Information, Compression, and Errors

3035/GZ01 Networked Systems
Kyle Jamieson
Lecture 2

Department of Computer Science
University College London
Last time: Building good links

1. Links go through hostile environments, wired and wireless
   - Errors will occur on links
   - How can we detect and correct these errors?

2. There is limited bandwidth available on any link
   - How can we compress the information we send over links?
Today

1. How do we measure information?
   - Information theory from first principles
2. Source coding (compression)
   - Huffman coding
   - Adaptive coding: Lempel-Ziv-Welch
3. Detecting and correcting errors
Today: Information, Compression, and Errors

- These techniques are pervasive throughout the network stack
- Based on theory, but broadly applicable in practice:
  - All layers of the Internet stack
  - Hard disk drives
  - Magnetic tape media
  - Optical media (CD, DVD, etc)
  - Satellite, mobile communications
  - Deep space, submarine communications
The simplest event: coin flip

“I am going to flip an ideal, unbiased coin”

**Before:** one of two outcomes will happen  
**After:** know which outcome  
Heads = 0, Tails = 1: intuitively, learn one bit of information

<table>
<thead>
<tr>
<th>Probability mass function</th>
<th>Event space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Probability</td>
</tr>
<tr>
<td>H</td>
<td>½</td>
</tr>
<tr>
<td>T</td>
<td>½</td>
</tr>
</tbody>
</table>
Equally-likely outcomes

Draw one card from a deck; tell me its suit. How much information are you giving me?

♠=00, ♣=01, ♥=10, ♦=11: learn two bits of information

Definition: an experiment with \( L \) equally probable outcomes yields \( \log_2(L) \) bits of information.

Examples:
Pick a random month of the year, tell me the month:
\( \log_2(12) = 3.39 \) bits
Pick a random day of the year, tell me the day:
\( \log_2(365) \approx 8.51 \) bits
Unequally-likely outcomes

Suppose we have stack of 32 cards: 4♠, 4♣, 8♥, 16♦

Experiment: Pick a card uniformly at random, tell me the suit. How much information are you giving me?

Initial uncertainty = \( \log_2(32) = 5 \) bits of information

“I picked a ♠.”

Final uncertainty = \( \log_2(4) = 2 \) bits of information

Learn 3 bits of information
Unequally-likely outcomes

Suppose we have stack of 32 cards: $4\times\spadesuit, 4\times\heartsuit, 8\times\clubsuit, 16\times\diamondsuit$

Experiment: Pick a card uniformly at random

Initial uncertainty $= \log_2(32) = 5$ bits of information

“I picked a ♦.”

Final uncertainty $= \log_2(16) = 4$ bits

Learn 1 bit of information
Summary: Unequally-likely outcomes

Suppose we have stack of 32 cards: 4♣, 4♠, 8♥, 16♦

Experiment: Pick a card uniformly at random

Initial uncertainty = \( \log_2(32) = 5 \) bits of information

“I picked a ♠.” \( \rightarrow \) 3 bits of information

“I picked a ♦.” \( \rightarrow \) 1 bit of information

“I picked the ace of ♠.” \( \rightarrow \) 5 bits of information

“-” \( \rightarrow 5-5 = 0 \) bits of information
Unequally-likely outcomes

P(pick a ♠) = 1/8 \rightarrow 3\text{ bits}
P(pick a ♣) = 1/8 \rightarrow 3\text{ bits}
P(pick a ♥) = 1/4 \rightarrow 2\text{ bits}
P(pick a ♦) = 1/2 \rightarrow 1\text{ bit}

Event space (outcome space) A
Partition into outcomes \{A_i\}, \ i = 1..L

**Definition:** Information learned from outcome A_i (bits) is:

\[ I(A_i) = \log_2 \left( \frac{1}{P(A_i)} \right) \]
Entropy measures information rate

