3. LOSSLESS COMPRESSION METHODS

Question: What is the minimum number of bits required to recover perfectly (in a noiseless/lossless fashion) a discrete random vector (samples, group of samples, pixels, group of pixels)?

In the Shannon information processing context, this is generally re-phrased as:

1. What is the minimum rate for zero distortion?
2. How it is achieved?

First question is answered in 1949 by the Shannon Source Coding Theorem of Chapter 2. (It is possible to code and transmit discrete information with arbitrarily small or even no error if the rate is not to exceed the entropy.) Huffmann gave an optimal solution to the second one with an elegant code design.

Consider a fixed-rate or a fixed length code:

Input Letter	Codeword
a_0	00
a_1	01
a_2	10
a_3	11

Can we **uniquely** decode individual codewords and sequences if we know where the blocks start?

Yes. If we know where blocks start.

Example 3.1: "0000011011" decodes as: a_0, a_0, a_1, a_2, a_3 "a0a0$a_1a_2a_3$".

- Where is the compression? There are four letters in the alphabet and four codewords of length two. Therefore, no compression is possible.
- A real world example: ASCII (American Standard Code for Information Interchange): It assigns binary numbers to all letters, numbers, punctuation for use in computers and digital communication and storage. All letters have same number of binary symbols (7).

Example 3.2:

a : 110001
b : 110010
c : 110011
d : 110100
e : 110101
...

If there are M letters in the alphabet, we need, at least, R bits,

$$2^R > M$$

- Generally, a fixed rate code gives no lossless compression unless original representation was inefficient.
- **Conclusion:** To get lossless compression need a variable length code.

Example 3.3: Variable rate code:

Input Letter	Codeword
a_0	0
a_1	10
a_2	101
a_3	0101

This code cannot always be decoded in a noiseless fashion when the code is applied to a sequence of inputs. Furthermore, the ambiguity can never be resolved regardless of bits received in the future. For instance, **0101** can be decoded as a_0 followed by $\mathbf{a_2}$ or $\mathbf{a_3}$!

Solution: Uniquely decodable codes:

Input Letter	Codeword
a_0	0
a_1	10
a_2	110
a_3	111

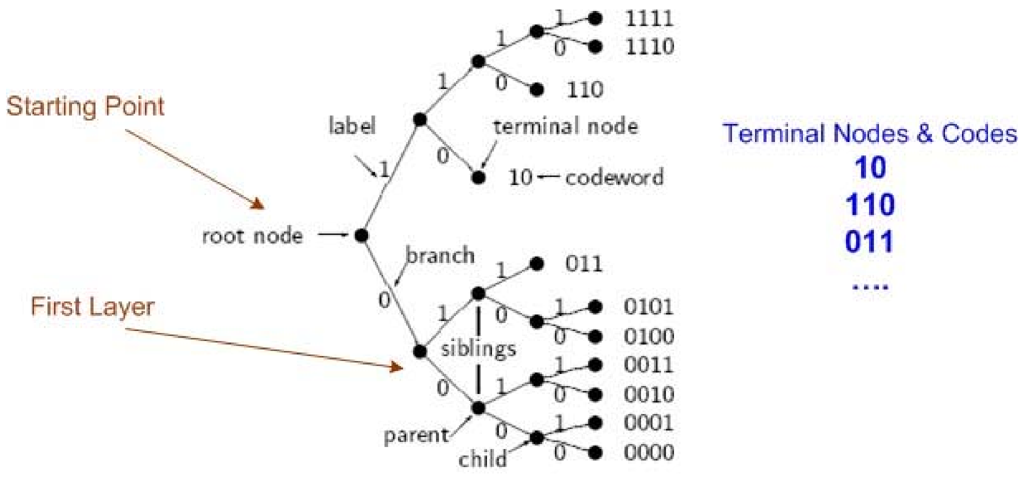
If we know where start, all valid encoded sequences of codewords have unique decoding.

Prefix-free and Tree-structured Codes

- **Prefix condition:** No codeword is a prefix of another codeword. Such a code is said to be ***prefix-free.***
- For prefix-free codes uniquely decodability is guaranteed.
- Essentially, no loss of generality.
- Average length of these codes are defined by:

$$\bar{l}(\alpha) = Average\{l(\alpha(X))\} = \sum p(a).l(\alpha(a)) \quad bits/symbol \tag{3.1}$$

- For any uniquely decodable code, there is a tree-structured code with the same collection of codeword lengths and the same average length.
- Binary prefix-free codes can be depicted as a binary tree:



Tree Formation: We split the root of the tree into two branches to generate left and right siblings (children) and associate "1" or "0" depending upon which branch we go. Whenever a child does not produce siblings, we prune the tree at that point and this node is called a "**terminal node**" and the branch labels left to right form the **codeword**.

Theorem: Optimum binary prefix-free codes have following properties:

(i) *If the codeword for input symbol a has length $l(a)$, then $p(a) > p(b)$ implies that $l(a) \le l(b)$; that is, more probable input symbols have shorter (at least, not longer) codewords.*

(ii) *The two least probable input symbols have codewords which are equal in length and differ only in the final symbol.*

This theorem provides an iterative design technique for **Huffman Codes**. Let us now re-visit Example 2.1 and design a Huffman code for this case.

Example 2.1 (re-visited): Consider the following 6 source symbols to be used as messages: S={A,B,C,D,E,F} with occurrence probabilities {0.25, 0.20, 0.16, 0.15, 0.13, 0.11}.

