From the Homework...

- **Problem 4.** Ben Bitdiddle woke up in the middle of the night with the following great idea: in order to implement double-bit error correction he would use the (8,4,3) code described in lecture – which can correct single-bit errors – to encode a message twice. In other words, after the message was encoded for the first time with the (8,4,3) code, the resulting bit stream would be re-encoded with the same code a second time.

  a. If the original message had 80 bits, how many bits will be in the doubly-encoded message?

  b. Will Ben’s scheme work, i.e., will he be able to correct double-bit errors? Briefly describe why or why not.
In search of a better code

• Problem: information about a particular message unit (bit, byte, ..) is captured in just a few locations, ie, the message unit and some number of parity units. So a small but unfortunate set of errors might wipe out all the locations where that info resides, causing us to lose the original message unit.

• Potential Solution: figure out a way to spread the info in each message unit throughout all the codewords in a block. Require only some fraction good codewords to recover the original message.

Spreading the wealth...

• Idea: oversampled polynomials. Let

\[ P(x) = m_0 + m_1x + m_2x^2 + \ldots + m_{k-1}x^{k-1} \]

where \( m_0, m_1, \ldots, m_{k-1} \) are the \( k \) message units to be encoded. Transmit value of polynomial at \( n \) different predetermined points \( v_0, v_1, \ldots, v_{n-1} \):

\[ P(v_0), P(v_1), P(v_2), \ldots, P(v_{n-1}) \]

Use any \( k \) of the received values to construct a linear system of \( k \) equations which can then be solved for \( k \) unknowns \( m_0, m_1, \ldots, m_{k-1} \). Each transmitted value contains info about all \( m_i \).

• Note that using integer arithmetic, the \( P(v) \) values are numerically greater than the \( m_i \) and so require more bits to represent than the \( m_i \). In general the encoded message would require a lot more bits to send than the original message!
Solving for the $m_i$

- Solving $k$ linearly independent equations for the $k$ unknowns (i.e., the $m_i$):

$$\begin{pmatrix}
1 & v_0 & v_0^2 & \cdots & v_0^{k-1} \\
1 & v_1 & v_1^2 & \cdots & v_1^{k-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & v_{k-1} & v_{k-1}^2 & \cdots & v_{k-1}^{k-1}
\end{pmatrix}
\begin{pmatrix}
m_0 \\
m_1 \\
\vdots \\
m_{k-1}
\end{pmatrix}
= 
\begin{pmatrix}
P(v_0) \\
P(v_1) \\
\vdots \\
P(v_{k-1})
\end{pmatrix}$$

- Solving a set of linear equations using Gaussian Elimination (multiplying rows, switching rows, adding multiples of rows to other rows) requires add, subtract, multiply and divide operations.

- These operations (in particular division) are only well defined over fields, eg, rational numbers, real numbers, complex numbers -- not at all convenient to implement in hardware.

**Finite Fields to the Rescue**

- Reed’s & Solomon’s idea: do all the arithmetic using a finite field (also called a Galois field). If the $m_i$ have $B$ bits, then use a finite field with order $2^B$ so that there will be a field element corresponding to each possible value for $m_i$.

- For example with $B = 2$, here are the tables for the various arithmetic operations for a finite field with 4 elements. Note that every operation yields an element in the field, i.e., the result is the same size as the operands.

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>×</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>-A</th>
<th>A⁻¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

$A + (-A) = 0$ \quad $A * (A^{-1}) = 1$
How many values to send?

- Note that in a Galois field of order $2^8$ there are at most $2^8$ unique values $v$ we can use to generate the $P(v)$ -- if we send more than $2^8$ values, some of the equations we might use when solving for the $m_i$ may not be linearly independent and we won't have enough information to find a unique solution for the $m_i$.

- Reed-Solomon codes use $n = 2^8-1$ ($n$ is the number of $P(v)$ values we generate and send).
  - For many applications $B = 8$, so $n = 255$
  - A popular R-S code is (255,223), i.e., a code block consists of 223 8-bit data bytes + 32 check bytes

Use for error correction

- If one of the $P(v_i)$ is received incorrectly, if it's used to solve for the $m_i$, we'll get the wrong result.

- So try all possible ($n$ choose $k$) subsets of values and use each subset to solve for $m_i$. Choose solution set that gets the majority of votes.
  - No winner? Uncorrectable error... throw away block.

- ($n,k$) code can correct up to $(n-k)/2$ errors since we need enough good values to ensure that the correct solution set gets a majority of the votes.
  - R-S (255,223) code can correct up to 16 symbol errors; good for error bursts: 16 consecutive symbols = 128 bits!
Erasures are special

- If a particular received value is known to be erroneous (an “erasure”), don’t use it all!
  - How to tell when received value is erroneous? Sometimes there’s channel information, e.g., carrier disappears.
  - See next slide for clever idea based on concatenated R-S codes

- \((n,k)\) R-S code can correct \(n-k\) erasures since we only need \(k\) equations to solve for the \(k\) unknowns.

- Any combination of \(E\) errors and \(S\) erasures can be corrected so long as \(2E + S \leq n-k\).

Example: CD error correction

- On a CD: two concatenated R-S codes

  32-byte block 32-byte block ... 32-byte block 32-byte block 32-byte block 32-byte block
  32-byte block 32-byte block ... 32-byte block 32-byte block 32-byte block 32-byte block
  28-byte block 28-byte block ... 28-byte block 28-byte block 28-byte block 28-byte block
  28-byte block 28-byte block ... 28-byte block 28-byte block 28-byte block 28-byte block
  24-byte block 24-byte block ... 24-byte block 24-byte block 24-byte block 24-byte block
  24-byte block 24-byte block ... 24-byte block 24-byte block 24-byte block 24-byte block

  De-interleave
  (32,28) code Handles up to 2 byte errors
  De-interleave
  (28,24) code Handles up to 4 byte erasures
  De-interleave

Result: correct up to 3500-bit error bursts (2.4mm on CD surface)