Compression

- **Lossless**
  - Data received = data sent
  - Used for executables, text files, numeric data

- **Lossy**
  - Data received does not != data sent
  - Used for images, video, audio
Simple lossless algorithms

- **Run Length Encoding (RLE)**
  - Example: AAABBCDDDDD encoding as 3A2B1C4D
  - Good for scanned text (8-to-1 compression ratio)
  - Used by fax machines
  - Can increase size for data with variation (e.g., some images)

- **Differential Pulse Code Modulation (DPCM)**
  - Example: AAABBCDDDDD encoding as A0001123333
  - Change reference symbol if delta becomes too large
  - Works better than RLE for many digital images (1.5-to-1)
Huffman coding

- Consider data as sequence of symbols
- Suppose you know the frequency of each symbol
  - Assign symbols weight proportional to frequency
- Use shorter code for more frequent characters
- Example: NYU centrex phones
  - Consider each phone number to be a symbol
  - Most phone numbers dialed within NYU are 212-99x-xxxx
  - So compress phone number, by just dialing x-xxxx
  - But now must dial 9 for external phone numbers
    (longer code, but less frequently used)
- Usually, have smaller symbol alphabet
  - E.g., symbols are bytes, codes are in bits
Huffman trees

- Can view Huffman codes as paths in a tree
  - Symbols with heavier weight are higher in tree

- Example: \( B = 16, \ A = 8, \ C = 4, \ D = 4 \)

\[
\begin{array}{c|c}
\text{Symbol} & \text{Code} \\
\hline
A & 00 \\
B & 1 \\
C & 011 \\
D & 010 \\
\end{array}
\]
Building Huffman trees

- Given weights, can build a suitable Huffman tree
  - Take two symbols with least weight
  - Assign suffix “0” to one, suffix “1” to the other
  - Add new symbol with combined weight to figure out prefix

- Example: \( B = 16, \ A = 8, \ C = 4, \ D = 4 \)
  - Let suffix of \( D \) be “0”, of \( C \) be “1”.
  - New weights: \( B = 16, \ A = 8, \ C\text{-and-}D = 8 \)
  - Next two are \( A \) and \( C\text{-and-}D = 8 \); give \( A \) suffix “0”
  - Now give \( A\text{-and-}B\text{-and-}C \) suffix “0”, \( B \) suffix “1”
  - Result is tree is on previous slide
Transmitting Huffman trees

- Decoder needs Huffman tree

- One approach: Hard-code tree in encoder/decoder
  - E.g., could come up with suitable weights for English text

- Another approach: Include tree w. message
  - Deterministically agree on one tree per set of weights:
    - All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent;
    - Shorter codes lexicographically precede longer codes
  - Now just need to transmit length of each symbol’s code (which can be further compressed)
Arithmetic coding

- Like Huffman, assume symbol weights known

- Assign each symbol a range in \([0, 1)\)
  - Example: Say weights are \(B = 16\), \(A = 8\), \(C = 4\), \(D = 4\)
    \(B = [0, 0.5), A = [0.5, 0.75), C = [0.75, 0.875), D = [0.875, 1)\)
  - Now for each symbol, subdivide the range
  - For String “BBA”, ranges will subdivide to get:
    \( [0, 0.5), [0, 0.25), [0.125, 0.1875)\)
  - Transmit compressed string as rational number in range

- Potentially better than Huffman codes
  - Can better accommodate non-power-of-two weights
  - Less often used (in part because of patents)
Dictionary-Based Methods

• Build dictionary of common terms
  - Variable length strings

• Transmit index into dictionary for each term

• Lempel-Ziv (LZ) is the best-known example

• Commonly achieve 2-to-1 ratio on text

• Variation of LZ used to compress GIF images
  - First reduce 24-bit color to 8-bit color
  - Treat common sequence of pixels as terms in dictionary
  - Not uncommon to achieve 10-to-1 compression (x3)
Gzip compression (LZ77)

- Big idea: Include backreferences to past data
- Consider string “Blah blah blah blah blah!”
- Notice repeat: “Blah blah blah blah!”
- Encode “Blah blah b” as “Blah b[D = 5, L = 5]”
  - Go back $D$ characters, copy string of $L$ bytes
- But why stop there. Can say:
  “Blah b[D = 5, L = 18]!”
  - Length $L$ can extend into copied string!