Note that sum of probabilities is 1.0 since there are no other messages and possibilities left. One way to represent this symbol set is to assign $\overline{L} = 3 - bit$ long codewords to each symbol as it is shown in Table 2.1.

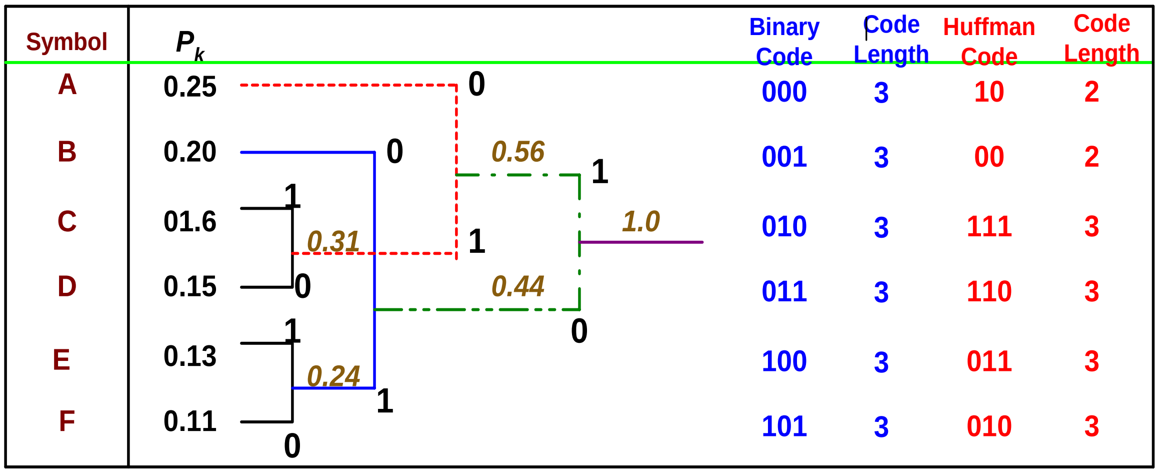
- 8 possible ways to arrange 3 bits: 000, 001,…, 111.
- Chose any 6 out of these combinations.
- Compute the entropy and average length of this coding procedure.

$$H(S) = -\sum_{k=1}^{6} p_k . \log_2(p_k) = 2.5309 \quad bit/symbol$$

- It is easy to see that the difference $\overline{L} - H(S) = 3.0 - 2.5309 = 0.4691\, bits/symbol$ is almost ½ bits per symbol away from the theoretical bound *H(S)*.
- Another scheme is to use a variable-length bit assignment according to some nice rule! As also shown un the table.

Huffman Code Design Stage:

- Organize the symbols in a descending order (probability-wise), if any two symbol has the same probability assume a tie-breaker rule.
- Consider the least probable two entries (F, E): Assign “1” to more probable, “0” to the other and combine these two branches and the new branch will have the sum of probabilities: (0.11+0.13=0.24).
- Now consider the least probable two entries including the combined one and repeat the assignment. If you have two entries with identical probabilities then use the tie-breaker rule (we do not face this issue in this example!).
- Continue the above until you combine the last two entries.
- Codewords are read from “RIGHT-to-LEFT” as tabulated.
- Final column stands for the length of each codeword.
- We compute the average length of the Huffman code and compare it against the entropy of the source to see how good the design is.



Symbol	P_k	Binary Code	Code Length	Huffman Code	Code Length
A	0.25	000	3	10	2
B	0.20	001	3	00	2
C	01.6	010	3	111	3
D	0.15	011	3	110	3
E	0.13	100	3	011	3
F	0.11	101	3	010	3

The average length of this code is:

$$\bar{L}_{Huffman} = 0.25x2 + 0.20x2 + 0.16x3 + 0.15x3 + 0.13x3 + 0.11x3 = 2.55 \quad bits/symbol$$

The difference in this case is lowered to:

$$\bar{L} - H(S) = 2.55 - 2.5309 = 0.0191\, bits/symbol\,,$$

which is almost perfect.

It can be shown that the source ***S***, has an average code length bound by:

$$H(S) \le \bar{L}_{Huffman} \le H(S) + 1 \tag{3.2}$$

It is known in the information theory community that we have a uniquely decodable code ***C*** with with K codewords of length $\{l_i;\ for\ i = 1,2,\cdots,K\}$, the Kraft-McMillan inequality says:

$$\sum_{i=1}^{K} 2^{-l_i} \le 1 \tag{3.3}$$

Example 3.4: Extended Huffman Code. Consider a source with alphabet $A = \{a_1, a_2, a_3\}$ with probabilities {0.8, 0.02, 0.18}. If you compute the entropy of this we get $H(S) = 0.816\ bit/symbol$ and the Huffman code is give by:

Letter	Codeword
a_1	0
a_2	11
a_3	10

The average length of this code is equal to $l_{Huffman} = 1.2\ Bits/symbol,$ which is 0.384 bits/symbol away from the entropy bound.

In many applications, we can reduce the coding rate by creating blocks of symbols and code them together. In this case, it has been shown that the rate R of the code (slightly different than saying average length since we have blocks of symbols not individual symbols)

$$H(S) \le R \le H(S) + \frac{1}{n} \tag{3.4}$$

where n is the number of alphabet symbols we block them together. Let us now use this to encode pair of letters of this example.

