Error-free compression

- Useful in application where no loss of information is tolerable. This maybe due to accuracy requirements, legal requirements, or less than perfect quality of original image.

- Compression can be achieved by removing coding and/or interpixel redundancy.

- Typical compression ratios achievable by lossless techniques is from 2 to 10.

Variable Length Coding

- This is used to reduce coding redundancy.

- Coding redundancy is present in any image with a non-uniform histogram (i.e. when all the graylevels are not equally likely).

- Given an image with, say 256 graylevels, \( \{a_0, a_1, \ldots, a_{255}\} = \{0,1,\ldots,255\} \). This is our set of source symbols.

- For each graylevel \( a_k \), we need its probability \( p(a_k) \) in the image. This may be obtained from the image histogram: \( p(a_k) = n_k/n \), \( n_k \) = # pixels with value \( a_k \), \( n \) = total # pixels.

- To each graylevel \( a_k \), we need to assign a codeword (a binary string). Suppose \( l_k \) is the length of codeword (= # bits required to represent \( a_k \)) for symbol \( a_k \).

- Total number of bits required to represent the image is

\[
\sum_{k=0}^{N-1} l_k n_k = n \sum_{k=0}^{N-1} l_k \left( \frac{n_k}{n} \right) = n \sum_{k=0}^{N-1} l_k p(a_k) = nL_{\text{avg}}
\]
• Naturally, we need an encoding scheme with $L_{\text{avg}}$ as small as possible. From Shannon’s theorem, we know that $L_{\text{avg}} \geq H(z)$.

• As mentioned earlier, the Huffman procedure specifies a code with

\[
H(z) \leq L_{\text{avg}} < H(z) + 1
\]

**Huffman Code**

• The algorithm is best illustrated by means of an example.

• Given a source which generates one of six possible symbols $A = \{a_1, a_2, \ldots, a_6\}$ with corresponding probabilities $\{0.1, 0.4, 0.06, 0.1, 0.04, 0.3\}$.

• Arrange the symbols in descending order of their probability of occurrence.

• Successively reduce the number of source symbols by replacing the two symbols having least probability, with a “compound symbol.” This way, the number of source symbols is reduced by one at each stage.

• The compound symbol is placed at an appropriate location in the next stage, so that the probabilities are again in descending order. Break ties using any arbitrary but consistent rule.

• Code each reduced source starting with the smallest source and working backwards.
Source Reduction:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>$a_6$</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5$</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code Assignment:

<table>
<thead>
<tr>
<th>Symb.</th>
<th>Prob.</th>
<th>Code</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>0.4</td>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>$a_6$</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
<td>00</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.1</td>
<td>011</td>
<td>0.1</td>
<td>011</td>
<td>0.2</td>
<td>010</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
<td>011</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.06</td>
<td>01010</td>
<td>0.1</td>
<td>0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5$</td>
<td>0.04</td>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
L_{\text{avg}} = \sum_i p_i l_i = 0.4(1) + 0.3(2) + 0.1(3) + 0.1(4) + 0.06(5) + 0.04(5) \\
= 2.2 \text{ bits/symbol}
\]

\[
H(z) = -\sum_i p_i \log(p_i) = \left[ 0.4(\log(0.4)) + 0.3(\log(0.3)) + 0.1(\log(0.1)) + 0.1(\log(0.1)) + 0.06(\log(0.06)) + 0.04(\log(0.04)) \right] \\
= 2.14 \text{ bits/symbol}
\]
The resulting code is called a Huffman code. It has some interesting properties:

- The source symbols can be encoded (and decoded) one at a time.
- It is called a **block code** because each source symbol is mapped into a fixed sequence of code symbols.
- It is **instantaneous** because each codeword in a string of code symbols can be decoded without referencing succeeding symbols.
- It is **uniquely decodable** because any string of code symbols can be decoded in only one way.

**Example:** Given the encoded string, 01010011100, it can be decoded as follows:

\[
01010011100 \rightarrow 01010011100 \rightarrow a_30111100 \rightarrow a_3a_11100 \\
\rightarrow a_3a_1a_2100 \rightarrow a_3a_1a_2a_200 \rightarrow a_3a_1a_2a_2a_6
\]

The Huffman code is optimal (in terms of average codeword length) for a given set of symbols and probabilities, subject to the constraint that the symbols be coded one at a time.

