



# Fundamentals of Multimedia

2<sup>nd</sup> Edition 2014

Ze-Nian Li

Mark S. Drew


Jiangchuan Liu

Part II:

Multimedia Data Compression

Chapter 7 :

Lossless Compression Algorithms

- 
- In this Part we examine the role played in multimedia by data compression, perhaps the most important enabling technology that makes modern multimedia systems possible.
  - So much data exist, in archives, via streaming, and elsewhere, that it has become critical to compress this information.
  - We start off in Chap. 7 looking at lossless data compression i.e., involving no distortion of the original signal once it is decompressed or reconstituted.

# Source and Channel Coding

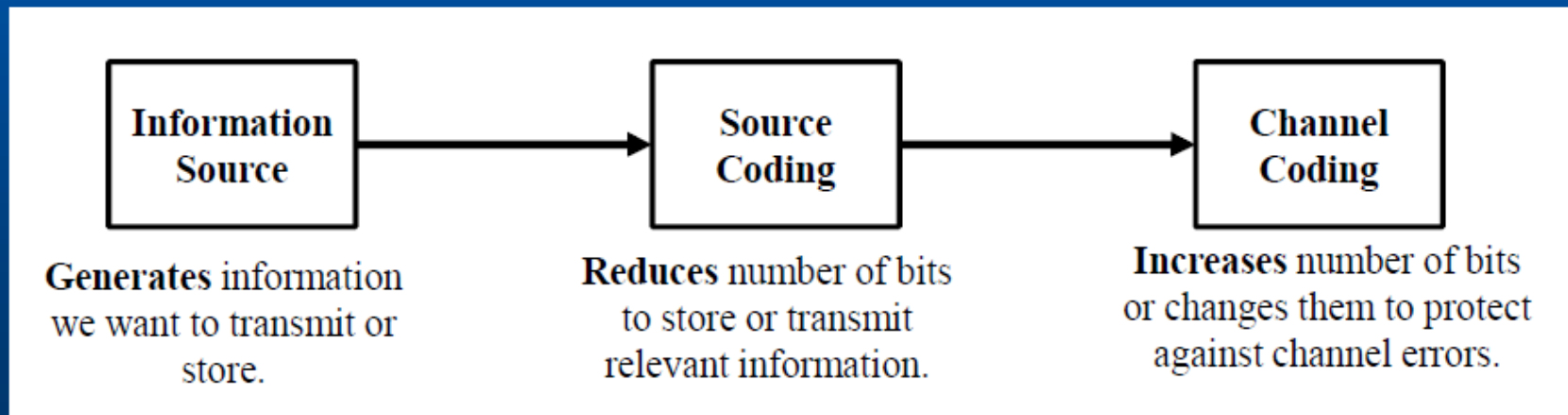
## Shannon's Separation Principle

### ◆ Assumptions:

- ◆ Single source and user
- ◆ Unlimited complexity and delay



Claude E. Shannon, 1916-2001



**Coding** related elements in a communication system.

What about joint source and channel coding?

# 7.1 Introduction

- **Compression:** the process of coding that will effectively reduce the total number of bits needed to represent certain information.
- Figure 7.1 depicts a general data compression scheme, in which compression is performed by an encoder and decompression is performed by a decoder.

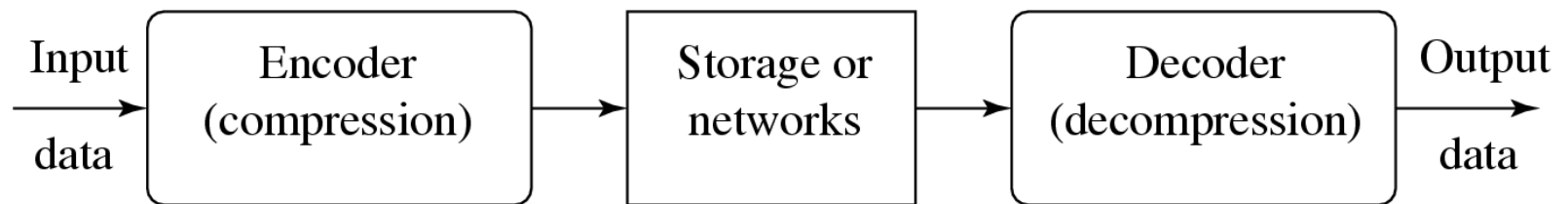


Fig. 7.1: A General Data Compression Scheme.