Letter	Probability	Code
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

If we compute the average length for this extended code we get $l_{Huffman} = 1.7516\ Bits/Symbol$, which corresponds to 0.8758 Bits/Symbol per letter, which is only 0.06 bits/symbol away from the entropy. This we have achieved significant of compression and came very close to the theoretically achievable limit, the entropy.

Adaptive Huffman Coding:

Huffman coding above requires knowledge of the alphabet and the probabilities. The first is easy in most applications. However, statistics of real sources are not known as in image, audio and written data compression. Then we need a 2-pass procedure (i) to train the code by measuring probabilities and (ii) employ it on data. To make it practical these two steps must be integrated and a 1-pass procedure is needed, hence the need for an adaptive coding algorithm. One such algorithm is covered by Sayood (see reference cited in P.3.)[1] (Download a copy of a pdf file put together from that work.)

Sayood has applied this adaptive algorithm to lossless image. Audio, and text compression. Tables below display performance of Huffman (2-pass) and adaptive Huffman (1-pass) coding a set of four images.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	6.90	56,533	1.16
Sensin	7.38	60,474	1.27
Earth	4.78	39,193	1.67
Omaha	7.00	57,356	1.14

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	3.93	32,261	2.03
Sensin	4.63	37,896	1.73
Earth	4.82	39,504	1.66
Omaha	6.39	52,321	1.25

As we can see from these tables, there is a 50-100% improvement on compression when pixel differences are adaptively encoded instead of a two-pass non-adaptive Huffman coding.

Similarly, he has experimented with text of his Chapter 3, and the non-adaptive vs adaptive storage requirement were 70,000 bytes and 43,000 bytes, i.e., rather significant improvement.

[1] K. Sayood, *Introduction to Data Compression*, Morgan-Kaufman, 1996.

Finally, he has played with a CD-quality audio sampled at 44.1 kHz with 16-bit resolution. The results are shown in the following two tables (top: non-adaptive, bottom: adaptive).

Huffman coding of 16-bit CD-quality audio.

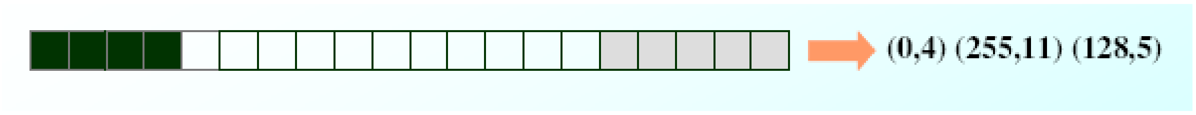
File Name	Original File Size (bytes)	Entropy (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	12.8	725,420	1.30
Cohn	402,442	13.8	349,300	1.15
Mir	884,020	13.7	759,540	1.16

Huffman coding of differences of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy of Differences (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	9.7	569,792	1.65
Cohn	402,442	10.4	261,590	1.54
Mir	884,020	10.9	602,240	1.47

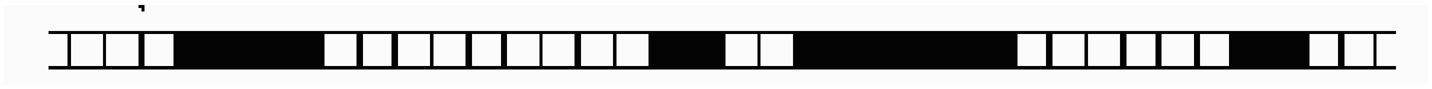
Run-Length Coding:

In a number of compression applications, such as image, fax, speech, we may see that a particular symbol repeats itself many times before being interrupted by another symbol. Consider a short string (20 pixels long) of an 8-bit digitized image (0-255) shows only three colors: Black: “0” White: “255” and Gray: “128”



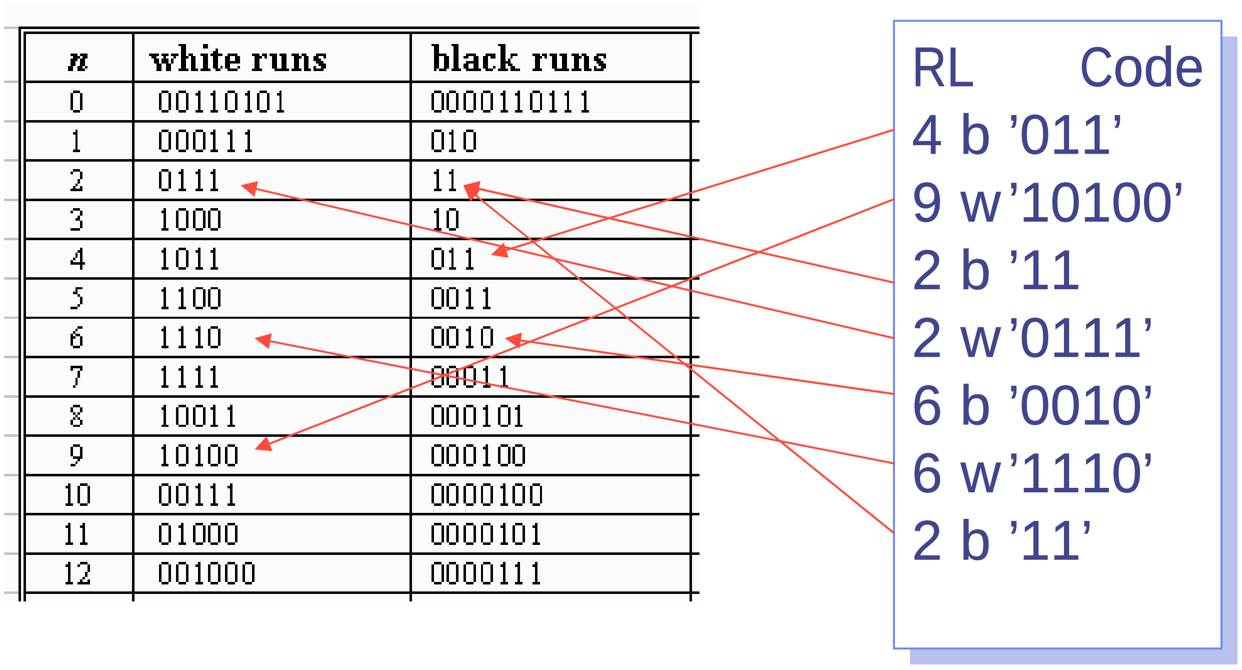
- If we want to apply compression on this string, we need to code 20 different pixels in some fashion.
- Let us recognize that there are consecutive groups of 4 black pixels, 11 white pixels and 5 gray pixels.
- Can we encode these three pairs of numbers? The answer is yes by using Run-Length Coding (RLC or RLE).

