Ch. 2: Compression Basics
Multimedia Systems

Prof. Thinh Nguyen (Based on Prof. Ben Lee’s Slides)
Oregon State University
School of Electrical Engineering and Computer Science
Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding
Why Compression?

Uncompressed  Compressed

MB

100,000

10,000

1,000

100

10

1

Uncompressed
10K
2K
4K
60K
79K

Compressed
4.5M
1.6M
18M
53M
342M
1.1G

Text Fax Computer Videoconf Audio Image 35MM Video- ISDN Slide -conference Quality Stereo -erence Quality ISDN Video- Quality TV TV-Quality Quality High Quality

1Page 1Page 1Image 1Slide 5 Minute Presentation

18M 53M 342M 1.1G 620M 37G
Complexity Example

- “NTSC” Quality Computer Display
  - $640 \times 480$ pixel image
  - 3 bytes per pixel (Red, Green, Blue)
  - 30 Frames per Second

- Bandwidth requirement
  - 26.4 MB/second
  - Exceeds bandwidth of almost all disk drives

- Storage requirement
  - CD-ROM would hold 25 seconds worth
  - 30 minutes would require 46.3 GB

- Some form of compression required!
Compression Methods

- JPEG for still images
- MPEG (1/2/4) for audio/video playback and VoD (retrieval mode applications)
- DVI (Digital Visual Interactive) for still and continuous video applications (two modes of compression)
  - Presentation Level Video (PLV) - high quality compression, but very slow. Suitable for applications distributed on CD-ROMs
  - Real-time Video (RTV) - lower quality compression, but fast. Used in video conferencing applications.
Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding

Chapter 2: Compression Basics
Compression is used to save storage space, and/or to reduce communications capacity requirements.

Classification

- **Lossless**
  - No information is lost.
  - Decompressed data are identical to the original uncompressed data.
  - Essential for text files, databases, binary object files, etc.

- **Lossy**
  - Approximation of the original data.
  - Better *compression ratio* than lossless compression.
  - Tradeoff between compression ratio and fidelity.
  - e.g., Audio, image, and video.
Classification (2)

- **Entropy Coding**
  - Lossless encoding
  - Used regardless of media’s specific characteristics
  - Data taken as a simple digital sequence
  - Decompression process regenerates data completely
  - e.g., run-length coding, Huffman coding, Arithmetic coding

- **Source Coding**
  - Lossy encoding
  - Takes into account the semantics of the data
  - Degree of compression depends on data content
  - e.g., content prediction technique - DPCM, delta modulation

- **Hybrid Coding** (used by most multimedia systems)
  - Combine entropy with source encoding
  - e.g., JPEG, H.263, DVI (RTV & PLV), MPEG-1, MPEG-2, MPEG-4
Symbol Codes

- Mapping of letters from one alphabet to another
  - ASCII codes: computers do not process English language directly. Instead, English letters are first mapped into bits, then processed by the computer.
  - Morse codes
  - Decimal to binary conversion

- Process of mapping from letters in one alphabet to letters in another is called coding.
Symbol Codes

- Symbol code is a simple mapping.
- Formally, symbol code $C$ is a mapping $X \rightarrow D^+$
  - $X$ = the source alphabet
  - $D$ = the destination alphabet
  - $D^+$ = set of all finite strings from $D$
- Example:
  - $\{X, Y, Z\}$ $\{0, 1\}^+$, $C(X) = 0$, $C(Y) = 10$, $C(Z) = 1$
- **Codeword**: the result of a mapping, e.g., 0, 10, …
- **Code**: a set of valid codewords.
Symbol Codes

- Extension: $C^+$ is a mapping $X^+ \rightarrow D^+$ formed by concatenating $C(x_i)$ without punctuation
  - Example: $C(XXYXZY) = 001001110$

- Non-singular: $x_1 \neq x_2 \rightarrow C(x_1) \neq C(x_2)$

- Uniquely decodable: $C^+$ is non-singular
  - That is $C^+(x^+)$ is unambiguous
**Prefix Codes**

- Instantaneous or Prefix code:
  - No codeword is a prefix of another

- Prefix → uniquely decodable → non-singular

- Examples:

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>11</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>Y</td>
<td>00</td>
<td>10</td>
<td>110</td>
<td>011</td>
</tr>
<tr>
<td>Z</td>
<td>01</td>
<td>11</td>
<td>1110</td>
<td>0111</td>
</tr>
</tbody>
</table>
Form a $D$-ary tree where $D = |D|$

- $D$ branches at each node
- Each branch denotes a symbol $d_i$ in $D$
- Each leaf denotes a source symbol $x_i$ in $X$
- $C(x_i) =$ concatenation of symbols $d_i$ along the path from the root to the leaf that represents $x_i$
- Some leaves may be unused

\[ C(WWYXZ) = \]
Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding
Entropy and Information Theory

- Exploit statistical redundancies in the data to be compressed.
- The average information per symbol in a file, called *entropy* $H$, is defined as

$$
H = \sum_{i} p_i \log_2 \frac{1}{p_i}
$$

where, $p_i$ is the probability of the $i^{th}$ distinct symbol.
- Entropy is the lower bound for lossless compression, i.e., when the occurring probability of each symbol is fixed, each symbol should be represented with at least with $H$ bits on average.
Entropy

- The closer the compressed bits per symbol to entropy, the better the compression.
- For example, in an image with uniform distribution of gray-level intensity, i.e., $p_i = 1/256$, then the number of bits needed to code each gray level is 8 bits. The entropy of this image is 8. **No compression in this case!**
- We can show that, for an optimal code, the length of a codeword $i, L_i$, satisfies
  \[ \log_2\left(\frac{1}{P_i}\right) \leq L_i < \log_2\left(\frac{1}{P_i}\right) + 1 \]
- Thus, average length of a code word, $E[L]$, satisfies
  \[ H(X) \leq E[L] < H(X) + 1 \]
Entropy and Compression

- In compression, we are interested in constructing an optimum code, i.e., one that gives the minimum average length for the encoded message.
- Easy if the probability of occurrence for all the symbols are the same, e.g., last example.
- But if symbols have different probability of occurrence..., e.g., Alphabet.
- See Huffman and Arithmetic coding.
Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding
Lossless Compression Techniques

- Run length encoding (RLE)
- Variable Length Coding (VLC)
  - Huffman encoding
  - Arithmetic encoding
- Dictionary-Based Coding
  - Lempel-Ziv-Welch (LZW) encoding
Run-Length Encoding

- Redundancy is exploited by not transmitting consecutive pixels or character values that are equal.
- The value of long runs can be coded once, along with the number of times the value is repeated (length of the run).
- Particularly useful for encoding black and white images where the data units would be single bit pixels, e.g., fax.

Methods
- Binary RLE
- Packbits
Binary RLE

- Transmit length of each run of 0’s.
- Do not transmit runs of 1’s.
- For a run length $n \times 2^k - 1 + b$, where $k$ is the number of bits used to encode the run length
  - Transmit $n \times k$ 1’s, followed by the value $b$ encoded in $k$ bits.
- Two consecutive 1’s are implicitly separated by a zero-length run.
- Appropriate for bi-level images, e.g., FAX.
Binary RLE

Suppose 4 bits are used to represent the run length

What if stream started with 1? It would start with 0000!
Binary RLE Performance

- Worst case behavior:
  - If each transition requires $k$-bits to encode, and we can have a transition on each bit, $k-1$ of those bits are wasted. Bandwidth expanded by $k$ times!

- Best case behavior:
  - A sequence of $2^k-1$ consecutive 0 bits can be represented using $k$ bits. Compression ratio is roughly $2^k-1/k$
    - e.g., if $k = 8$, $2^k - 1 = 255$
    - $255:8 \gg 32:1$ compression (lossless).
RLE - Packbits

- Used by Apple Macs.
- One of the standard forms of compression available in TIFF format (TIFF compression type 32773).
  - TIFF = Tag Image File format.
- Each run is encoded into bytes as follows:

```plaintext
<table>
<thead>
<tr>
<th>Count</th>
<th>Value0</th>
<th>...</th>
<th>Value_{n-1}</th>
</tr>
</thead>
</table>
```