**Disadvantage:**

- For a source with \( J \) symbols, we need \( J - 2 \) source reductions. This can be computationally intensive for large \( J \) (ex. \( J = 256 \) for an image with 256 gray levels).
Suboptimal approaches

Truncated Huffman coding:

- This is generated by encoding only the $\psi$ most probable symbols, for some integer $\psi < J$.

- A prefix code followed by a suitable fixed-length code is used to encode the rest of the symbols.

- Other examples of suboptimal codes are B-code, binary shift code, and Huffman shift code.

<table>
<thead>
<tr>
<th>Table 6.5 Variable-Length Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Symbol</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Block 1</td>
</tr>
<tr>
<td>$a_{1}$</td>
</tr>
<tr>
<td>$a_{2}$</td>
</tr>
<tr>
<td>$a_{3}$</td>
</tr>
<tr>
<td>$a_{4}$</td>
</tr>
<tr>
<td>$a_{5}$</td>
</tr>
<tr>
<td>$a_{6}$</td>
</tr>
<tr>
<td>$a_{7}$</td>
</tr>
<tr>
<td>Block 2</td>
</tr>
<tr>
<td>$a_{8}$</td>
</tr>
<tr>
<td>$a_{9}$</td>
</tr>
<tr>
<td>$a_{10}$</td>
</tr>
<tr>
<td>$a_{11}$</td>
</tr>
<tr>
<td>$a_{12}$</td>
</tr>
<tr>
<td>$a_{13}$</td>
</tr>
<tr>
<td>$a_{14}$</td>
</tr>
<tr>
<td>Block 3</td>
</tr>
<tr>
<td>$a_{15}$</td>
</tr>
<tr>
<td>$a_{16}$</td>
</tr>
<tr>
<td>$a_{17}$</td>
</tr>
<tr>
<td>$a_{18}$</td>
</tr>
<tr>
<td>$a_{19}$</td>
</tr>
<tr>
<td>$a_{20}$</td>
</tr>
<tr>
<td>$a_{21}$</td>
</tr>
</tbody>
</table>

Entropy 4.0

Average Length 5.0 4.05 4.24 4.65 4.59 4.13
Bit-plane Coding

- A grayscale image is decomposed into a series of binary images and each binary image is compressed by some binary compression method.

- This removes coding and interpixel redundancy.

**Bit-plane decomposition:**

- Given a grayscale image with $2^m$ graylevels, each grayvalue can be represented by $m$-bits, say $(a_{m-1}, a_{m-2}, \ldots, a_1, a_0)$.

- The grayvalue $r$ represented by $(a_{m-1}, a_{m-2}, \ldots, a_1, a_0)$ is given by the base 2 polynomial

$$r = a_{m-1} 2^{m-1} + a_{m-2} 2^{m-2} + \cdots + a_1 2^1 + a_0 2^0$$

- This bit representation can be used to decompose the grayscale image into $m$ binary images (bit-planes).

- Alternatively, one can use the $m$-bit Gray code $(g_{m-1}, g_{m-2}, \ldots, g_1, g_0)$ to represent a given grayvalue.

- The Gray code $(g_{m-1}, g_{m-2}, \ldots, g_1, g_0)$ can be obtained from $(a_{m-1}, a_{m-2}, \ldots, a_1, a_0)$ by the following relationship:

$$g_{m-1} = a_{m-1}, \quad \text{and for } 0 \leq i \leq m-2, \quad g_i = a_i \oplus a_{i+1}$$

where $\oplus$ denotes exclusive OR of bits.

- The Gray code of successive graylevels differ at only one position.

  127 $\rightarrow$ 01111111  (binary representation)  01000000  (Gray code)
  128 $\rightarrow$ 10000000  (binary representation)  11000000  (Gray code)
• The resulting binary images are then compressed (error-free).

• We will study a popular encoding scheme called run-length encoding (RLC).

**Runlength encoding**

• Each row of a bit plane (or binary image) is represented by a sequence of lengths (integers) that denote the successive runs of 0 and 1 pixels.