Example 3.4 RLC Based on ISO Standard:



Here we have only B/W pixels in the following fashion: …, 4b, 9w, 2b, 2w, 6b, 6w, 2b, ...

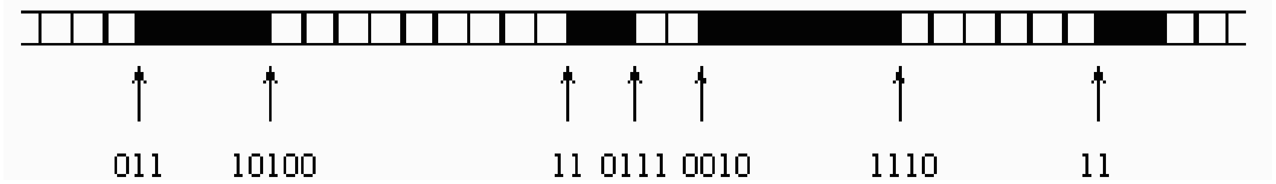
ISO Standard on RLC (RLE) has come up with variable-length binary codes for different runs of white and black pixels as shown in the table below. These codes are all pre-fix and there is no ambiguity when they arrive at the receiver. 4 black pixels are assigned “011” whereas 6 white pixels are represented by: “1110”



n	white runs	black runs
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100
10	00111	0000100
11	01000	0000101
12	001000	0000111

RL	Code
4 b	'011'
9 w	'10100'
2 b	'11
2 w	'0111'
6 b	'0010'
6 w	'1110'
2 b	'11'

Information transmitted for this piece of B/W picture becomes: ..0111010011011100101110111…..



- There are tables for 13-63 bits long white and black runs
- There are more coarse tables for 64-1728 bits long stings and finally
- EOL is represented by: 12 consecutive "0" followed by a single "1"
- Similarly, these three sets of tables for color imagery, one for each of red (R), green (G), and blue (B) components.

Example 3.5 RLC in JPEG: In JPEG image compression, one of the last steps is to encode certain coefficients after many processing stages using two quantizer: (a) DPCM quantizer[2] for the location (1,1) and RLC for the rest. If the following 8x8 was the resultant coefficient set:

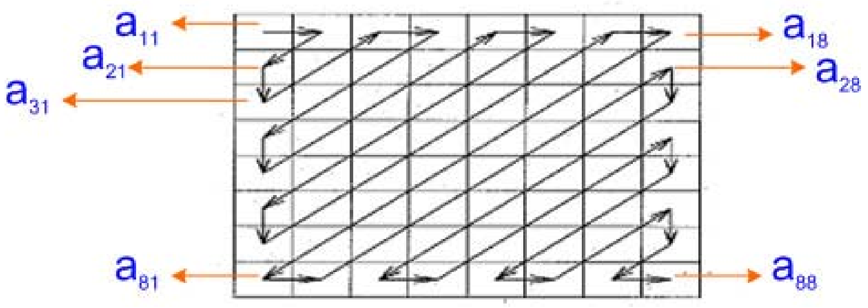
$$\begin{matrix} 88 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & -4 & 0 & 0 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{matrix}$$

Let us forget the term at location (1,1), i.e., 88. For the remaining 65 terms, we need at least 4 bits per symbol since we have the range between [-4 to +5]. The number of bits needed for this 8x8

[2] DPCM: Differential Pulse Code Modulation, which will be discussed later together with other lossy codecs.

data set would be: (4 bits/symbol x 65 symbols)= 260 bits. In addition we will need at least 7 bits to represent **88** but let us use 8 bits. Thus the overall bit rate would be **268** bits /block.

