “flipped bits” after processing:

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://via.placeholder.com/150" alt="Image" /></td>
<td><img src="https://via.placeholder.com/150" alt="Image" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Communications</th>
<th>Data Storage</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0011</th>
<th>1011</th>
</tr>
</thead>
</table>
Errors can occur separately or together:

**Single-Bit Errors:**

....10101010101011010100101010100101010101001011001101....

**Burst Errors:**

....10101010101011010100101010010101010101001011001101....

**We will assume errors are:**

- Single Bits
- Independent
- Highly unlikely

**Example:** Suppose a single bit has a “one in a million” chance of being flipped:

\[
\text{Prob( 1 error )} = 10^{-6} = 0.000001
\]

\[
\text{Prob( 2 errors )} = 10^{-12} = 0.0000000001
\]

\[
\text{Prob( 3 errors )} = 10^{-18} = 0.0000000000000001
\]
Two encoding techniques are used to improve the accuracy of binary information during communication and storage:

**Error-Detecting Codes**: Transmit the message and report whether an error occurred:

- **Message**: 0011
- **Decoding Algorithm**
- **Communication Channel**
- **Encoding Algorithm**
- **Error/OK!**
- **Message**: 0011
Two encoding techniques are used to improve the accuracy of binary information during communication and storage:

Error-Detecting Codes: Transmit the message and report whether an error occurred:

Message: 0011

Encoding Algorithm

Communication Channel

Decoding Algorithm

1011

Error/OK!
Two encoding techniques are used to improve the accuracy of binary information during communication and storage:

Error-Detecting Codes: Transmit the message and report whether an error occurred:

Error-Correcting Codes: Transmit the message and try to repair any errors:
Error detection codes are widely used on the Internet because you can always ask for another copy of the message:

When reading data from storage devices (DVDs, CDs, Hard Disks, Bar Codes, ISBN numbers) you can try reading again, but maybe the medium is damaged: no way to get another copy of the message.....

Thus, we need more powerful Error Correcting Codes.
Parity methods for error detection are based on counting the number of 1’s:

**Even Parity** = even number of 1s: 0110 0000 1001 1111

**Odd Parity** = odd number of 1s: 0111 0100 1000 1101

Error Detection Codes use a Parity Bit to force a bit sequence to have even parity:

<table>
<thead>
<tr>
<th>Message</th>
<th>Encoded Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 (odd)</td>
<td>00011 (even)</td>
</tr>
<tr>
<td>1001 (even)</td>
<td>10010 (even)</td>
</tr>
<tr>
<td>1111 (even)</td>
<td>11110 (even)</td>
</tr>
<tr>
<td>1011 (odd)</td>
<td>10111 (even)</td>
</tr>
</tbody>
</table>

Note: Message can be any length!
Error Detection

Check for even parity and delete parity bit

Message: 0011
Communication Channel
Calculate parity bit

No error

0011

00110

00110

0011

00110

Calculate parity bit

Message: 0011

Check for even parity and delete parity bit

1011
Error!

10110

10110

00110

Communication Channel

Calculate parity bit
Error Detection

This method can detect any odd number of bit errors:

<table>
<thead>
<tr>
<th>Message</th>
<th>Encoded Message</th>
<th>Corrupted Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 (o)</td>
<td>00011 (e)</td>
<td>10011 (odd – 1 error)</td>
</tr>
<tr>
<td>1001 (e)</td>
<td>10010 (e)</td>
<td>10011 (odd – 1 error)</td>
</tr>
<tr>
<td>1111 (e)</td>
<td>11110 (e)</td>
<td>00100 (odd – 3 errors)</td>
</tr>
<tr>
<td>1011 (o)</td>
<td>10111 (e)</td>
<td>01000 (odd – 5 errors)</td>
</tr>
</tbody>
</table>
Error Detection

BUT it will fail to detect an even number of errors:

<table>
<thead>
<tr>
<th>Message</th>
<th>Encoded Message</th>
<th>Corrupted Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 (o)</td>
<td>00011 (e)</td>
<td>10011 (even – 2 errors)</td>
</tr>
<tr>
<td>1001 (e)</td>
<td>10010 (e)</td>
<td>01111 (even – 4 errors)</td>
</tr>
</tbody>
</table>

When errors are rare and independent, this works well:

**Recall:** Suppose a single bit has a “one in a million” chance of being flipped:

\[
\begin{align*}
\text{Prob( 1 error )} & = 10^{-6} = 0.000001 \\
\text{Prob( 2 errors )} & = 10^{-12} = 0.0000000000001
\end{align*}
\]
Error Correcting Codes add additional bits to locate the errors; if we know the locations of the errors, we can flip them back!

