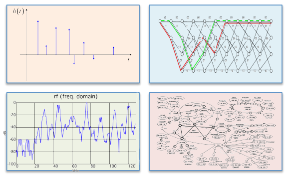


INTRODUCTION TO EECS II
DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2010
Lecture #23: Adaptive Source Coding
December 1, 2010

- Information & entropy
- Adaptive codes: LZW algorithm
- Lossy compression (JPEG overview)

LZW: An Adaptive Variable-length Source Code

- Algorithm first developed by Ziv and Lempel, 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**(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!
- This idea is a crucial one – build a table (mapping between codes and strings) but DON'T transmit it!

LZW Encoding (Compression)

```
for i in xrange(256): table.append(chr(i))    # code for each 8-bit character is its ASCII value
f = open(sys.argv[1], 'r')
data = f.read()
compressed = []    # final output compressed list of codes (each code is some string)

string = data[0]
for symbol in data[1:]:
    if (string + symbol) in table:    # "+" is concatenation, of course!
        string = string + symbol
    else:
        code = table.index(string)    # code for "string", from table
        compressed.append(code)
        table.append(string+symbol)
        string = symbol
compressed.append(table.index(string)) # don't forget last code!
return compressed
```

lzw('abcabcabcabcabcabcabcabcabcabcabcabc')

<...> indicates current value of STRING

```
<> READ a
<a> READ b, ab not in table
 XMIT 'a'  ADD  0: ab
<b> READ c, bc not in table
 XMIT 'b'  ADD  1: bc
<c> READ a, ca not in table
 XMIT 'c'  ADD  2: ca
<a> READ b, ab in table
<ab> READ c, abc not in table
 XMIT [ 0] ADD  3: abc
<c> READ a, ca in table
<ca> READ b, cab not in table
 XMIT [ 2] ADD  4: cab
<b> READ c, bc in table
<bc> READ a, bca not in table
 XMIT [ 1] ADD  5: bca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ a, abca not in table
 XMIT [ 3] ADD  6: abca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ,a,abca in table
<abca> READ b, abcab not in table
 XMIT [ 6] ADD  7: abcab
<b> READ c, bc in table
<bc> READ a, bca in table
<bca> READ b, bcab not in table
 XMIT [ 5] ADD  8: bcab
<b> READ c, bc in table
<bc> READ a, bca in table
<bca> READ b, bcab in table
<bcab> READ c, bcabc not in table
 XMIT [ 8] ADD  9: bcabc
<c> READ a, ca in table
<ca> READ b, cab in table
<cab> READ c, cabc not in table
 XMIT [ 4] ADD 10: cabc
<c> READ a, ca in table
<ca> READ b, cab in table
<cab> READ c, cabc in table
<cabc> READ a, cabca not in table
 XMIT [10] ADD 11: cabca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ a, abca in table
<abca> READ b, abcab in table
<abcab> READ c, abcabc not in table c
 XMIT [ 7] ADD 12: abcabc
<c> READ –end—
 XMIT 'c'
```

LZW Decoding - Simple (but Incorrect!)

```
dtable = []
for i in xrange(256): dtable.append(chr(i))    # same init table as compressor
code = compressed[0]    # compressed is a list of codes input to decoding function
decompressed = dtable[code]
string = decompressed

for code in compressed[1:]:
    entry = dtable[code]

    dtable.append(string + entry[0])
    string = entry
    decompressed = decompressed + entry

return decompressed
```

LZW Decoding (Decompression) - Correct Version

```
dtable = []
for i in xrange(256): dtable.append(chr(i))    # same init table as compressor
code = compressed[0]    # compressed is a list of codes input to decoding function
decompressed = dtable[code]
string = decompressed

for code in compressed[1:]:
    if code < len(dtable):     # i.e., code is in dtable
        entry = dtable[code]
    else:    # this part is subtle – why wouldn't code be in table?
        entry = string + string[0]

    dtable.append(string + entry[0])
    string = entry
    decompressed = decompressed + entry

return decompressed
```

```
wzl(['a', 'b', 'c', 0, 2, 1, 3, 6, 5, 8, 4, 10, 7, 'c'])

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 'c'  RCV 'c'     ADD 12: abcabc
```

String table reconstructed from received codes

Lossless vs. Lossy Compression

- Huffman and LZW encodings are *lossless*, i.e., we can reconstruct the original bit stream exactly: bits$_{OUT}$ = bits$_{IN}$.
  - What we want for "naturally digital" bit streams (documents, messages, datasets, …)
- Any use for *lossy* encodings: bits$_{OUT}$ ≈ bits$_{IN}$?
  - "Essential" information preserved
  - Appropriate for sampled bit streams (audio, video) intended for human consumption via imperfect sensors (ears, eyes).

Perceptual Coding

- Start by evaluating input response of bitstream consumer (eg, human ears or eyes), i.e., how consumer will perceive the input.
  - Frequency range, amplitude sensitivity, color response, …
  - Masking effects
- Identify information that can be removed from bit stream without perceived effect, e.g.,
  - Sounds outside frequency range, or masked sounds
  - Visual detail below resolution limit (color, spatial detail)
  - Info beyond maximum allowed output bit rate
- Encode remaining information efficiently
  - Use DCT-based transformations (real instead of complex)
  - Quantize DCT coefficients
  - Entropy code (eg, Huffman encoding) results

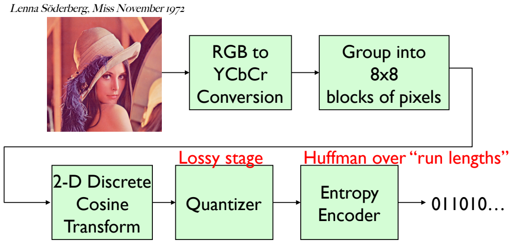
Perceptual Coding Example: Images

- Characteristics of our visual system
  ⇒ opportunities to remove information from the bit stream
  - More sensitive to changes in luminance than color
    ⇒ spend more bits on luminance than color (encode separately)
  - More sensitive to large changes in intensity (edges) than small changes
    ⇒ quantize intensity values
  - Less sensitive to changes in intensity at higher spatial frequencies
    ⇒ use larger quanta at higher spatial frequencies
- So to perceptually encode image, we would need:
  - Intensity at different spatial frequencies
  - Luminance (grey scale intensity) separate from color intensity

JPEG Image Compression

JPEG = Joint Photographic Experts Group

Lenna Söderberg, Miss November 1972



Performed for each 8x8 block of pixels

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: adaptive lossless compression – efficient on text, used in GIF, etc.
- JPEG: lossy compression; *perceptual coding*