- Header Byte
- Trailer Bytes
- 0: Literal run
- 1: Fill run
Packbits - Literal Run

- Header byte: MSB = 0 and 7 bits describe the length of the run of literal bytes.
- Trailer bytes: Characters are stored verbatim.
- Example: “Mary had a little lamb”
  <22>, M, a, r, y, , h, a, d, , a, , l, i, t, t, l, e, , l, a, m, b
Packbits - Fill Run

- Header byte: MSB = 1 and 7 bits describe the length of the run.
- Trailer byte: Byte that should be copied
- Example: “AAAAABBBBB”
  \(<128+4>A<128+5>B\)
- What about for “ABCCCCCCCCCCDEFFFFFGGG”?
  \(<129>A<129>B<137>C<129>D<129>E<132>F<131>G\)
Packbits Performance

- **Worst case behavior**
  - If entire image is transmitted without compression, the overhead is a byte (for the header byte) for each 128 bytes.
  - Alternating runs add more overhead.
  - Poorly designed encoder could do worse.

- **Best case behavior**
  - If entire image is redundant, then 2 bytes are used for each 128 consecutive bytes. This provides a compression ratio of 64:1.
  - Achieves a typical compression ratio of 2:1 to 5:1.
Huffman Coding

- Input values are mapped to output values of varying length, called *variable-length code* (VLC):
  - Most probable inputs are coded with fewer bits.
  - No code word can be prefix of another code word.
Huffman Code Construction

- Initialization: Put all nodes (values, characters, or symbols) in a sorted list.
- Repeat until the list has only one node:
  - Combine the two nodes having the lowest frequency (probability). Create a parent node assigning the sum of the children’s probabilities and insert it into the list.
  - Assign code 0, 1 to the two branches, and delete children from the list.
Example of Huffman Coding

P(000) = .5 → .5 → .5 → .5 → .5 → .5 → 1.0

P(001) = .2 → .2 → .2 → .2 → .3 → .5

P(010) = .1 → .1 → .12 → .18 → .2

P(011) = .08 → .08 → .1 → .12

P(100) = .05 → .07 → .08

P(101) = .04 → .05

P(110) = .03

000 → 0
001 → 11
010 → 1000
011 → 1001
100 → 1011
101 → 10100
110 → 10101
Huffman Decoding

\[
\begin{align*}
P(000) &= .5 \\
P(001) &= .2 \\
P(010) &= .1 \\
P(011) &= .08 \\
P(100) &= .05 \\
P(101) &= .04 \\
P(110) &= .03
\end{align*}
\]
Efficiency of Huffman Coding

- Entropy $H$ of the previous example:
  - $0.5 + 0.2 \times 2.32 + 0.1 \times 3.21 + 0.08 \times 3.64 + 0.05 \times 4.32 + 0.04 \times 4.64 + 0.03 \times 5.06 = 2.129$

- Average number of bits $E[L]$ used to encode a symbol:
  - $1 \times 0.5 + 2 \times 0.2 + 4 \times 0.1 + 4 \times 0.08 + 4 \times 0.05 + 5 \times 0.04 + 5 \times 0.03 = 2.17$

- Efficiency ($2.129/2.17$) of this code is 0.981. Very close to optimum!

- If we had simply used 3 bits
  - $3 \times (0.5 + 0.2 + 0.1 + 0.08 + 0.05 + 0.04 + 0.03) = 3$

- So Huffman code is better!
Adaptive Huffman Coding

- The previous algorithm requires the statistical knowledge, which is often not available (e.g., live audio, video)
- Even if it’s available, it could be heavy overhead if the coding table is sent before the data.
  - This is negligible if the data file is large.
- Solution - adjust the Huffman tree on the fly, based on data previously seen.