Repeat the experiment of drawing a card from a deck; average information per card draw (bits) is:

\[ H = \sum_{i=1}^{L} P(A_i)I(A_i) = \sum_{i=1}^{L} P(A_i) \log_2 \left( \frac{1}{P(A_i)} \right) \]

(only sum over events where \( P(A_i) > 0 \))

Source \rightarrow Series of events

We call this the entropy of a discrete source.
Example #2: Biased coin toss

Source

Series of events \{H, T\}

\[ P(H) = p; \quad P(T) = 1 - p \]

\[
H = \sum_{i=1}^{L} P(A_i) \log_2 \left( \frac{1}{P(A_i)} \right) = p \log_2 \left( \frac{1}{p} \right) + (1 - p) \log_2 \left( \frac{1}{1 - p} \right)
\]
Today

1. How do we measure information?
2. Source coding (compression)
   – Huffman coding
   – Adaptive coding: Lempel-Ziv-Welch
3. Detecting and correcting errors
**Source coding problem**: How can we recover the information in $M$ exactly, while minimizing the size of $C$?
Applications of source coding

- **Image compression** \((M=\text{image}; \, \text{channel}=\text{web})\)
  - GIF widely used on the web (and source of controversy)
  - TIFF image file format
- **File compression** \((M=\text{file}; \, \text{channel}=\text{filesystem})\)
  - UNIX compress, gzip, bzip2
- **Media** \((M=\text{music or a movie}; \, \text{channel}=\text{CD or DVD})\)
Applying information theory to source coding

• We saw how to characterize information in:

  - Message $M$ is a sequence of symbols
    - Symbol $s \in \{ A, B, C, \ldots \}$ alphabet
    - Suppose we know symbol probabilities
    - A symbol = an event in our theoretical model
**Source coding: example**

**Sender**

\[ M \rightarrow \text{Source encoding} \rightarrow C \rightarrow \text{Channel} \rightarrow C \rightarrow \text{Source decoding} \rightarrow M \]

**Channel**

\[ H(M) = \frac{7}{12} \log_2 \left( \frac{12}{7} \right) + 2 \cdot \frac{1}{8} \log_2 (8) + 2 \cdot \frac{1}{12} \log_2 (12) \]

\[ = 1.8011 \text{ bits/symbol} \]

\[ \Rightarrow 1801 \text{ bits for 1000 symbols} \]

**Naïve encoding:** 3 bits/symbol

\[ \Rightarrow 3000 \text{ bits for 1000 symbols} \]

Can we improve?
Variable-length encoding

• David Huffman, student of Fano at MIT in 1950
  – Choice of assignment or term paper, outdid Fano-Shannon

• Key idea: higher probability symbols get shorter encodings than lower probability symbols

Samuel Morse
1791–1892
1. Insert all possible symbols into a priority queue:
   \[ S = \{(D, 1/12), (E, 1/12), (B, 1/8), (C, 1/8), (A, 7/12)\} \]
   – Pair each symbol with its frequency of occurrence
   – Priority queue orders by frequency of occurrence
2. Construct graph \( G \) with one node per symbol:
   \[
   G = \begin{array}{c}
   A & B & C & D & E
   \end{array}
   \]
Huffman coding merging step

\[ S = \{(D, \frac{1}{12}), (E, \frac{1}{12}), (B, \frac{1}{8}), (C, \frac{1}{8}), (A, \frac{7}{12})\} \]

- **Merging step**: remove two least frequently occurring symbols (D, E), create new node (DE) in \( G \) and \( S \) with sum of frequencies:

\[ S = \{(B, \frac{1}{8}), (C, \frac{1}{8}), (DE, \frac{1}{6}), (A, \frac{7}{12})\} \]

\[ G = \begin{array}{c}
A \quad B \quad C \quad DE \\
\end{array} \]