# 7.1 Introduction

- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**; otherwise, it is **lossy**.
- **Compression ratio:**

$$\text{compression ratio} = \frac{B_0}{B_1} \quad (7.1)$$

- $B_0$  – number of bits before compression
- $B_1$  – number of bits after compression
- In general, we would desire any codec (encoder/decoder scheme) to have a compression **ratio** much larger than **1.0**.
- The higher the compression ratio, the better the **lossless** compression scheme, as long as it is computationally feasible.



# Lossless and Lossy

## ◆ Lossless

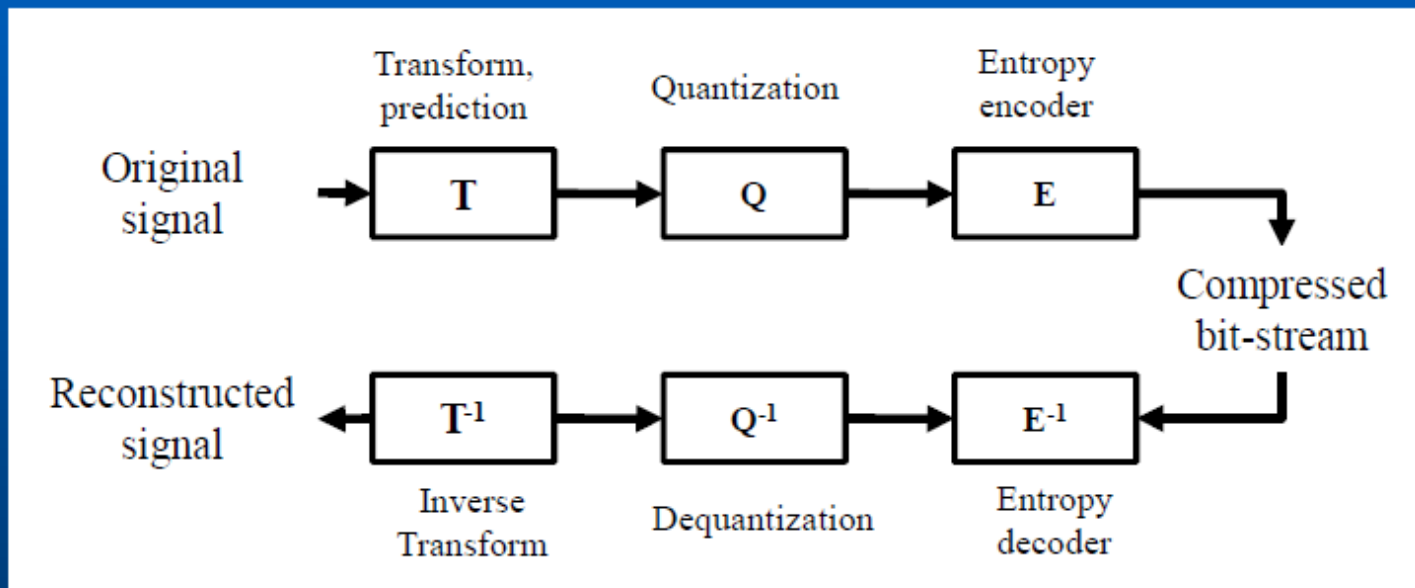
- ◆ **Exact reconstruction is possible.**
- ◆ **Applied to general data.**
- ◆ **Lower compression rates.**
- ◆ **Examples: Run-length, Huffman, Lempel-Ziv.**

## ◆ Lossy

- ◆ **Higher compression rates.**
- ◆ **Applied to audio, image and video.**
- ◆ **Examples: CELP, JPEG, MPEG-2.**



# Codec (Encoder and Decoder)



General structure of a Codec.

## 7.2 Basics of Information Theory

- The **entropy**  $\eta$  of an information source with alphabet  $S = \{s_1, s_2, \dots, s_n\}$  is:

$$\eta = H(S) = \sum_{i=1}^n p_i \log_2 \frac{1}{p_i} \quad \bullet \quad (7.2)$$

$$= - \sum_{i=1}^n p_i \log_2 p_i \quad \bullet \quad (7.3)$$

- $p_i$  – probability that symbol  $s_i$  will occur in  $S$ .
- $\log_2 \frac{1}{p_i}$  – indicates the amount of information (self-information as defined by Shannon) contained in  $s_i$  which corresponds to the number of bits needed to encode  $s_i$ .