Encoder
Initial_code();
while ((c=getc(input))!=eof {
    encode (c,output);
    update_tree(c);
}

Decoder
Initial_code();
while ((c=decode(input)) !=eof){
    putc (c,output);
    update_tree(c);
}
Adaptive Huffman Coding

- `initial_code()` assigns symbols with some initially agreed-upon codes without knowing the frequency counts for them.
  - e.g., ASCII
- `update_tree()` constructs an adaptive Huffman tree.
  - Huffman tree must maintain its sibling property
    - All nodes (internal and leaf) are arranged in the order of increasing counts => from left to right, bottom to top.
  - When swap is necessary, the farthest nodes with count $N$ is swapped with the nodes whose count has just been increased to $N+1$. 
Adaptive Huffman Coding (1)

Sibling Property
Increasing order of weights

Hit on A
Adaptive Huffman Coding (2)

1. w=2+1  2. w=2  3. w=2  4. w=2

5. w=4  6. w=4  7. w=8

A  B  C  D

E

9. w=18

2nd hit on A

Swap

Farthest from A with count = N = 2

Encoded symbols:
A=011, B=001, C=010
D = 000, E = 1
Adaptive Huffman Coding (3)

Encoded symbols:
A=10, B=1111, C=110
D = 1110, E =0

2 more hits on A
Adaptive Huffman Coding - Initialization

- Initially there is no tree.
- Symbols sent for the first time use fixed code:
  - e.g., ASCII
- Any new node spawned from “New”.
- Except for the first symbol, any symbol sent afterwards for the first time is preceded by a special symbol “New”.
  - Count for “New” is always 0.
  - Code for symbol “New” depends on the location of the node “New” in the tree.
- Code for symbol sent for the first time is
  - code for “New” + fixed code for symbol
- All other codes are based on the Huffman tree.
Initialization - Example

- Suppose we are sending AABCDAD...
  - Fixed code: A = 00001, B = 00010, C = 00011, D = 00100, ...
  - “New” = 0

- Initially, there is no tree.
- Send **A**ABCDA
- Update tree for “A”
  - Output = 00001
    - Decode 00001 = A
    - Update tree for “A”

- Send **A**ABCDA
- Update tree for “A”
  - Output = 1
    - Decode 1 = A
    - Update tree for “A”
Initialization - Example

- Send AABBCDA
- Update tree for “B”

```
  w=3
     /
    /
   /
 0 1
```

Output = 0 00010

- Decode 0 00010 = B
- Update tree for “B”

```
  w=3
     /
    /
   /
 0 1
```

- Send AABBCDA
- Update tree for “C”

```
  w=4
     /
    /
   /
 0 1
```

Output = 00 00011

- Decode 00 00011 = C
- Update tree for “C”

```
  w=4
     /
    /
   /
 0 1
```
Initialization - Example

- Send AABCDA
- Update tree for “D”

Output = 000 00100

- Decode 000 00100 = D
- Update tree for “D”

Exchange
Initialization - Example

- Send AABCDA
- Update tree for “A”

Output = 0

- Decode 0 = A
- Update tree for “A”

Final transmitted code

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Chapter 2: Compression Basics
Why not always use Huffman code? (since it is optimal)

- **Good**
  - It is optimal, i.e., shortest possible symbol code

- **Bad**
  - Bad for skewed probability
  - Employ block coding helps, i.e., use all $N$ symbols in a block. The redundancy is now $I/N$. However,
    - Must re-compute the entire table of block symbols if the probability of a single symbol changes.
    - For $N$ symbols, a table of $|X|^N$ must be pre-calculated to build the tree
    - Symbol decoding must wait until an entire block is received
Arithmetic Coding

- Developed by IBM (too bad, they didn’t patent it)
- Majority of current compression schemes use it

- Good:
  - No need for pre-calculated tables of big size
  - Easy to adapt to change in probability
  - Single symbol can be decoded immediately without waiting for the entire block of symbols to be received.
Arithmetic Coding

- Each symbol is coded by considering prior data
  - Relies on the fact that coding efficiency can be improved when symbols are combined.
  - Yields a single code word for each string of characters.
  - Each symbol is a portion of a real number between 0 and 1.