- Label new edges in \( G \) with 0, 1:

\[ \begin{array}{c}
\text{DE} \\
D \quad E \\
\end{array} \]

\[ 0 \rightarrow 1 \]
Huffman coding algorithm

\[ S = \{(B, 1/8), (C, 1/8), (DE, 1/6), (A, 7/12)\} \]
- Continue merging until there is one root:
\[ S = \{(BC, 1/4), (DE, 1/6), (A, 7/12)\} \]
Huffman coding algorithm

\[ S = \{(BC, 1/4), (DE, 1/6), (A, 7/12)\} \]

- Continue merging until there is one root:
\[ S = \{(BCDE, 5/12), (A, 7/12)\} \]

\[ G = \]

![Huffman Tree Diagram]
Huffman coding algorithm: result

\( S = \{(ABCDE, 1)\} \): terminate.

Read off encoding from \( G \):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>( P(X) )</th>
<th>Huffman encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7/12</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1/8</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>1/8</td>
<td>101</td>
</tr>
<tr>
<td>D</td>
<td>1/12</td>
<td>110</td>
</tr>
<tr>
<td>E</td>
<td>1/12</td>
<td>111</td>
</tr>
</tbody>
</table>

Average bits/Symbol: 1.833  
\( H(M) = 1.8011 \)
Decoding Huffman codes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>$P(X)$</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7/12</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1/8</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>1/8</td>
<td>101</td>
</tr>
<tr>
<td>D</td>
<td>1/12</td>
<td>110</td>
</tr>
<tr>
<td>E</td>
<td>1/12</td>
<td>111</td>
</tr>
</tbody>
</table>

- Do we need to add separators between codewords? No: there is no valid codeword that is a prefix of another.

“Greedy” decoding algorithm: $b_1b_2b_3b_4b_5b_6b_7…$
The Huffman code is a prefix code

Suppose codeword $X$ were a prefix of codeword $Z$.
So $Z = XY$.
Then in $G$, $Z$ is a child of $X$:

Contradiction: we only labeled leaves as valid codewords.

Therefore, no codeword is a prefix of another.
Huffman coding: limitations

1. Symbol frequencies changing within the message
   - Leads to suboptimal results
   - If message changes to all Bs, then 3 bits/symbol >> 1.8011

2. Assumption: coding each symbol individually
   • Improvement: coding recurring sequences of symbols
     - e.g. *Run-length encoding*: “AAAAABBC” → A × 5, B × 2, C

   • Are there different algorithms that harness these observations?
Lempel-Ziv-Welch (LZW) compression

- Begin with a codeword table, encoding all possible letters of alphabet

<table>
<thead>
<tr>
<th>String</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
<td>0</td>
</tr>
<tr>
<td>“B”</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>“Z”</td>
<td>25</td>
</tr>
</tbody>
</table>

- Key idea: learn sequences of characters in text
- Build up sequences first two characters long, then three, etc.
- Need some way of forgetting old sequences after some time

Algorithm

LZW-ENCODE(message)

\[
\text{cur}_\text{seq} \leftarrow \text{message}[1]
\]

\[
\text{for}\ \text{idx} \text{ in}\ 2..\text{len(message)}\ \text{do}:
\]

\[
c \leftarrow \text{message}[\text{idx}]
\]

\[
\text{if} \ \text{cur}_\text{seq} + c \ \text{in}\ \text{table}:
\]

\[
\text{cur}_\text{seq} \leftarrow \text{cur}_\text{seq} + c
\]

\[
\text{else}:
\]

\[
\text{output} \ \text{table}[:\text{cur}_\text{seq}]
\]

\[
\text{table}[:\text{cur}_\text{seq} + c] \leftarrow \text{size}(\text{table})
\]

\[
\text{cur}_\text{seq} \leftarrow c
\]

\[
\text{output} \ \text{table}[:\text{cur}_\text{seq}]
\]
LZW Example

Message: ANANDANDY

Initial table:

<table>
<thead>
<tr>
<th>String</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
<td>0</td>
</tr>
<tr>
<td>“B”</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>“Z”</td>
<td>25</td>
</tr>
</tbody>
</table>

