Source Coding

- Information & Entropy
- Variable-length codes: Huffman’s algorithm
- Adaptive variable-length codes: LZW
Where we've gotten to...

With channel coding (along with block numbers and CRC), we have a way to reliably send bits across a channel:

Next step: think about recoding the message bitstream to send the **information** it contains in as few bits as possible.
Many message streams use a “natural” fixed-length encoding: 7-bit ASCII characters, 8-bit audio samples, 24-bit color pixels.

If we’re willing to use variable-length encodings (message symbols of differing lengths) we could assign short encodings to common symbols and longer encodings to other symbols... this should shorten the average length of a message.
Measuring information content

Suppose you’re faced with N equally probable choices, and I give you a fact that narrows it down to M choices. Claude Shannon offered the following formula for the information you’ve received.

$$\log_2(\frac{N}{M}) \text{ bits of information}$$

Examples:
- information in one coin flip: \(\log_2(2/1) = 1\) bit
- roll of 2 dice: \(\log_2(36/1) = 5.2\) bits
- outcome of a Red Sox game: 1 bit
  (well, actually, are both outcomes equally probable?)

Information is measured in bits (binary digits) which you can interpret as the number of binary digits required to encode the choice(s).
When choices aren’t equally probable

When the choices have different probabilities \( (p_i) \), you get more information when learning of a unlikely choice than when learning of a likely choice.

\[
\text{Information from choice}_i = \log_2(1/p_i) \text{ bits}
\]

We can use this to compute the average information content taking into account all possible choices:

\[
\text{Average information content in a choice} = \sum p_i \cdot \log_2(1/p_i)
\]

This characterization of the information content in learning of a choice is called the information entropy or Shannon’s entropy.
Example

Average information content in a choice
\[= (.333)(1.58) + (.5)(1) + (2)(.083)(3.58)\]
\[= 1.626 \text{ bits}\]

Can we find an encoding where transmitting 1000 choices is close to 1626 bits on the average?

The “natural” fixed-length encoding uses two bits for each choice, so transmitting the results of 1000 choices requires 2000 bits.

<table>
<thead>
<tr>
<th>choice (_i)</th>
<th>(p_i)</th>
<th>(\log_2(1/p_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
<td>1/3</td>
<td>1.58 bits</td>
</tr>
<tr>
<td>“B”</td>
<td>1/2</td>
<td>1 bit</td>
</tr>
<tr>
<td>“C”</td>
<td>1/12</td>
<td>3.58 bits</td>
</tr>
<tr>
<td>“D”</td>
<td>1/12</td>
<td>3.58 bits</td>
</tr>
</tbody>
</table>
Variable-length encodings
(David Huffman, MIT 1950)

Use shorter bit sequences for high probability choices, longer sequences for less probable choices.

<table>
<thead>
<tr>
<th>choice_i</th>
<th>p_i</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
<td>1/3</td>
<td>11</td>
</tr>
<tr>
<td>“B”</td>
<td>1/2</td>
<td>0</td>
</tr>
<tr>
<td>“C”</td>
<td>1/12</td>
<td>100</td>
</tr>
<tr>
<td>“D”</td>
<td>1/12</td>
<td>101</td>
</tr>
</tbody>
</table>

Average information:

\[
= (.333)(2) + (.5)(1) + (2)(.083)(3)
= 1.666 \text{ bits}
\]

Transmitting 1000 choices takes an average of 1666 bits... better but not optimal.

To get a more efficient encoding (closer to information content) we need to encode sequences of choices, not just each choice individually. This is the approach taken by most file compression algorithms...
Huffman’s Coding Algorithm

• Begin with the set $S$ of symbols to be encoded as binary strings, together with the probability $P(x)$ for each symbol $x$. The probabilities sum to 1 and measure the frequencies with which each symbol appears in the input stream. In the example from the previous slide, the initial set $S$ contains the four symbols and their associated probabilities from the table.

• Repeat the following steps until there is only 1 symbol left in $S$:
  – Choose the two members of $S$ having lowest probabilities. Choose arbitrarily to resolve ties.
  – Remove the selected symbols from $S$, and create a new node of the decoding tree whose children (sub-nodes) are the symbols you've removed. Label the left branch with a "0", and the right branch with a "1".
  – Add to $S$ a new symbol that represents this new node. Assign this new symbol a probability equal to the sum of the probabilities of the two nodes it replaces.
Huffman Coding Example

- Initially \( S = \{ (A, 1/3) \ (B, 1/2) \ (C, 1/12) \ (D, 1/12) \} \)

- First iteration
  - Symbols in \( S \) with lowest probabilities: C and D
  - Create new node
  - Add new symbol to \( S = \{ (A, 1/3) \ (B, 1/2) \ (CD, 1/6) \} \)

- Second iteration
  - Symbols in \( S \) with lowest probabilities: A and CD
  - Create new node
  - Add new symbol to \( S = \{ (B, 1/2) \ (ACD, 1/2) \} \)

- Third iteration
  - Symbols in \( S \) with lowest probabilities: B and ACD
  - Create new node
  - Add new symbol to \( S = \{ (BACD, 1) \} \)

- Done
Huffman Codes – the final word?

• Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately.

• Huffman codes have the biggest impact on average message length when some symbols are substantially more likely than other symbols.

• You can improve the results by adding encodings for symbol pairs, triples, quads, etc. But the number of possible encodings quickly becomes intractable.

• Symbol probabilities change message-to-message, or even within a single message.

• Can we do adaptive variable-length encoding?
Adaptive Variable-length Codes

- Algorithm first developed by Lempel and Ziv, later improved by Welch. Now commonly referred to as the “LZW Algorithm”
- As message is processed a “string table” is built which maps symbol sequences to a fixed-length code
  - Table size = $2^{(\text{size of fixed-length code})}$
- Note: String table can be reconstructed by the decoder based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!
LZW Encoding

STRING = get input symbol
WHILE there are still input symbols DO
  SYMBOL = get input symbol
  IF STRING + SYMBOL is in the string table THEN
    STRING = STRING + SYMBOL
  ELSE
    output the code for STRING
    add STRING + SYMBOL to the string table
    STRING = SYMBOL
  END
END

output the code for STRING
lzw('abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc')

READ a
READ <a> b
XMIT 'a' ADD 0: ab
READ <b> c
XMIT 'b' ADD 1: bc
READ <c> a
XMIT 'c' ADD 2: ca
READ <a> bc
XMIT [ 0] ADD 3: abc
READ <c> ab
XMIT [ 2] ADD 4: cab
READ <b> ca
XMIT [ 1] ADD 5: bca
READ <a> bca
READ <a> bcab
XMIT [ 6] ADD 7: abcab
READ <a> bcab
XMIT [ 7] ADD 8: bcab
READ <b> cab
READ <c> abc
XMIT [ 8] ADD 10: cabc
READ <c> abca
XMIT [10] ADD 11: cabca
READ <a> bcabc
READ <a> abcab
READ <b> cabca
XMIT [12] ADD 14: bcabca
READ <a> bcabca
XMIT [ 9] ADD 15: abcabca
READ <a> bc
XMIT [ 3]
LZW Decoding

Read CODE
output CODE
STRING = CODE

WHILE there are still codes to receive DO
Read CODE
IF CODE is not in the translation table THEN
ENTRY = STRING + STRING[0]
ELSE
ENTRY = get translation of CODE
END
output ENTRY
add STRING+ENTRY[0] to the translation table
STRING = ENTRY
END
wzl(['a', 'b', 'c', 0, 2, 1, 3, 6, 5, 8, 4, 10, 7, 11, 9, 12, 3])

```
READ 'a'  RCV 'a'
READ 'b'  RCV 'b'    ADD 0:  ab
READ 'c'  RCV 'c'    ADD 1:  bc
READ [0]  RCV 'ab'   ADD 2:  ca
READ [2]  RCV 'ca'   ADD 3:  abc
READ [1]  RCV 'bc'   ADD 4:  cab
READ [3]  RCV 'abc'  ADD 5:  bca
READ [6]  RCV 'abca' ADD 6:  abca
READ [5]  RCV 'bca'  ADD 7:  abcab
READ [8]  RCV 'bcab' ADD 8:  bcab
READ [4]  RCV 'cab'  ADD 9:  bcabc
READ [10] RCV 'cabc' ADD 10: cabc
READ [7]  RCV 'abcab' ADD 11: cabca
READ [12] RCV 'abcabc' ADD 14: bcabca
READ [3]  RCV 'abc'  ADD 15: abcabca
```

String table reconstructed from received codes
Summary

• Source coding: recode message stream to remove redundant information, aka compression. Our goal: match data rate to actual information content.

• Information content from choice \( i \) = \( \log_2(1/p_i) \) bits

• Shannon’s Entropy: average information content on learning a choice = \( \sum p_i \cdot \log_2(1/p_i) \)

• Huffman’s encoding algorithm builds optimal variable-length codes when symbols encoded individually

• LZW algorithm implements adaptive variable-length encoding