# Entropy, Definition

- ◆ The entropy,  $H$ , of a discrete random variable  $X$  is a measure of the **amount of uncertainty** associated with the value of  $X$ .

Information Theory  $X \rightarrow$  Information Source

Point of View  $P(x) \rightarrow$  Probability that symbol  $x$  in  $X$  will occur

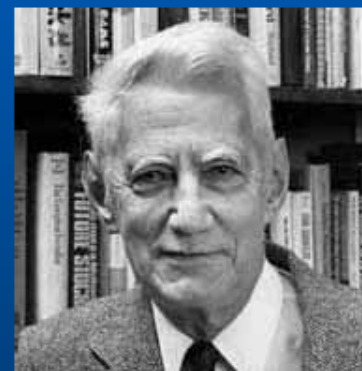
$$H(X) = \sum_{x \in X} P(x) \cdot \log_2 \frac{1}{P(x)}$$

- ◆ Measure of information content (in bits)
- ◆ A quantitative measure of the **disorder** of a system
- ◆ It is impossible to compress the data such that the average number of bits per symbol is less than the Shannon entropy of the source (in noiseless channel)
- ◆ The Intuition Behind the Formula

$$P(x) \uparrow \Rightarrow \text{amount of uncertainty} \downarrow \Rightarrow H \sim \frac{1}{P(x)}$$

bringing it to the world of bits  $\Rightarrow H \sim \log_2 \frac{1}{P(x)} = I(x)$ , information content of  $x$

weighted average number of bits required to encode each possible value  $\Rightarrow \sum x P(x)$  and  $\sum$

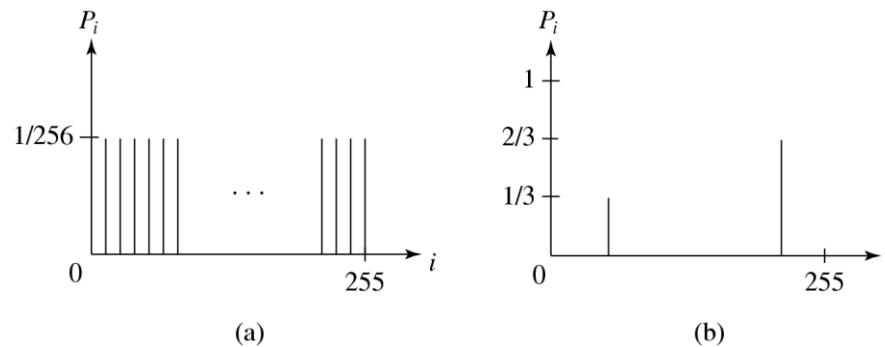


Claude E. Shannon  
1916-2001

## 7.2 Basics of Information Theory

- What is **entropy**? is a measure of the number of specific ways in which a system may be arranged, commonly understood as a measure of the disorder of a system.
- As an example, if the information source  $S$  is a **gray-level digital image**, each  $s_i$  is a gray-level intensity ranging from 0 to  $(2^k - 1)$ , where  $k$  is the number of bits used to represent each pixel in an uncompressed image.
- We need to find the entropy of this image; which the number of bits to represent the image after compression.

# Distribution of Gray-Level Intensities



- Fig. 7.2 Histograms for Two Gray-level Images.
- Fig. 7.2(a) shows the histogram of an image with *uniform* distribution of gray-level intensities, i.e.,  $\forall i p_i = 1/256$ . Hence, the entropy of this image is:
$$\log_2 256 = 8 \quad (7.4)$$
- Fig. 7.2(b) shows the histogram of an image with *two* possible values (binary image). Its entropy is 0.92.

# Distribution of Gray-Level Intensities

- It is interesting to observe that in the above uniform-distribution example (fig. 7-2 (a)) we found that  $\alpha = 8$ , *the minimum average number of bits to represent each gray-level intensity is at least 8*. **No** compression is possible for this image.
- In the context of imaging, this will correspond to the “**worst case**,” where neighboring pixel values have no similarity.

## Entropy and Code Length

- As can be seen in Eq. (7.3): the entropy  $\eta$  is a weighted-sum of terms  $\log_2 1/p_i$  hence it represents the *average* amount of information contained per symbol in the source  $S$ .
- The entropy  $\eta$  specifies the lower bound for the average number of bits to code each symbol in  $S$ , i.e.,

$$\eta \leq \bar{l} \quad (7.5)$$

$\bar{l}$  - the average length (measured in bits) of the codewords produced by the encoder.