Now let us turn these 15 symbols into a 1-dimensional array in a zig-zag fashion used in JPEG and other image compression standards. This ordering is best seen in the following sketch:



After the scan, the order of terms in 8x8 would be:

(1,2), (2,1), 1 5	SUM of indices: 3
(3,1), (2,2), (1,3), -2 0 0	4
(1,4), (2,3), (3,2), (4,1), 0 -4 1 0	5
(5,1), (4,2), (3,3), (2,4), (1,5), 0 0 0 0 0	6
(1,6), (2,5), (3,4), (4,3), (5,2), (6,1), 0 0 0 0 0 0	7
(7,1), (6,2), (5,3), (4,4), (3,5), (2,6), (1,7), 2 0 0 3 0 0 0	8
(8,1), (2,7), (3,6), (5,4), (4,5), (5,4), (6,3), (7,2), (8,1), 0 0 0 0 0 0 0 0 0	9
(8,2), (7,3), (6,4), (5,5), (4,6), (3,7), (2,8), 0 0 0 0 0 0 0	10
(3,8), (4,7), (5,6), (6,5), (7,4),(8,3), 0 0 0 0 0 0	11
(8,4), (7,5), (6,6), (5,7), (4,8), 0 0 0 0 0	12
(5,8), (6,7), (7,6), (8,5), 0 0 0 0	13
(8,6), (7,7), (6,8), 0 0 0	14
(7,8), (8,7), 0 0	15
(8,8). -1	16

Let us write the run-lengths for this example:
(**1**,1), (**5**,1), (**-2**,1), (**0**,3), (**-4**,1), (**1**,1), (**0**,12), (**2**,1), (**0**,2), (**3**,1), (**0**,39), (**-1**,1)

Excluding **88**, we need to handle 12 sets of parameters instead of 65 pixels. Here we need to encode **0** four places 4/12; **1** two places 2/12, **2** once, **3** once, **5** once, **-1** once, **-2** once, **-4** once

1/12 each. Let us design an Huffman code for the amplitudes using the following source code and the function called Huffman

```
% Let us design a Huffmann code for the ac doefficients of an 4x4 image block for
% the example 3.5.
% dc coefficient, i.e., location (1,1) is normally encoded with a different codec (DPCM).
% Applied by: H. Abut - March 2006
% Enter data input
% Note: Our probability array was: prob= [4/12 2/12 1/12 1/12 1/12 1/12 1/12 1/12]
prob=input('Enter Values for Probabilities \n');
% Call Huffman code design function
[code,l_ave]=huffman(prob);
% Display results
disp ('Everage code length='); disp(l_ave);
disp('Code values');
disp(code)
```

Results:

```
Everage code length=
   2.8333
Code values
      0
     101
   1000
   1001
   1110
   1111
   1100
   1101
```

Bit Rate Computation: Let us use fixed number of bits for run sizes: since we have runs 1 – 39 we need 6 bits for each run.

Bits for a_{21} through a_{88} =1 bit x (4x 6)+ 3 bits x(2x6)+4 bits x(6x6)= 24+36+144=204

Total bits= 8 bits for a_{11}+204= 212 bits /block

Compression rate: 268/212=1: 1.264

All this effort did NOT yield much savings. However, in an image frame there are hundreds of 8x8 blocks and the types of runs will be repeated many times and the Huffman coder will have a much better compression feature.

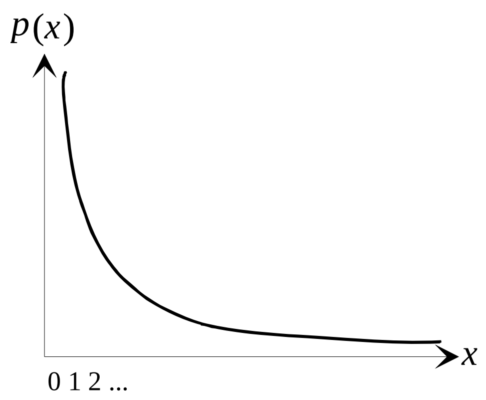
The m-file for Huffman Coder:

```
function [h,l]=huffman(p);
% HUFFMAN Huffman code generator
%       [h,l]=huffman(p), Huffman code generator
%       returns h the Huffman code matrix, and l the
%       average codeword length for a source with
%   probability vector p.
%   This function is included with permission from the text by Proakis and Salehi,
%   Contemporary Communication Systems, Chapter 4,
%   PWS Publishing Company, Boston, MA, 1998.

if length(find(p<0))~=0,
  error('Not a prob. vector, negative component(s)')
end
if abs(sum(p)-1)>10e-10,
  error('Not a prob. vector, components do not add up to 1')
end
n=length(p); q=p; m=zeros(n-1,n);
for i=1:n-1
  [q,l]=sort(q); m(i,:)=[l(1:n-i+1),zeros(1,i-1)]; q=[q(1)+q(2),q(3:n),1];
end
for i=1:n-1
  c(i,:)=blanks(n*n);
end
c(n-1,n)='0'; c(n-1,2*n)='1';
for i=2:n-1
  c(n-i,1:n-1)=c(n-i+1,n*(find(m(n-i+1,:)==1))...
  -(n-2):n*(find(m(n-i+1,:)==1)));
  c(n-i,n)='0'; c(n-i,n+1:2*n-1)=c(n-i,1:n-1); c(n-i,2*n)='1';
  for j=1:i-1
    c(n-i,(j+1)*n+1:(j+2)*n)=c(n-i+1,...
    n*(find(m(n-i+1,:)==j+1)-1)+1:n*find(m(n-i+1,:)==j+1));
  end
end
for i=1:n
  h(i,1:n)=c(1,n*(find(m(1,:)==i)-1)+1:find(m(1,:)==i)*n);
  l1(i)=length(find(abs(h(i,:))~=32));
end
l=sum(p.*l1);
```

Golomb-Rice codes

Golomb codes are a class of prefix codes which are suboptimal but very easy to implement. They are frequently used to encode differential (residual) signals formed by subtracting the difference between two speech segments or two consecutive image frames. These differences are known to have probability distributions (histograms) of two-sided exponential type (Laplacian). As in the case of Huffman coding, symbols are arranged in descending probability order, and non-negative integers are assigned to the symbols, beginning with 0 for the most probable symbol (see the figure.)