Final table:

<table>
<thead>
<tr>
<th>String</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
<td>0</td>
</tr>
<tr>
<td>“B”</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>“Z”</td>
<td>25</td>
</tr>
<tr>
<td>“AN”</td>
<td>26</td>
</tr>
<tr>
<td>“NA”</td>
<td>27</td>
</tr>
<tr>
<td>“AND”</td>
<td>28</td>
</tr>
<tr>
<td>“DA”</td>
<td>29</td>
</tr>
<tr>
<td>“ANDY”</td>
<td>30</td>
</tr>
</tbody>
</table>

Output:


0, 13, 26, 3, 28, 24
Decoding LZW

- Sender transmits *table* to receiver; receiver inverts table to form *inverse_table*:

<table>
<thead>
<tr>
<th>Codeword</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“A”</td>
</tr>
<tr>
<td>1</td>
<td>“B”</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>25</td>
<td>“Z”</td>
</tr>
<tr>
<td>26</td>
<td>“AN”</td>
</tr>
<tr>
<td>27</td>
<td>“NA”</td>
</tr>
<tr>
<td>28</td>
<td>“AND”</td>
</tr>
<tr>
<td>29</td>
<td>“DA”</td>
</tr>
<tr>
<td>30</td>
<td>“ANDY”</td>
</tr>
</tbody>
</table>

- Increases overhead when message is short relative to table size!

Algorithm

LZW-DECODE(*symbols*)

```
for idx in 1..len(*symbols*) do:
    symbol = *symbols*[idx]
    str = *inverse_table*[symbol]
    output str
```

- Input to LZW-Decode:
  
  0, 13, 26, 3, 28, 24

- At point ❶, know that *encoder* joined “A” and “N” to make “AN”
- At point ❷, encoder joined “N” and “A” to make “NA”
Decoding LZW “on-the-fly”

- Only add a new table entry when we output a symbol.
- New table entry is the current symbol output, concatenated with the first letter of next symbol output.

```
LZW-ENCODE(message)
cur_seq ← message[1]
for idx in 2..len(message) do:
c ← message[idx]
if cur_seq + c in table:
cur_seq ← cur_seq + c
else:
output table[cur_seq]
table[cur_seq + c] ← size(table)
cur_seq ← c
output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>“A”</td>
<td>“AN” : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>“N”</td>
<td>“NA” : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>“AN”</td>
<td>“AND” : 28</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>“D”</td>
<td>“DA” : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>“AND”</td>
<td>“ANDY” : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>“Y”</td>
<td></td>
</tr>
</tbody>
</table>
Decoding LZW “on-the-fly”

- Only add a new table entry when we output a symbol
- New table entry is the current symbol output, concatenated with the first letter of next symbol output

```
LZW-ENCODE(message)
  cur_seq ← message[1]
  for idx in 2..len(message) do:
    c ← message[idx]
    if cur_seq + c in table:
      cur_seq ← cur_seq + c
    else:
      output table[cur_seq]
      table[cur_seq + c] ← size(table)
      cur_seq ← c
  output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>&quot;A&quot;</td>
<td>&quot;AN&quot; : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>&quot;N&quot;</td>
<td>&quot;NA&quot; : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>&quot;AN&quot;</td>
<td>&quot;AND&quot; : 28</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>&quot;D&quot;</td>
<td>&quot;DA&quot; : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>&quot;AND&quot;</td>
<td>&quot;ANDY&quot; : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>&quot;Y&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Decoding LZW “on-the-fly”

- Only add a new table entry when we output a symbol.
- New table entry is the current symbol output, concatenated with the first letter of next symbol output.

```python
LZW-ENCODE(message)
cur_seq ← message[1]
for idx in 2..len(message) do:
c ← message[idx]
if cur_seq + c in table:
cur_seq ← cur_seq + c
else:
  output table[cur_seq]
table[cur_seq + c] ← size(table)
cur_seq ← c
output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>“A”</td>
<td>“AN” : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>“N”</td>
<td>“NA” : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>“AN”</td>
<td>“AND” : 28</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>“D”</td>
<td>“DA” : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>“AND”</td>
<td>“ANDY” : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>“Y”</td>
<td></td>
</tr>
</tbody>
</table>
```
Decoding LZW “on-the-fly”