## 7.3 Run-Length Coding

- RLC is one of the simplest forms of data compression.
- The basic idea is that if the information source has the property that symbols tend to form **continuous groups**, then such symbol and the length of the group can be coded.
- Consider a screen containing plain black text on a solid white background.
- There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. Let us take a hypothetical single scan line, with B representing a black pixel and W representing white:  
**WWWWWWBWWWWBWWWWWWWWBWWWW**
- If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:  
**5W1B4W3B6W1B3W**
- The run-length code represents the **original 21** characters in only **14**.



## 7.3 Run-Length Coding

- **Memoryless Source:** an information source that is independently distributed. Namely, the value of the current symbol does not depend on the values of the previously appeared symbols.
- Instead of assuming memoryless source, *Run-Length Coding (RLC)* exploits memory present in the information source.
- **Rationale for RLC:** if the information source has the property that symbols tend to form continuous groups, then such symbol and the length of the group can be coded.

## Run-length encoding

BBBBHHDDXXXKKKKWWZZZZ



4B2H2D4X4K2W4Z

```
00000000000000000000000000000000000000
00000000000000000000000000000000000000
0000000000011111111111111111110000000000
000000000001000000000000000000010000000000
000000000001000000000000000000010000000000
000000000001000000000000000000010000000000
00000000000111111111111111111100000000000
00000000000000000000000000000000000000
```

Image of a rectangle



0, 40

0, 40

0,10 1,20 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,20 0,10

0,40

## 7.4 Variable-Length Coding

Variable-length coding (**VLC**) is one of the best-known entropy coding methods

Here, we will study the

- Shannon–Fano algorithm,
- Huffman coding, and
- adaptive Huffman coding.

# VLC vs FLC Coding

## Fixed Length Coding (FLC)

A simple example

The message to code:



Message length: 10 symbols

5 different symbols → at least 3 bits

Codeword table

|   |     |
|---|-----|
| ▶ | 000 |
| ♣ | 001 |
| 😊 | 010 |
| ♠ | 011 |
| ☀ | 100 |






Total bits required to code:  $10 * 3 = 30$  bits

# Variable Length Coding (VLC)

**Intuition:** Those symbols that are more frequent should have smaller codes, yet since their length is not the same, there must be a way of distinguishing each code

The message to code: 

Codeword table

| Symbol   | Freq. | Code |
|--|-------|------|
|   | 3     | 00   |
|   | 3     | 01   |
|   | 2     | 10   |
|   | 1     | 110  |
|  | 1     | 111  |

To identify end of a codeword as soon as it arrives, no codeword can be a **prefix** of another codeword

How to find the optimal codeword table?

Total bits required to code:  $3*2 + 3*2 + 2*2 + 3 + 3 = 24$  bits

## 7.4.1 Shannon–Fano Algorithm

- To illustrate the algorithm, let us suppose the symbols to be coded are the characters in the word HELLO.
- The frequency count of the symbols is

|        |   |   |   |   |
|--------|---|---|---|---|
| Symbol | H | E | L | O |
| Count  | 1 | 1 | 2 | 1 |

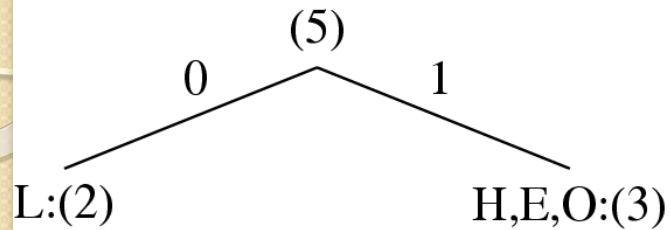
- The encoding steps of the Shannon–Fano algorithm can be presented in the following *top-down manner*:
- 1. Sort the symbols according to the frequency count of their occurrences.
- 2. Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.