- Arithmetic vs. Huffman
  - Arithmetic encoding does not encode each symbol separately; Huffman encoding does.
  - Compression ratios of both are similar.
Arithmetic Coding Example

Example: $P(a) = 0.5$, $P(b) = 0.25$, $P(c) = 0.25$

Coding the sequence “b,a,a,c”

Binary rep. of final interval

$[0.0111, 0.011101)$
Arithmetic Coding Algorithm

BEGIN
  low = 0.0; high = 1.0; range = 1.0;
  while (symbol != terminator) {
    low = low + range * Range_low(symbol);
    high = low + range * Range_high(symbol);
    range = high - low;
  }
  output a code so that low <= code < high;
END
Arithmetic Coding

- Size of the final subinterval, i.e., range $s$, is $0.4375 - 0.453125 = 0.015625$, which is also determined by the product of the probabilities of the source message, $P(b) \times P(a) \times P(a) \times P(c)$.

- The number of bits needed to specify the final subinterval is given by at most $\lceil -\log_2 s \rceil = 6$ bits.

- Choose the smallest binary fraction that lies within $[0.0111, 0.011101)$.
  - Code is 0111 (from 0.0111)!

- Entropy $H$ is $0.5(1) + 0.25(2) + 0.25(2) = 1.5$ bits/symbol

- 4 symbols encoded with 6 bits gives 1.5 bits/symbol!

- Fixed length code would have required 2 bits/symbol.
Code Generator for Encoder

BEGIN
    code = 0;
    k = 1;
    while (value(code) < low) {
        assign 1 to the kth binary fraction bit;
        if (value(code) > high)
            replace the kth bit by 0;
        k = k + 1;
    }
END
Code Generator Example

For the previous example low = 0.4375 and high = 0.453125.

- \( k = 1 \): NIL < low, 0.1 (0.5) > high => continue (0.0)
- \( k = 2 \): 0.0 (0.0) < low, 0.01 (0.25) < high => continue (0.01)
- \( k = 3 \): 0.01 (0.25) < low, 0.011 (0.375) < high => continue (0.011)
- \( k = 4 \): 0.011 (0.375) < low, 0.0111 (0.4375) < high => continue (0.0111)
- \( k = 5 \): 0.0111 (0.4375) = low => terminate

Output code is 0111 (from 0.0111)
Decoding

BEGIN
    get binary code and convert to decimal value = value(code);
    do {
        find a symbol so that
        \( \text{Range}_\text{low}(\text{symbol}) \leq \text{value} < \text{Range}_\text{high}(\text{symbol}) \)
        output symbol;
        low = \text{Range}_\text{low}(\text{symbol});
        high = \text{Range}_\text{high}(\text{symbol});
        range = high - low;
        value = \left( \text{value} - \text{low} \right) / \text{range}
    }
    until symbol is a terminator
END
Decoding Example

\[
\begin{align*}
0.4375 - 0.25 &= 0.25 \\
\text{P}(b) &= 0.75 \\
\text{P}(a) &= 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{P}(a) &= 0.5 \\
\text{P}(b) &= 0.5 \\
\text{P}(c) &= 0 \\
\end{align*}
\]
Arithmetic Coding

• Need to know when to stop, e.g., in the previous example, 0 can be c, cc, ccc, etc. Thus, a special character is included to signal end of message.

• Also, precision can grow without bound as length of message grows. Thus, fixed precision registers are used with detection and management of overflow and underflow.

• Thus, both use of message terminator and fixed-length registers reduces the degree of compression.
Both encoder and decoder are assumed to have the same dictionary (table)

Encoder codes the index

<table>
<thead>
<tr>
<th>index</th>
<th>pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>ab</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>abc</td>
</tr>
</tbody>
</table>
Ziv-Lempel Coding (ZL or LZ)


- Adaptive dictionary technique.
  - Store previously coded symbols in a buffer.
  - Search for the current sequence of symbols to code.
  - If found, transmit buffer offset and length.
Output triplet <offset, length, next>
Transmitted to decoder: 8 3 d 0 0 e 1 2 f

If the size of the search buffer is $N$ and the size of the alphabet is $M$ we need

$$\lfloor \log(N + 1) \rfloor + \lfloor \log(N + 1) \rfloor + \lfloor \log M \rfloor$$

bits to code a triplet.