Probability distribution function assumed by Golomb and Rice codes.

To encode an integer *n*, it is divided into two components, to the most significant part n_M (quotient) and to the least significant part n_L (remainder):

$$n_M = \left\lfloor \frac{n}{m} \right\rfloor \quad and \quad n_L = n \bmod m \qquad n = n_M \cdot m + n_L \tag{3.2}$$

where $m = 2^k$ *m* is the parameter of the Golomb coding and *k* is a positive integer.

1. n_M is outputted using *unary code* as shown in the table below.
2. n_L is outputted using binary code.

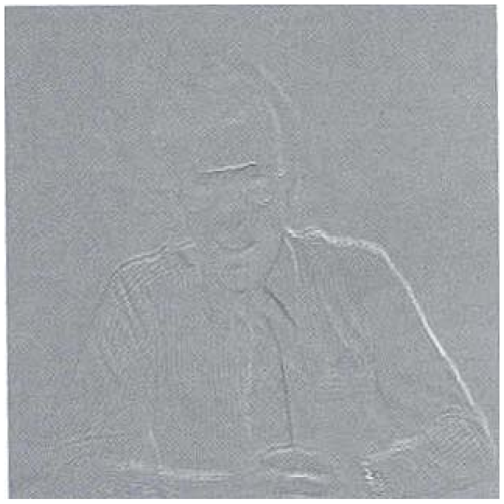
In practice, n_L is computed by masking out all but the *k* low order bits.

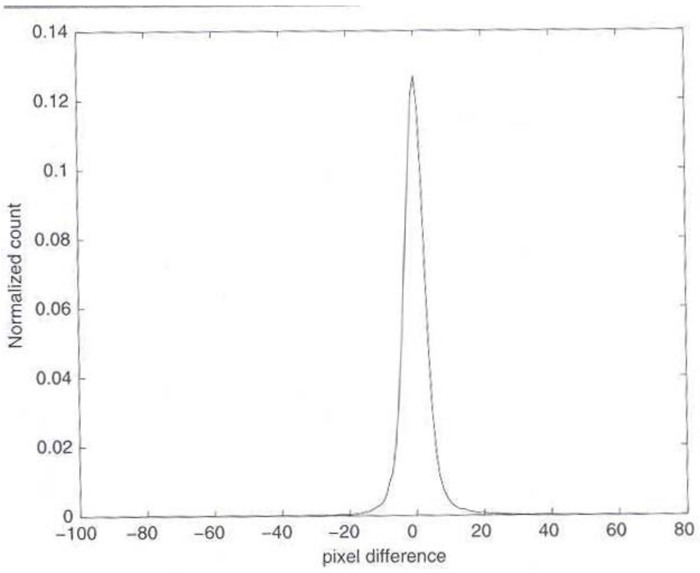
Example: 3.6 An example of the Golomb coding with the parameter $m = 4 = 2^2$:

n	n_M	n_L	**Code of n_M**	**Code of n_L**	**GR Code**
0	0	0	0	00	**000**
1	0	1	0	01	**001**
2	0	2	0	10	**010**
3	0	3	0	11	**011**
4	1	0	10	00	**1000**
5	1	1	10	01	**1001**
6	1	2	10	10	**1010**
7	1	3	10	11	**1011**
8	2	0	110	00	**11000**
:	:	:	:	:	:

Example 3.7 Golomb-Rice Coding of Trevor Image Sequence : Two consecutive frames (5 and 6) from the Trevor Sequence (one of the standard video streams used in image processing and compression community) and their pixel-by-pixel differences are shown in the next figure. The original pixel values were 8-bit (0-255) gray-levels, black=0 and white=255. Normalized histogram (approximation to probability density) versus pixel difference values are also shown. The last plot is very similar to the exponential (Laplacian) probability curve as discussed above.

Performance of the Golomb-Rice lossless compression for several frames are also shown below. Average bit rate was 4.357 bits/pixel as a result of an average compression ratio: 1:1.836





Frame #i + *1* – *frame* #i	*Bits/pixel*
6–5	4.35
7–6	4.37
8–7	4.36
9–8	4.40
10–9	4.38
11–10	4.38
12–11	4.35
13–12	4.33
14–13	4.33
15–14	4.32
Average bit rate	4.357
Mean comp. ratio	1.836

GRAY-CODE

Better results can be achieved if the binary representation of symbols (pixels) are transformed into *Gray codes* before partitioning an image into the bit planes. Consider an 8-bit image consisting of only two pixel values, which are either 127 and 128. Corresponding binary representations are:

0111 1111
1000 0000

Now, even if the image could be compressed by 1 bit/pixel by a trivial algorithm, the bit planes are completely random since the values 127 and 128 differ in every bit position.

Gray coding is a method of mapping a set of numbers into a binary alphabet such that successive numerical changes result in a change of only one bit in the binary representation. Thus, when two neighboring pixels differ by one, only a single bit plane is affected. Following figure shows the binary code (BC) and their corresponding Gray code representations (GC) in the case of 4-bit number of 0 to 15. One method of generating a Gray code representation from the binary code is by the following logical operation:

$$GC = BC \oplus (BC >> 1) \tag{3.3}$$

where "⊕" denotes bit-wise exclusive-OR operation, and ">>" denotes bit-wise logical right-shift operation. Note that the *i*th GC bit plane is constructed by performing an exclusive OR on the *i*th and (*i*+1)th BC bit planes. The most significant bit planes of the binary and the Gray codes are identical. Figures shows the bit planes of binary and Gray codes.