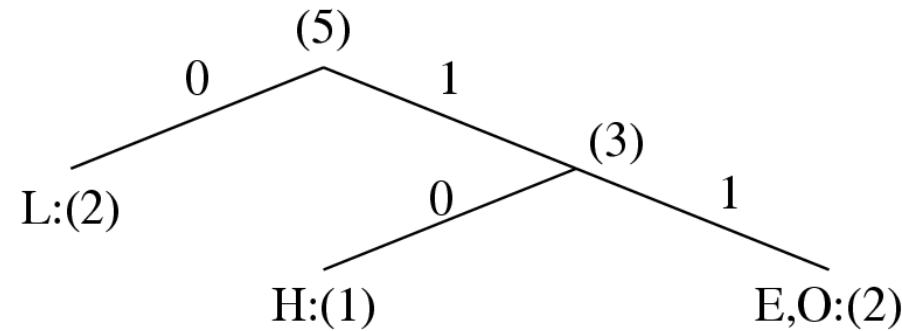
## 7.4.1 Shannon–Fano Algorithm

- A natural way of implementing the above procedure is to build a binary tree.
- As a convention, let us assign bit 0 to its left branches and 1 to the right branches.
- Initially, the symbols are sorted as LHEO.
- As Fig. 7.3 shows, the first division yields two parts: L with a count of 2, denoted as L:(2); and H, E and O with a total count of 3, denoted as H, E, O:(3).
- The second division yields H:(1) and E, O:(2).
- The last division is E:(1) and O:(1).

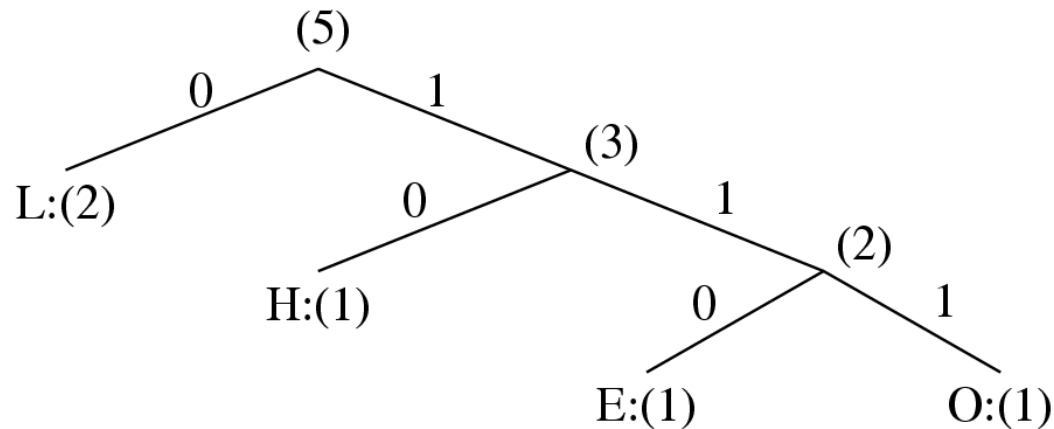
## 7.4.1 Shannon–Fano Algorithm



(a)



(b)

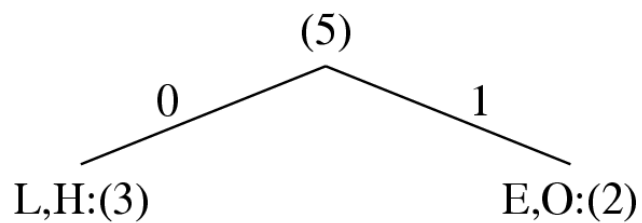


(c)

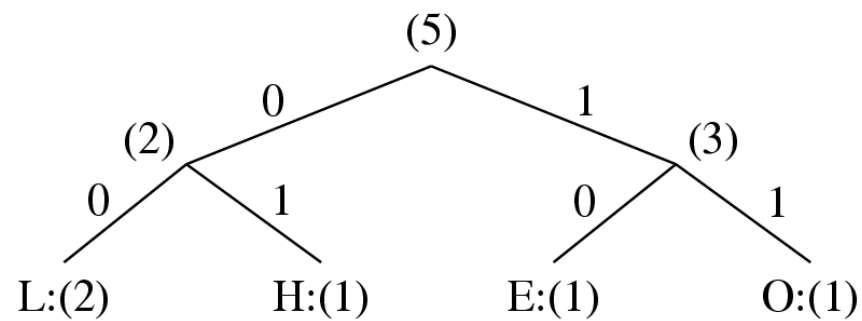
Fig. 7.3: Coding Tree for HELLO by Shannon-Fano.