**Variation:** Use a VLC to code the triplets!
Drawbacks of LZ77

- Repetitive patterns with a period longer than the search buffer size are not found.
- If the search buffer size is 4, the sequence
  \[ a \ b \ c \ d \ e \ a \ b \ c \ d \ e \ a \ b \ c \ d \ e \ a \ b \ c \ d \ e \ \ldots \]
  will be expanded, not compressed.
Lempel-Ziv-Welch Coding

- Patterns that occur frequently can be substituted by single bytes.
  - e.g. “begin”, “end”, “if”…
- Dynamically builds a dictionary of phrases from the input stream (in addition to the dictionary of 256 ASCII characters).
- Checks to see if a phase is recorded in the dictionary
  - If not, add the phase in the dictionary
  - Otherwise, output a code for that phrase.
- Can be applied to still images, audio, video.
- A good choice for compressing text files (used in GIF, .Z, and .gzip compression algorithms).
LZW Compression

Example: Input string is "^WED^WE^WEE^WEB^WET".

\[
s = \text{first input character}
\]
\[
\text{while ( not EOF )} \{ \\
    c = \text{next input character;}
    \text{if } s+c \text{ exists in the dictionary} \\
    \hspace{1em} s = s+c;
    \text{else} \{ \\
        \hspace{1em} \text{output the code for } s; \\
        \hspace{1em} \text{add } s+c \text{ to the dictionary;}
        \hspace{1em} s = c;
    \}
\}
\]
\[
\text{output code for } s;
\]

19-symbol input has been reduced to 7-symbol + 5-code output!
LZW Decompression

Example: Compressed string is 
"^WED<256>E<260><261><257>B<260>T".

<table>
<thead>
<tr>
<th>s</th>
<th>k</th>
<th>Output</th>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td>^</td>
<td>^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>W</td>
<td>W</td>
<td>256</td>
<td>^W</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>E</td>
<td>257</td>
<td>WE</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>D</td>
<td>258</td>
<td>ED</td>
</tr>
<tr>
<td>D</td>
<td>&lt;256&gt;</td>
<td>^W</td>
<td>259</td>
<td>D^</td>
</tr>
<tr>
<td>&lt;256&gt;</td>
<td>E</td>
<td>E</td>
<td>260</td>
<td>^WE</td>
</tr>
<tr>
<td>E</td>
<td>&lt;260&gt;</td>
<td>^WE</td>
<td>261</td>
<td>E^</td>
</tr>
<tr>
<td>&lt;260&gt;</td>
<td>&lt;261&gt;</td>
<td>E^</td>
<td>262</td>
<td>^WEE</td>
</tr>
<tr>
<td>&lt;261&gt;</td>
<td>&lt;257&gt;</td>
<td>WE</td>
<td>263</td>
<td>E^W</td>
</tr>
<tr>
<td>&lt;257&gt;</td>
<td>B</td>
<td>B</td>
<td>264</td>
<td>WEB</td>
</tr>
<tr>
<td>B</td>
<td>&lt;260&gt;</td>
<td>^WE</td>
<td>265</td>
<td>B^</td>
</tr>
<tr>
<td>260</td>
<td>T</td>
<td>T</td>
<td>266</td>
<td>^WET</td>
</tr>
</tbody>
</table>

Decompressor builds its own dictionary, so only the code needs to be sent!
LZW Algorithm Details (I)

Example: Input string is “ABABBABCABBABBAX”.

<table>
<thead>
<tr>
<th>s</th>
<th>c</th>
<th>Output</th>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>256</td>
<td>AB</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>B</td>
<td>257</td>
<td>BA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>B</td>
<td>256</td>
<td>258</td>
<td>ABB</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>B</td>
<td>257</td>
<td>259</td>
<td>BAB</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>B</td>
<td>260</td>
<td>BC</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
<td>261</td>
<td>CA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABB</td>
<td>A</td>
<td>258</td>
<td>262</td>
<td>ABBA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABB</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABBA</td>
<td>X</td>
<td>262</td>
<td>263</td>
<td>ABBA(X)</td>
</tr>
<tr>
<td>X</td>
<td>...</td>
<td>X</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Compressed string is “AB<256><257>BC<258><262>X…”.