\(^{a}\)example from gzip site
Gzip details

- Compressed file is composed of blocks of data
- First bit is BFINAL – set if final block
- Next two bits are BTYPE
  - 00 – no compression (if compression makes things worse)
  - 01 – compressed with fixed Huffman codes
  - 10 – compressed with dynamic Huffman codes
  - 11 – reserved (error)

- No compression blocks (up to $2^{16} - 1$ Bytes):
  
  |  |  |  |  |  |
  |---|---|---|---|
  +---+---+---+---+================================+
  |LEN | NLEN |... LEN bytes of literal data...|
  +---+---+---+---+================================+

  (NLEN is one’s compliment of LEN)
Huffman coding of gzip

- **Use two Huffman trees**
  - One tree for symbols & length ($L$) values
  - Second tree for back distances ($D$)
  - Note $L$ value always followed by $D$, so no ambiguity

- **Symbol/length alphabet has 285 symbols**
  - 0–255 → are data bytes
  - 256 → end of block
  - 257–285 → length of backreference

- $L \in \{3, \ldots, 258\}$. **How to encode w. 29 values?**
**Encoding $L$**

- **Add extra bits after some symbols**
  - E.g., Symbol 264 means $L = 10$
  - 265 means $L \in \{11, 12\}$, extra bit follows

<table>
<thead>
<tr>
<th>Extra Code</th>
<th>Bits</th>
<th>Length(s)</th>
<th>Extra Code</th>
<th>Bits</th>
<th>Lengths</th>
<th>Extra Code</th>
<th>Bits</th>
<th>Lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>0</td>
<td>3</td>
<td>267</td>
<td>1</td>
<td>15,16</td>
<td>277</td>
<td>4</td>
<td>67–82</td>
</tr>
<tr>
<td>258</td>
<td>0</td>
<td>4</td>
<td>268</td>
<td>1</td>
<td>17,18</td>
<td>278</td>
<td>4</td>
<td>83–98</td>
</tr>
<tr>
<td>259</td>
<td>0</td>
<td>5</td>
<td>269</td>
<td>2</td>
<td>19–22</td>
<td>279</td>
<td>4</td>
<td>99–114</td>
</tr>
<tr>
<td>260</td>
<td>0</td>
<td>6</td>
<td>270</td>
<td>2</td>
<td>23–26</td>
<td>280</td>
<td>4</td>
<td>115–130</td>
</tr>
<tr>
<td>261</td>
<td>0</td>
<td>7</td>
<td>271</td>
<td>2</td>
<td>27–30</td>
<td>281</td>
<td>5</td>
<td>131–162</td>
</tr>
<tr>
<td>262</td>
<td>0</td>
<td>8</td>
<td>272</td>
<td>2</td>
<td>31–34</td>
<td>282</td>
<td>5</td>
<td>163–194</td>
</tr>
<tr>
<td>263</td>
<td>0</td>
<td>9</td>
<td>273</td>
<td>3</td>
<td>35–42</td>
<td>283</td>
<td>5</td>
<td>195–226</td>
</tr>
<tr>
<td>264</td>
<td>0</td>
<td>10</td>
<td>274</td>
<td>3</td>
<td>43–50</td>
<td>284</td>
<td>5</td>
<td>227–257</td>
</tr>
<tr>
<td>265</td>
<td>1</td>
<td>11,12</td>
<td>275</td>
<td>3</td>
<td>51–58</td>
<td>285</td>
<td>0</td>
<td>258</td>
</tr>
<tr>
<td>266</td>
<td>1</td>
<td>13,14</td>
<td>276</td>
<td>3</td>
<td>59–66</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Encoding \( D \)

- Similar scheme for \( D \), which has 30 symbols

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td>33-48</td>
<td>20</td>
<td>9</td>
<td>1025-1536</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>11</td>
<td>4</td>
<td>49-64</td>
<td>21</td>
<td>9</td>
<td>1537-2048</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>12</td>
<td>5</td>
<td>65-96</td>
<td>22</td>
<td>10</td>
<td>2049-3072</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>13</td>
<td>5</td>
<td>97-128</td>
<td>23</td>
<td>10</td>
<td>3073-4096</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5,6</td>
<td>14</td>
<td>6</td>
<td>129-192</td>
<td>24</td>
<td>11</td>
<td>4097-6144</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>7,8</td>
<td>15</td>
<td>6</td>
<td>193-256</td>
<td>25</td>
<td>11</td>
<td>6145-8192</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>9-12</td>
<td>16</td>
<td>7</td>
<td>257-384</td>
<td>26</td>
<td>12</td>
<td>8193-12288</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>13-16</td>
<td>17</td>
<td>7</td>
<td>385-512</td>
<td>27</td>
<td>12</td>
<td>12289-16384</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>17-24</td>
<td>18</td>
<td>8</td>
<td>513-768</td>
<td>28</td>
<td>13</td>
<td>16385-24576</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>25-32</td>
<td>19</td>
<td>8</td>
<td>769-1024</td>
<td>29</td>
<td>13</td>
<td>24577-32768</td>
</tr>
</tbody>
</table>
Gzip’s fixed Huffman codes