# Table 7.1: Result of Performing Shannon-Fano on HELLO

| Symbol           | Count | $\log_2 \frac{1}{p_i}$ | Code | # of bits used |
|------------------|-------|------------------------|------|----------------|
| L                | 2     | 1.32                   | 0    | 2              |
| H                | 1     | 2.32                   | 10   | 2              |
| E                | 1     | 2.32                   | 110  | 3              |
| O                | 1     | 2.32                   | 111  | 3              |
| TOTAL # of bits: |       |                        |      | 10             |



(a)



(b)

Fig. 7.4 Another coding tree for HELLO by Shannon-Fano.

**Table 7.2: Another Result of Performing Shannon-Fano on HELLO (see Fig. 7.4)**

| Symbol           | Count | $\log_2 \frac{1}{p_i}$ | Code | # of bits used |
|------------------|-------|------------------------|------|----------------|
| L                | 2     | 1.32                   | 00   | 4              |
| H                | 1     | 2.32                   | 01   | 2              |
| E                | 1     | 2.32                   | 10   | 2              |
| O                | 1     | 2.32                   | 11   | 2              |
| TOTAL # of bits: |       |                        |      | 10             |

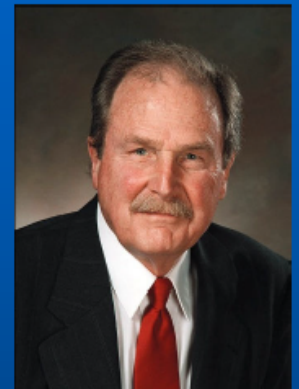
## 7.4.1 Huffman Coding Algorithm

- The Shannon–Fano algorithm delivers satisfactory coding results for data compression, but it was soon **outperformed** and overtaken by the **Huffman coding method**.
- The Huffman algorithm requires prior statistical knowledge about the information source, and such information is often not available.
- This is particularly true in multimedia applications, where future data is unknown before its arrival, as for example in live (or streaming) audio and video.
- Even when the statistics are available, the transmission of the symbol table could represent heavy overhead
- The solution is to use ***adaptive Huffman coding compression algorithms***, in which statistics are gathered and updated dynamically as the data stream arrives.



# Huffman Coding Algorithm

- ◆ Paper: "A Method for the Construction of Minimum-Redundancy Codes", 1952
- ◆ Results in "prefix-free codes"
- ◆ Most efficient
  - ◆ No other mapping will produce a smaller average output size,
  - ◆ If the actual symbol frequencies agree with those used to create the code.
- ◆ Cons:
  - ◆ Have to run through the **entire data** in advance to find frequencies.
  - ◆ 'Minimum-Redundancy' is not favorable for **error correction** techniques (bits are not predictable if e.g. one is missing).
  - ◆ Does not support **block of symbols**: Huffman is designed to code single characters only. Therefore at least one bit is required per character, e.g. a word of 8 characters requires at least an 8 bit code.



David A. Huffman  
1925-1999

## Huffman Coding

**ALGORITHM 7.1 Huffman Coding Algorithm** — a bottom-up approach

1. Initialization: Put all symbols on a list sorted according to their frequency counts.
2. Repeat until the list has only one symbol left:
  - (1) From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node.
  - (2) Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
  - (3) Delete the children from the list.
3. Assign a codeword for each leaf based on the path from the root.