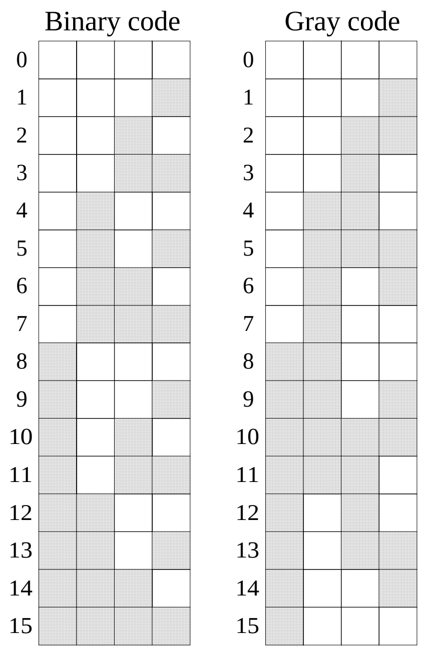
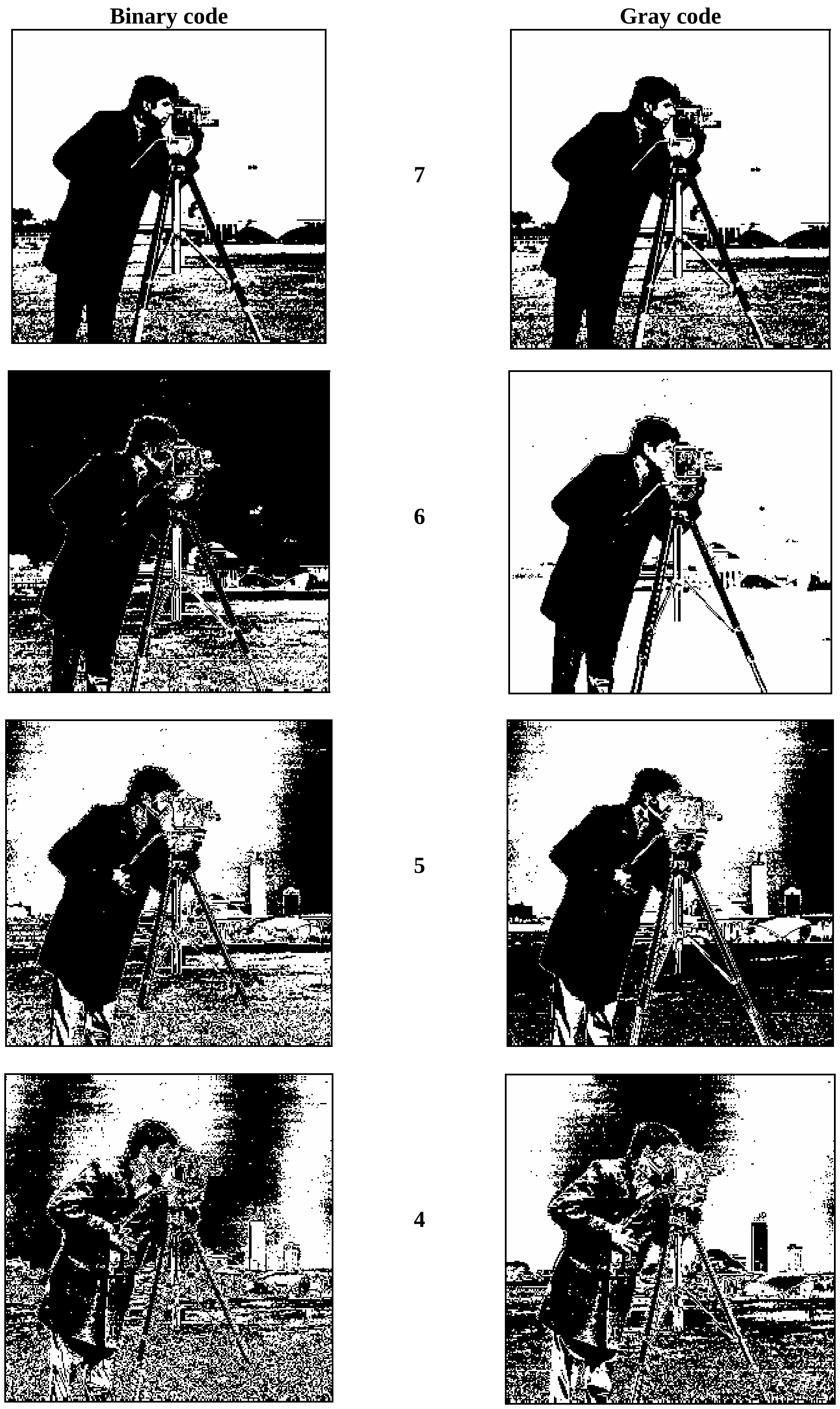
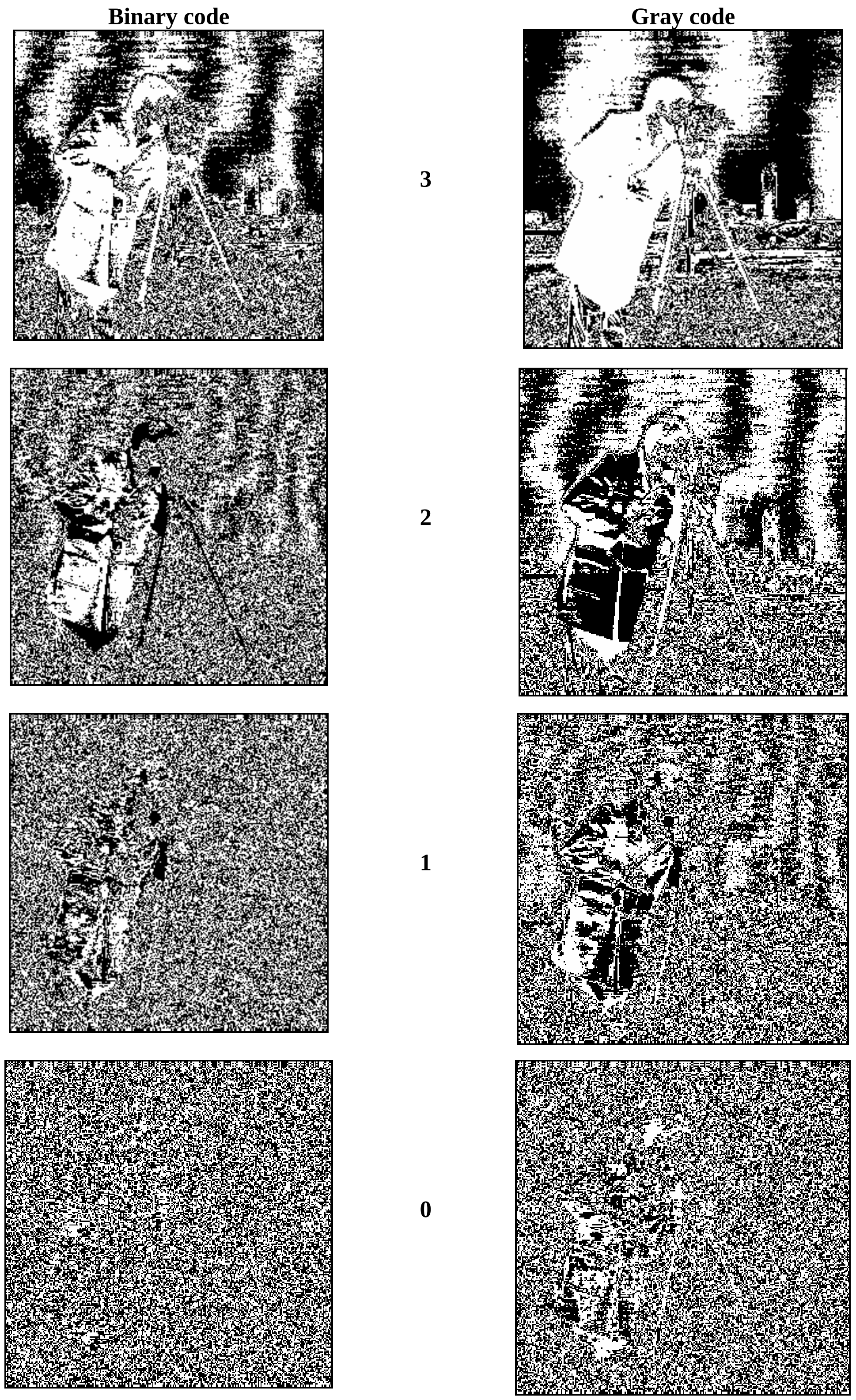


