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 = $2^m$ (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)

```python
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()
f = open(sys.argv[1], 'r')
for i in xrange(256): dtable.append(chr(i))    # same init table as compressor

string = data[0]
compressed = []     # final output compressed list of codes (each code is some string)

string = symbol
table.append(string+symbol)
compressed.append(code)
code = table.index(string)
string = string + symbol
for i in xrange(256): dtable.append(chr(i))    # code for each 8-bit character is its ASCII value
```

LZW Decoding (Decompression) – Correct Version

```python
dtable = []
code = compressed[0]    # compressed is a list of codes input to decoding function
dtable = []
for code in compressed[1:]:
    if code < len(dtable):
        entry = dtable[code]
    else:
        dtable.append(string + entry[0])
        string = entry
        decompressed = decompressed + entry
return decompressed
```

LZW Decoding - Simple (but Incorrect!)

```python
dtable = []
for i in range(256): dtable.append(chr(i))    # same init table as compressor
for symbol in data[1:]:
    string = symbol
    if (string + symbol) in table:    # concatenation, of course!
        string = string + symbol
    else:
        code = table.index(string)
        dtable.append(string + entry[0])
        string = entry
        decompressed = decompressed + entry
return decompressed
```

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 = $2^m$ (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!
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: \( \text{bits}_{\text{OUT}} = \text{bits}_{\text{IN}} \).
- What we want for "naturally digital" bit streams (documents, messages, datasets, …)
  - Any use for **lossy** encodings: \( \text{bits}_{\text{OUT}} \approx \text{bits}_{\text{IN}} \)?
  - "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 (e.g., 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 (e.g., 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

  - RGB to YCbCr Conversion
  - Group into 8x8 blocks of pixels
  - 2D Discrete Cosine Transform
  - Quantizer
  - Entropy Encoder
  - Huffman over "run lengths"
  - 011010…

  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 = \( 2p_i \log_2(1/p_i) \).
- 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