- \( D \) values just encoded as 5 bits
  - Values 30 and 31 never used

- Symbol/\( L \) values use following code:

<table>
<thead>
<tr>
<th>Lit Value</th>
<th>Bits</th>
<th>Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 143</td>
<td>8</td>
<td>00110000 - 10111111</td>
</tr>
<tr>
<td>144 - 255</td>
<td>9</td>
<td>110010000 - 111111111</td>
</tr>
<tr>
<td>256 - 279</td>
<td>7</td>
<td>00000000 - 0010111</td>
</tr>
<tr>
<td>280 - 287</td>
<td>8</td>
<td>11000000 - 11000111</td>
</tr>
</tbody>
</table>

- Why might these be a good choice?
Dynamic Huffman codes

- **Must first transmit two Huffman trees (symb/L & D)**
  - Recal means xmitting code length for each symbol
  - Use code length 0 for unused symbols
  - Include length of table, truncate after last non-0

- **Use Huffman + RLE on the Huffman trees! Alphabet:**

<table>
<thead>
<tr>
<th>Extra Code</th>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 15</td>
<td>0</td>
<td>Represent code lengths of 0 - 15</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>Copy the previous code length 3 - 6 times</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>Copy previous code length 3 - 10 times</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>Copy previous code length 11 - 138 times</td>
</tr>
</tbody>
</table>

  - Must transmit 3-bit code length for each of 19 symbols (0–18)
  - Transmit lengths in order that often helps truncation:
    16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
Lossy image compression

- JPEG: Joint Photographic Expert Group (ISO/ITU)
- Lossy still-image compression
- Three phase process
  - Process in 8x8 block chunks (macro-block)
  - DCT: transforms signal from spatial domain into and equivalent signal in the frequency domain (loss-less)
  - Apply a quantization to the results (lossy)
  - RLE (for 0s) + Huffman encoding (loss-less)
  - For color: each pixel is three values (YUV)
    \[ Y = \text{luminescence}, \quad U = c_1(B - Y), \quad V = c_2(R - Y) \]
• **Quantization Table**

\[
\begin{pmatrix}
3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 \\
5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\
7 & 9 & 11 & 13 & 15 & 17 & 19 & 21 \\
9 & 11 & 13 & 15 & 17 & 19 & 21 & 23 \\
11 & 13 & 15 & 17 & 19 & 21 & 23 & 25 \\
13 & 15 & 17 & 19 & 21 & 23 & 25 & 27 \\
15 & 17 & 19 & 21 & 23 & 25 & 27 & 29 \\
17 & 19 & 21 & 23 & 25 & 27 & 29 & 31
\end{pmatrix}
\]

• **Encoding Pattern**

![Encoding Pattern Diagram]
MPEG

- Motion Picture Expert Group
- Lossy compression of video

First approximation: JPEG on each frame
- Y value encoded with 16 × 16 “macroblock”
- U, V channels downsampled into 8 × 8 macroblocks

Also remove inter-frame redundancy
- E.g., when large objects move across image
MPEG (cont)

- **Frame types**
  - I frames: intrapicture
  - P frames: predicted picture
  - B frames: bidirectional predicted picture

- **Example sequence transmitted as I P B B I B B**
MPEG (cont)

- **B and P frames**
  - Coordinate for the macroblock in the frame
  - Motion vector relative to previous reference frame (B, P)
  - Motion vector relative to subsequent reference frame (B)
  - Delta for each pixel in the macro block

- **Effectiveness**
  - Typically 90-to-1
  - As high as 150-to-1
  - 30-to-1 for I frames
  - P and B frames get another 3 to 5×
Transmitting MPEG

- Adapt the encoding
  - Resolution
  - Frame rate
  - Quantization table
  - Specified in GOP (group of pictures) header

- Packetization (best to lose whole frames)

- Dealing with loss

- GOP-induced latency
MP3

• CD Quality
  - 44.1 kHz sampling rate
  - $2 \times 44.1 \times 1000 \times 16 = 1.41$ Mbps
  - Actually requires more for synchronization & error correction

• Strategy
  - Split into some number of frequency bands
  - Divide each subband into a sequence of blocks
  - Encode each block using DCT + Quantization + Huffman
  - Trick: how many bits assigned to each subband