<table>
<thead>
<tr>
<th>s</th>
<th>k</th>
<th>Output</th>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>B</td>
<td>256</td>
<td>AB</td>
</tr>
<tr>
<td>B</td>
<td>&lt;256&gt;</td>
<td>AB</td>
<td>257</td>
<td>BA</td>
</tr>
<tr>
<td>&lt;256&gt;</td>
<td>&lt;257&gt;</td>
<td>BA</td>
<td>258</td>
<td>ABB</td>
</tr>
<tr>
<td>&lt;257&gt;</td>
<td>B</td>
<td>B</td>
<td>259</td>
<td>BAB</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>C</td>
<td>260</td>
<td>BC</td>
</tr>
<tr>
<td>C</td>
<td>&lt;258&gt;</td>
<td>ABB</td>
<td>261</td>
<td>CA</td>
</tr>
<tr>
<td>&lt;258&gt;</td>
<td>&lt;262&gt;</td>
<td>???</td>
<td>262</td>
<td>???</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<262> = ABBA = A + BB + A

Whenever encoder encounters Character + String + Character…, it creates a new code and use it right away before the decoder has a chance to create it!
What if Input string is “ABABBABCABBXBBAX”.

Compressed string is “AB<256><257>BC<258>XB<257>X…”.

<table>
<thead>
<tr>
<th>s</th>
<th>c</th>
<th>Output</th>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>256</td>
<td>AB</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>B</td>
<td>257</td>
<td>BA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>B</td>
<td>256</td>
<td>258</td>
<td>ABB</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>B</td>
<td>257</td>
<td>259</td>
<td>BAB</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>B</td>
<td>260</td>
<td>BC</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
<td>261</td>
<td>CA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABB</td>
<td>X</td>
<td>258</td>
<td>262</td>
<td>ABBX</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
<td>X</td>
<td>263</td>
<td>XB</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>264</td>
<td>BB</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>X</td>
<td>257</td>
<td>265</td>
<td>BAX</td>
</tr>
<tr>
<td>X</td>
<td>...</td>
<td>X</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>k</th>
<th>Output</th>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>B</td>
<td>256</td>
<td>AB</td>
</tr>
<tr>
<td>B</td>
<td>&lt;256&gt;</td>
<td>AB</td>
<td>257</td>
<td>BA</td>
</tr>
<tr>
<td>&lt;256&gt;</td>
<td>&lt;257&gt;</td>
<td>BA</td>
<td>258</td>
<td>ABB</td>
</tr>
<tr>
<td>&lt;257&gt;</td>
<td>B</td>
<td>B</td>
<td>259</td>
<td>BAB</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>C</td>
<td>260</td>
<td>BC</td>
</tr>
<tr>
<td>C</td>
<td>&lt;258&gt;</td>
<td>ABB</td>
<td>261</td>
<td>CA</td>
</tr>
<tr>
<td>&lt;258&gt;</td>
<td>X</td>
<td>X</td>
<td>262</td>
<td>ABBX</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
<td>B</td>
<td>263</td>
<td>XB</td>
</tr>
<tr>
<td>B</td>
<td>&lt;257&gt;</td>
<td>BA</td>
<td>264</td>
<td>BB</td>
</tr>
<tr>
<td>&lt;257&gt;</td>
<td>X</td>
<td>X</td>
<td>265</td>
<td>BAX</td>
</tr>
</tbody>
</table>

=>

No problem!
s = NIL
While (not EOF) {
    k = next input code;
    entry = dictionary entry for k;
    /* Exception Handler */
    if (entry == NULL)
        entry = s + s[0];
    output entry;
    if (s != NIL)
        add s + entry[0] to dictionary;
    s = entry;
}
Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding

Chapter 2: Compression Basics

64
Lossy Encoding

- Coding is lossy
  - Consider sequences of symbols $S_1, S_2, S_3$, etc., where values are not zeros but do not vary very much.
    - We calculate difference from previous value => $S_1, S_2-S_1, S_3-S_2$, etc.

- e.g., Still image
  - Calculate difference between nearby pixels or pixel groups.
  - Edges characterized by large values, areas with similar luminance and chrominance are characterized by small values.
  - Zeros can be compressed by run-length encoding and nearby pixels with large values can be encoded as differences.