Illustration of four-bit binary and Gray codes.

Suppose that the bit planes are compressed by the MSP (most significant plane) first and the LSP last. A small improvement in a context based compression is achieved, if the context templates includes few pixels from the previously coded bit plane. Typically the bits included in the template are the bit of the same pixel that is to be coded, and possibly the one above it. This kind of templated is referred as a *3D-template*.



Binary and Gray code bit planes for test image *Camera*, bit planes 7 through 4.



Binary and Gray code bit planes for test image *Camera*, bit planes 3 through 0.

Lempel-Ziv Universal Coding: (LZW version is used in PKZip and Winzip)

Basic idea:

- Recursively parse input sequence into non-overlapping blocks of variable size, while constructing a dictionary of blocks seen thus far.
- Each sequence found in the dictionary is coded into its index in the dictionary.
- The dictionary is initialized with the available symbols, {0 & 1}.
- Each successive block in the parsing is chosen to be the largest (longest) word ***w*** that has appeared in the dictionary. Therefore, the word ***wa*** formed by concatenating ***w*** and the following symbol that is not in the dictionary.
- Before continuing the parsing ***wa*** is added to the dictionary and *a* becomes the first symbol in the next block.

Example 3.8: Consider the following input string:

0110011001011000010011 0

Steps:

1. Place parsed blocks in parentheses and
2. New dictionary word, i.e., the parsed block followed by the first symbol of the next block, as a subscript.

$$(\mathbf{0})_{01}(\mathbf{1})_{11}(\mathbf{1})_{10}(\mathbf{0})_{00}(\mathbf{01})_{011}(\mathbf{10})_{100}(\mathbf{01})_{010}(\mathbf{011})_{0110}(\mathbf{00})_{000}(\mathbf{00})_{001}(\mathbf{100})_{1001}(\mathbf{11})_{110}(\mathbf{0})$$

3. The parsed data implies a code by indexing the dictionary words in the order in which they were added and sending the indices to the decoder.
4. Decoder starts with the same tables as the encoder one and using the identical rules it decodes the input symbol string uniquely.

Example 3.9: Let us observe the animation on LZW Algorithm: http://www.data-compression.com/lempelziv.shtml