• Only add a new table entry when we output a symbol
• New table entry is the current symbol output, concatenated with the first letter of next symbol output

```
LZW-ENCODE(message)
cur_seq ← message[1]
for idx in 2..len(message) do:
c ← message[idx]
if cur_seq + c in table:
cur_seq ← cur_seq + c
else:
  output table[cur_seq]
table[cur_seq + c] ← size(table)
cur_seq ← c
output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>“A”</td>
<td>“AN” : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>“N”</td>
<td>“NA” : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>[AN]</td>
<td>[AND] : 28</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>[D]</td>
<td>“DA” : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>“AND”</td>
<td>“ANDY” : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>“Y”</td>
<td></td>
</tr>
</tbody>
</table>
Decoding LZW “on-the-fly”

- Only add a new table entry when we output a symbol.
- New table entry is the current symbol output, concatenated with the first letter of next symbol output.

```
LZW-ENCODE(message)
cur_seq ← message[1]
for idx in 2..len(message) do:
c ← message[idx]
  if cur_seq + c in table:
    cur_seq ← cur_seq + c
  else:
    output table[cur_seq]
    table[cur_seq + c] ← size(table)
    cur_seq ← c
output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>”A”</td>
<td>”AN” : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>”N”</td>
<td>”NA” : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>”AN”</td>
<td>”AND” : 28</td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>”AND”</td>
<td>”ANDY” : 30</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>”DA”</td>
<td>”DA” : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>”D”</td>
<td>”DA” : 29</td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>”AND”</td>
<td>”ANDY” : 30</td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>”AND”</td>
<td>”ANDY” : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>”Y”</td>
<td></td>
</tr>
</tbody>
</table>
Decoding LZW “on-the-fly”

- Only add a new table entry when we output a symbol
- New table entry is the current symbol output, concatenated with the first letter of next symbol output

```
LZW-ENCODE(message)
  cur_seq ← message[1]
  for idx in 2..len(message) do:
    c ← message[idx]
    if cur_seq + c in table:
      cur_seq ← cur_seq + c
    else:
      output table[cur_seq]
      table[cur_seq + c] ← size(table)
      cur_seq ← c
      output table[cur_seq]
```