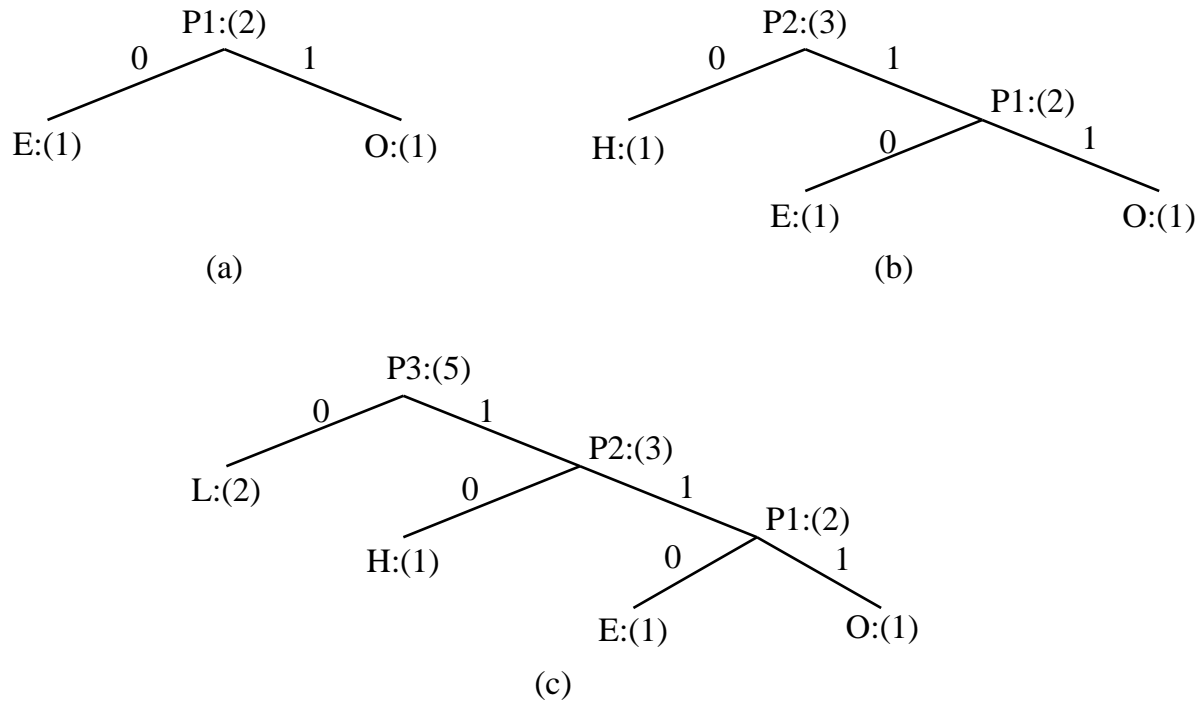


Fig. 7.5: Coding Tree for "HELLO" using the Huffman Algorithm.

## Huffman Coding (cont'd)

In Fig. 7.5, new symbols P1, P2, P3 are created to refer to the parent nodes in the Huffman coding tree. The contents in the list are illustrated below:

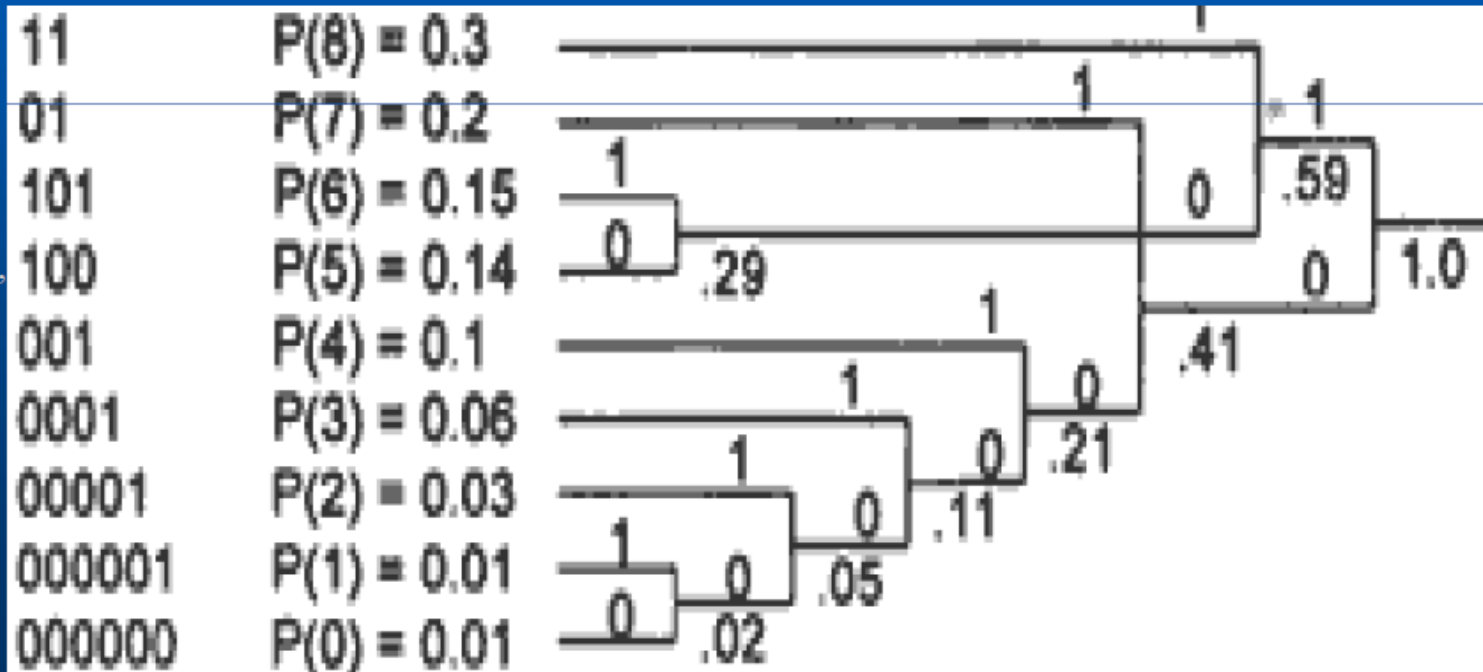
|                       |         |
|-----------------------|---------|
| After initialization: | L H E O |
| After iteration (a):  | L P1 H  |
| After iteration (b):  | L P2    |
| After iteration (c):  | P3      |