Here’s a simple-minded but expensive way to do it, that was used on early space flights:

Duplicate each message 3 times:

Message:  1011

Duplicated message:  1011011011

Receiver looks at the bits in each duplicate, and lets them “vote” on the correct bit:

1011011011 \rightarrow 11110111011 11110111011 1
11110111011 10......
Error Correcting Codes can do better than duplicating 3x, by using parity bits to check where the error occurred; if we know the location, we can flip it back!

One parity bit tells you there is an error, but nothing about its location:

\[ \text{01011} \xrightarrow{\text{flip}} \text{00011} \]

Two parity bits could tell which half of the message the error occurs in:

\[ \text{010101} \xrightarrow{\text{flip}} \text{000101} \]

With three parity bits we can find the exact location in a 4-bit message and repair it by flipping it back to its original value……
With **three parity bits** we can find the **exact location** in a 4-bit message and repair it by flipping it back to its original value……

Each message bit is checked by a unique set of 2 or 3 parity bits.
To encode a message, we set the parity bits so that each circle has even parity:

0 0 1 1 1 0 0
Note that 3 bits can represent $2^3 = 8$ possibilities, and each of the 7 bits is checked by a unique pattern of 3 parity bits. With “no error” that’s 8 possibilities!
Error Correction

When only one parity bit is wrong, then that bit must have been flipped:

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>Error</th>
<th>0 0 1 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>No Error</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>P_2</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>P_1</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>B_2</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>P_0</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>B_0</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>B_1</td>
<td>0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>B_3</td>
<td>0 0 1 1 0 0 0</td>
</tr>
</tbody>
</table>

Incorrect Transmission

B_0, B_1, B_2, B_3, P_0, P_1, P_2
Message Bits, Parity Bits

P_0, B_0, B_3, P_2
P_1, B_1, B_2

✓ ✓ ✓ ✓
When two parity bits are wrong, then the error was among $B_0$, $B_1$, or $B_2$ and the pattern tells us which one:

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>0 1 1 1 1 0 0</td>
</tr>
</tbody>
</table>

Incorrect Transmission
When three parity bits are wrong, then the error must have been $B_3$:

```
<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>Error</th>
<th>0 0 1 0 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>No Error</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>P_2</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>P_1</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>B_2</td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>P_0</td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>B_0</td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>B_1</td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>B_3</td>
<td></td>
</tr>
</tbody>
</table>
```

Incorrect Transmission: 0 0 1 1 1 0 0
Thus the decoding step fixes any single error produced by faulty transmission or storage:

- **Corrected Message**: 0 0 1 1
- **Corrupted Message**: 0 0 1 0
- **Message**: 0 0 1 1

Parity Check

Correction

Faulty Transmission

Encoding
If we want more message bits, we have to add more parity bits, following the idea that K parity bits can check at most $2^K$ bits:

For 4 parity bits:

$$2^4 = 16 \text{ possibilities} = \text{“no error” + 15 bit locations}$$

$$15 \text{ bits total} = 4 \text{ parity bits} + 11 \text{ message bits}$$

So K parity bits can correct $2^K - 1 - K$ message bits.

<table>
<thead>
<tr>
<th># Par. Bits</th>
<th># Mess. Bits</th>
<th>Total Message Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>57</td>
<td>63</td>
</tr>
</tbody>
</table>

Roughly, to send N bits, we need to add about $\log_2(N)$ parity bits; this is a lot better than using 3N bits to duplicate the message 3 times!
Error Correction

Error Correcting Codes are used just about everywhere!

Applications:
- Internet (at multiple levels)
- Cell-phone networks
- Deep Space probes
- Communication Satellites
- DVDs
- Computer Memory
- ISBN numbers for books
- Bar Codes
- QR-Codes

The final digit is added for error control.

Capacity: 7089 digits
Typical capacity: 20 to 30 digits