<table>
<thead>
<tr>
<th>curseq</th>
<th>c</th>
<th>output</th>
<th>New table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>“A”</td>
<td>“AN” : 26</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>“N”</td>
<td>“NA” : 27</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td>“AN”</td>
<td>“AND” : 28</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>“D”</td>
<td>“DA” : 29</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AN</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Y</td>
<td>“AND”</td>
<td>“ANDY” : 30</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>“Y”</td>
<td></td>
</tr>
</tbody>
</table>
```
Decoding LZW “on-the-fly”: notes

- Input to LZW-Decode: 0, 13, 26, 3, 28, 24
  ① ② ③ ④ ⑤
- At point ①, joined “A” and “N” to make “AN” : 26
- At point ②, joined “N” and “A” to make “NA” : 27
- At point ③, joined “AN” and “D” to make “AND” : 28
- At point ④, joined “D” and “A” to make “DA” : 29
- At point ⑤, joined “AND” and “Y” to make “ANDY” : 30

<table>
<thead>
<tr>
<th>Codeword</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“A”</td>
</tr>
<tr>
<td>1</td>
<td>“B”</td>
</tr>
<tr>
<td>2</td>
<td>“C”</td>
</tr>
<tr>
<td>3</td>
<td>“D”</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>“N”</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>“Y”</td>
</tr>
<tr>
<td>25</td>
<td>“Z”</td>
</tr>
<tr>
<td>26</td>
<td>“AN”</td>
</tr>
<tr>
<td>27</td>
<td>“NA”</td>
</tr>
<tr>
<td>28</td>
<td>“AND”</td>
</tr>
<tr>
<td>29</td>
<td>“DA”</td>
</tr>
<tr>
<td>30</td>
<td>“ANDY”</td>
</tr>
</tbody>
</table>
Today

1. How do we measure information?
2. Source coding (compression)
   - Huffman coding
   - Adaptive coding: Lempel-Ziv-Welch
3. Detecting and correcting errors
   - Error detecting and correcting codes
   - Cyclic redundancy codes
   - The Internet checksum
Error control in the Internet stack

- Transport layer (L4)
  - Internet Checksum (IC) over TCP or UDP header and data

- Network layer (L3)
  - Internet Checksum (IC) over IP header only

- Link layer (L2)
  - Cyclic Redundancy Check (CRC)

- Physical layer (PHY)
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
A hypothetical network

- Sender transmits a packet of any length to the receiver through the network
- Network may take one of two actions:
  - Pass packet through correctly
  - Alter a *single* bit in the packet

**Problem:** errors happen, but receiver is oblivious!
Network may take one of two actions:
- Pass packet through correctly
- Alter a single bit in the packet

**Problem:** errors happen, but receiver is oblivious!

**Idea:** Add redundant bits to detect and/or correct errors
A simple error-detecting code

- Network may take one of two actions:
  - Pass packet through correctly
  - Alter a single bit in the packet

If an error happens, we can detect it.
Error **detecting** codes in general

- **Encoding**: The sender maps source bits ("0", "1") into **codewords** ("00", "11")
- **Protocol**: Codewords are the only bits sender may transmit
- In general the network may change any number of bits
- **Decoding** is a two-step process:
  1. The receiver maps received bits \(\rightarrow\) codewords
  2. The receiver maps codewords \(\rightarrow\) source bits and "—"
A simple error-correcting code

If an error happens, we can correct it by voting.
Error **correcting** codes in general

- **Encoding:** The sender maps source bits ("0", "1") into **codewords** ("000", "111")
- **Protocol:** Codewords are the only bits sender may transmit

Sender $\rightarrow$ Encode $\rightarrow$ Network $\rightarrow$ Decode $\rightarrow$ Receiver

- In general the network may change any number of bits
- **Decoding** is a two-step process:
  1. The receiver maps received bits $\rightarrow$ codewords
     - **Decoding rule:** Consider all possible codewords, choose the one that **most resembles** the received bits
  2. The receiver maps codewords $\rightarrow$ source bits
Which codeword most resembles received bits?