<table>
<thead>
<tr>
<th>1 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

• Two approaches:
  - Start position and lengths of runs of 1s for each row is used:
    
    \[(1,3)(7,2)(12,4)(17,2)(20,3)\]
    \[(5,13)(19,4)\]
    \[(1,3)(17,6)\]
  - Only lengths of runs, starting with the length of 1 run is used:
    
    \[3,3,2,3,4,1,2,1,3\]
    \[0,4,13,1,4\]
    \[3,13,6\]
• This technique is very effective in encoding binary images with large contiguous black and white regions, which would give rise to a small number of large runs of 1s and 0s.

• The run-lengths can in turn be encoded using a variable length code (ex. Huffman code), for further compression.

• Let $a_k$ be the fraction of runs of 0s with length $k$. Naturally, $(a_1, a_2, \ldots, a_M)$, would represent a vector of probabilities (the probability of a run of 0s being of length $k$).

• Let $H_0 = \sum_{i=1}^{M} a_i \log(a_i)$ be the entropy associated with $(a_1, a_2, \ldots, a_M)$ and $L_0 = \sum_{i=1}^{M} i a_i$ be the average length of runs of 0s.

• Let $b_k$ be the fraction of runs of 1s with length $k$. Naturally, $(b_1, b_2, \ldots, b_M)$, would represent a vector of probabilities (the probability of a run of 1s being of length $k$).

• Let $H_1$ be the entropy associated with $(b_1, b_2, \ldots, b_M)$ and $L_1 = \sum_{i=1}^{M} i b_i$ be the average length of runs of 1s.

• The approximate runlength entropy of the image is

$$H_{RL} = \frac{(H_0 + H_1) \text{bits/run}}{(L_0 + L_1) \text{symbols/run}}$$

• $H_{RL}$ provides an estimate of the average number of bits per pixel required to code the run lengths in a binary image, using a variable-length code.

• The concept of run-length can be extended to a variety of 2-D coding procedures.
Runlength Example

Runlength encoding of row # 152:
0, 17, 4, 13, 5, 3, 5, 2, 4, 3, 5, 4, 3, 2, 4, 5, 4, 4, 6, 3, 6, 2, 24, 4, 5, 12, 15, 3, 6, 2, 4, 3, 6, 2, 4, 3, 3, 4, 25, 4, 6, 10, 7 (43 runs)
Distribution of lengths of runs of 0 in image

\[ L_0 = 3.2 \ \text{pixels/run} \]
\[ H_0 = 2.2 \ \text{bits/run} \]

Distribution of lengths of runs of 1 in image

\[ L_1 = 20.3 \ \text{pixels/run} \]
\[ H_1 = 4.4 \ \text{bits/run} \]

\[ H_{RL} = \frac{H_0 + H_1}{L_0 + L_1} = 0.28 \ \text{bits/pixel} \]
Lossless Predictive Coding

- Does not require decomposition of grayscale image into bitplanes.
- Eliminate interpixel redundancy by extracting and coding only the *new information* in each pixel.
- New information in a pixel is the difference between its actual and predicted (based on “previous pixel values”) values.

\[ e_n = f_n - \hat{f}_n \text{ or } f_n = e_n + \hat{f}_n \]

\[ \hat{f}_n = \text{round} \left( \sum_{i=1}^{m} \alpha_i f_{n-i} \right) \]
Example: 1-D first order linear predictor

\[ \hat{f}(m,n) = \text{round} \left[ \alpha f(m, n - 1) \right] \]  

(previous pixel predictor)

- In each row, a pixel value is predicted based on the value of the pixel to its left.
- The resulting prediction error \( e(m, n) = f(m, n) - \hat{f}(m, n) \) is encoded.
- The first element of each row (i.e., first column of image) is also encoded (using, for example, a different Huffman code).
- Decoder reconstructs \( e(m, n) \) based on the codewords and obtains the original pixel values using \( f(m, n) = e(m, n) + \hat{f}(m, n) \)
• Possible data reduction: 8 bits/pixel w/o compression, 6.9 bits/pixel using only entropy coding, 4.4 bits/pixel using predictive coding.