Definition: **Hamming distance** \(d(c_1, c_2)\) between codewords \(c_1\) and \(c_2\): count of bits different in \(c_1\) and \(c_2\)

Example: \(d(11100, 01101) = 2\)

- **Decoding rule** (restated): given received bits \(r\), compute Hamming distance \(d(r, c)\) for all codewords \(c\), choose codeword \(c^*\) that minimizes the Hamming distance
How many errors can we cope with?

Definition: \( d_{\text{min}} = \min_{\text{all codewords}} (c_1, c_2) d(c_1, c_2) \)

- Use \( d_{\text{min}} \) to reason about:
  - how many bit errors a code can detect:
    \[ d_{\text{min}} = 5 \] \( \therefore \) \( d_{\text{min}} - 1 \) in general
  - how many bit errors a code can correct:
    \[ d_{\text{min}} = 5 \] \[ \therefore \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor \] in general
Today

1. How do we measure information?
2. Source coding (compression)
   – Huffman coding
   – Adaptive coding: Lempel-Ziv-Welch
3. Detecting and correcting errors
   – Error detecting and correcting codes
     • Practical error correction: Block codes
   – Cyclic redundancy codes
   – The Internet checksum
Practical codes: Parity check

Choose parity bit such that total number of 1’s is even:

$k$ data bits \hspace{1cm} parity bit

0111000110000000 1

One bit error: 0111000110010000 1 error detected

Two bit errors: 0111000110000011 0 error missed

Three bit errors: 0111000101000000 0 error detected
Practical codes: Block codes

Split message into $k$-bit blocks, and add $n-k$ parity bits to the end of each block:

- $k$ bits
- $n-k$ bits

**data bits** | **parity bits**

**codeword:** $n$ bits

This is called an **$(n, k)$ block code**

Definition: the **rate** of a block code is $R = k/n$
Correcting errors with parity bits

\[ P_1 = D_1 \oplus D_3 \oplus D_4 \]
\[ P_2 = D_1 \oplus D_2 \oplus D_3 \]
\[ P_3 = D_2 \oplus D_3 \oplus D_4 \]

- Construct a block code to correct single-bit errors?
- Each data bit “hits” a different subset of parity bits
  - Error in \( D_1 \) ⇒ parity checks \( P_1 \) and \( P_2 \) fail
  - Error in \( D_4 \) ⇒ parity checks \( P_1 \) and \( P_3 \) fail
  - Error in one \( P_i \) ⇒ data is okay
How many parity bits do we need?

- **Goal**: correct at most one bit error
- **Main idea**: the subset of parity bit checks that fails tells us which (if any) bit is incorrect
- Assign each of $n$ bits to a subset of parity bits

How many non-empty subsets of parity bits? $2^{n-k} - 1$

$\Rightarrow n \leq 2^{n-k} - 1$

$\therefore \# \text{ parity bits} = n - k \geq \log_2(n+1)$

$\# \text{ parity bits} = \Omega(\log(n))$

**Codeword**: $n$ bits
Two-dimensional parity check

- Building block for more advanced codes
- Detects all one-, two-, and three-bit errors
- Detects most four-bit errors

\[
p_j = d_{j,1} \oplus d_{j,2} \oplus d_{j,3} \oplus d_{j,4}
\]

\[
q_j = d_{1,j} \oplus d_{2,j} \oplus d_{3,j} \oplus d_{4,j}
\]

\[
r = p_1 \oplus p_2 \oplus p_3 \oplus p_4
\]

- Lower bound: \# parity bits = \(\Omega(\log(n))\)
  - 2-D parity check: \# parity bits = \(\Theta(\sqrt{n})\)
  - Not asymptotically efficient for large codewords
Correcting bursts of errors

- **Interleaving** turns bursts of errors into single-bit errors in many codewords.
- Tools for correcting single-bit errors become tools for correcting bursts of errors.
- With enough interleaving, can assume independent probability of a bit error: **bit error rate (BER)**
  - Design more sophisticated codes (not in 6007)
Error detection at the link layer: CRC

- **Problem:** Errors overcome the error-correction capability of the PHY
  - Need an error detection code with a low probability of missing an error

- **Link layer (L2)**
  - Cyclic Redundancy Check (CRC)

- **Physical layer (PHY)**
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
Cyclic redundancy check (CRC)

- Most popular method for error detection at the link layer
- Often implemented in hardware at the link layer
- All arithmetic is modulo-2
  - Addition = subtraction = XOR operation
  - Multiplication using standard arithmetic rules

**Algorithm**
- Generator $G$: pattern of $r + 1$ bits agreed on beforehand
- Sender: choose $R$ such that $G$ divides $M$
- Receiver: divide $M$ by $G$; accept if no remainder
**CRC: Calculating $R$**

\[ M = D \cdot 2^r \oplus R \]

Want $R$ such that $\exists n$:

\[ D \cdot 2^r \oplus R = nG, \text{ or} \]
\[ D \cdot 2^r = nG \oplus R \]

Therefore, to calculate the CRC $R$, shift $D$ left by $r$ bits and divide by $G$; the remainder is $R$. 

![Diagram](image-url)
CRC: Calculating R—example

\[ G \]
\[
\begin{array}{c}
 1001 \\
 101011 \\
 1001 \\
 1010 \\
 1001 \\
 011 \\
 01100 \\
 1001 \\
 1010 \\
 1001 \\
 011 \\
 1001 \\
 1010 \\
 1001 \\
 011 \\
 R
\end{array}
\]

\[ d = 6, \ r = 3 \]
Properties of the CRC

• Most often implemented at the link layer in hardware
  – Shift register begins with all 0s, shift data in bit by bit, read CRC bits R from register after $D \cdot 2^r$ has been shifted in
  – Generator $G = 1011$

• Error detection: correctly detects
  – All single-bit errors (provided $G$ starts and ends with 1)
  – All double-bit errors (for well-chosen generators)
  – Any odd number of errors (for well-chosen generators)
  – All burst errors of length less than $g$
L3/L4 error detection: the Internet Checksum

- **Transport layer (L4)**
  - Internet Checksum (IC) over TCP or UDP header and data
  - End to end (E2E) checksum
- **Network layer (L3)**
  - Internet Checksum (IC) over IP header only
- **Link layer (L2)**
  - Cyclic Redundancy Check (CRC)
  - L2 CRC improves performance
- **Physical layer (PHY)**
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
The Internet checksum

• Implemented in software: simple and fast

• At the sender:
  1. Break packet into 16-bit words
  2. Sum the words together using one’s complement
  3. Negate the result of the sum → include this 16-bit checksum in the relevant header field

• At the receiver:
  1. Break received packet (including checksum) into 16-bit words
  2. Sum the words together using one’s complement
  3. Does the result equal \(1111\, 1111\, 1111\, 1111\)?
    • If no errors, yes
    • If errors, usually no
Internet checksum: Example

- One’s complement binary representation
  - Negation is bitwise inversion
  - To add, binary addition with carryout from the most significant bit to the least significant bit

<table>
<thead>
<tr>
<th>1000:</th>
<th>1100:</th>
<th>1101:</th>
<th>1110:</th>
<th>1111:</th>
</tr>
</thead>
<tbody>
<tr>
<td>−7</td>
<td>−3</td>
<td>−2</td>
<td>−1</td>
<td>−0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0000:</th>
<th>0001:</th>
<th>0010:</th>
<th>0011:</th>
<th>0100:</th>
<th>0101:</th>
<th>0110:</th>
<th>0111:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Example (using four-bit words, but principles are identical):

\[
\begin{align*}
\text{Word #1:} & \quad 0100 \\
\text{Word #2:} & \quad 0101 \\
\text{Word #3:} & \quad 1000 \\
\end{align*}
\]

\[
\begin{align*}
#1 + #2: & \quad 1001 \\
#1 + #2 + #3: & \quad 0010 \\
\text{Checksum:} & \quad 1101
\end{align*}
\]
Properties of the Internet checksum

• Incremental updates
  – Want to forward a packet with a small change?
  – Use sums of words being changed (before and after)
• Uses one word (16 bits) for any message length
• Weaker error detection properties than CRC
  – Checksum is the negation of sum of data, so any single-bit error in message or data is detectable, so $d_{\text{min}} \geq 2$
  – Undetectable double-bit error:

<table>
<thead>
<tr>
<th>Word #1:</th>
<th>0100</th>
<th>Word #1:</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word #2:</td>
<td>0101</td>
<td>Word #2:</td>
<td>0100</td>
</tr>
<tr>
<td>#1 + #2:</td>
<td>1001</td>
<td>#1 + #2:</td>
<td>1001</td>
</tr>
<tr>
<td>Checksum:</td>
<td>0110</td>
<td>Checksum:</td>
<td>0110</td>
</tr>
</tbody>
</table>
Medium Access Control: CDMA, ALOHA, and Ethernet

Pre-Reading: P & D Sections 2.6, 2.7, Ethernet Paper (online)

NEXT TIME