# Huffman Coding Algorithm

- ◆ **Step 1: Take the two least probable symbols in the alphabet**
  - ◆ (longest codewords, equal length, differing in last digit)
- ◆ **Step2: Combine these two symbols into a single symbol, and repeat.**

P(n): Probability of  
symbol number n  
Here there is 9 symbols.

e.g. symbols can be  
alphabet letters 'a', 'b', 'c',  
'd', 'e', 'f', 'g', 'h', 'i'



## Properties of Huffman Coding

1. **Unique Prefix Property:** No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.
2. **Optimality:** *minimum redundancy code* - proved *optimal* for a given data model (i.e., a given, accurate, probability distribution):
  - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
  - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
  - The average code length for an information source  $S$  is strictly less than  $\eta + 1$ . Combined with Eq. (7.5), we have:

$$\bar{l} < \eta + 1 \quad (7.6)$$



## Extended Huffman Coding

- **Motivation:** All codewords in Huffman coding have integer bit lengths. It is wasteful when  $p_i$  is very large and hence  $\log_2 \frac{1}{p_i}$  is close to 0.

Why not group several symbols together and assign a single codeword to the group as a whole?

- **Extended Alphabet:** For alphabet  $S = \{s_1, s_2, \dots, s_n\}$ , if  $k$  symbols are grouped together, then the *extended alphabet* is:

$$S_{(k)} = \{\overbrace{s_1 s_1 \dots s_1}^{k \text{ symbols}}, \overbrace{s_1 s_1 \dots s_2}^{k \text{ symbols}}, \dots, \overbrace{s_1 s_1 \dots s_n}^{k \text{ symbols}}, \overbrace{s_1 s_1 \dots s_2 s_1}^{k \text{ symbols}}, \dots, \overbrace{s_n s_n \dots s_n}^{k \text{ symbols}}\}.$$

— the size of the new alphabet  $S_{(k)}$  is  $nk$ .

## Extended Huffman Coding (cont'd)

- It can be proven that the average # of bits for each symbol is:

$$\eta \leq \bar{l} < \eta + \frac{1}{k} \quad (7.7)$$

An improvement over the original Huffman coding, but not much.

- **Problem:** If  $k$  is relatively large (e.g.,  $k \geq 3$ ), then for most practical applications where  $n \gg 1$ ,  $nk$  implies a huge symbol table — impractical.

## Adaptive Huffman Coding

- **Adaptive Huffman Coding:** statistics are gathered and updated dynamically as the data stream arrives.

ENCODER

-----

```
Initial_code();
while not EOF
{
    get(c);
    encode(c);
    update_tree(c);
}
```

DECODER

-----

```
Initial_code();
while not EOF
{
    decode(c);
    output(c);
    update_tree(c);
}
```

## Adaptive Huffman Coding (Cont'd)

- *Initial code* assigns symbols with some initially agreed upon codes, without any prior knowledge of the frequency counts.
- *update tree* constructs an Adaptive Huffman tree.

It basically does two things:

- (a) increments the frequency counts for the symbols (including any new ones).
  - (b) updates the configuration of the tree.
- The *encoder* and *decoder* must use exactly the same *initial code* and *update tree* routines.

## Notes on Adaptive Huffman Tree Updating

- Nodes are numbered in order from left to right, bottom to top. The numbers in parentheses indicates the count.
- The tree must always maintain its *sibling property*, i.e., all nodes (internal and leaf) are arranged in the order of increasing counts.

If the sibling property is about to be violated, a *swap* procedure is invoked to update the tree by rearranging the nodes.

- When a swap is necessary, the farthest node with count  $N$  is swapped with the node whose count has just been increased to  $N + 1$ .

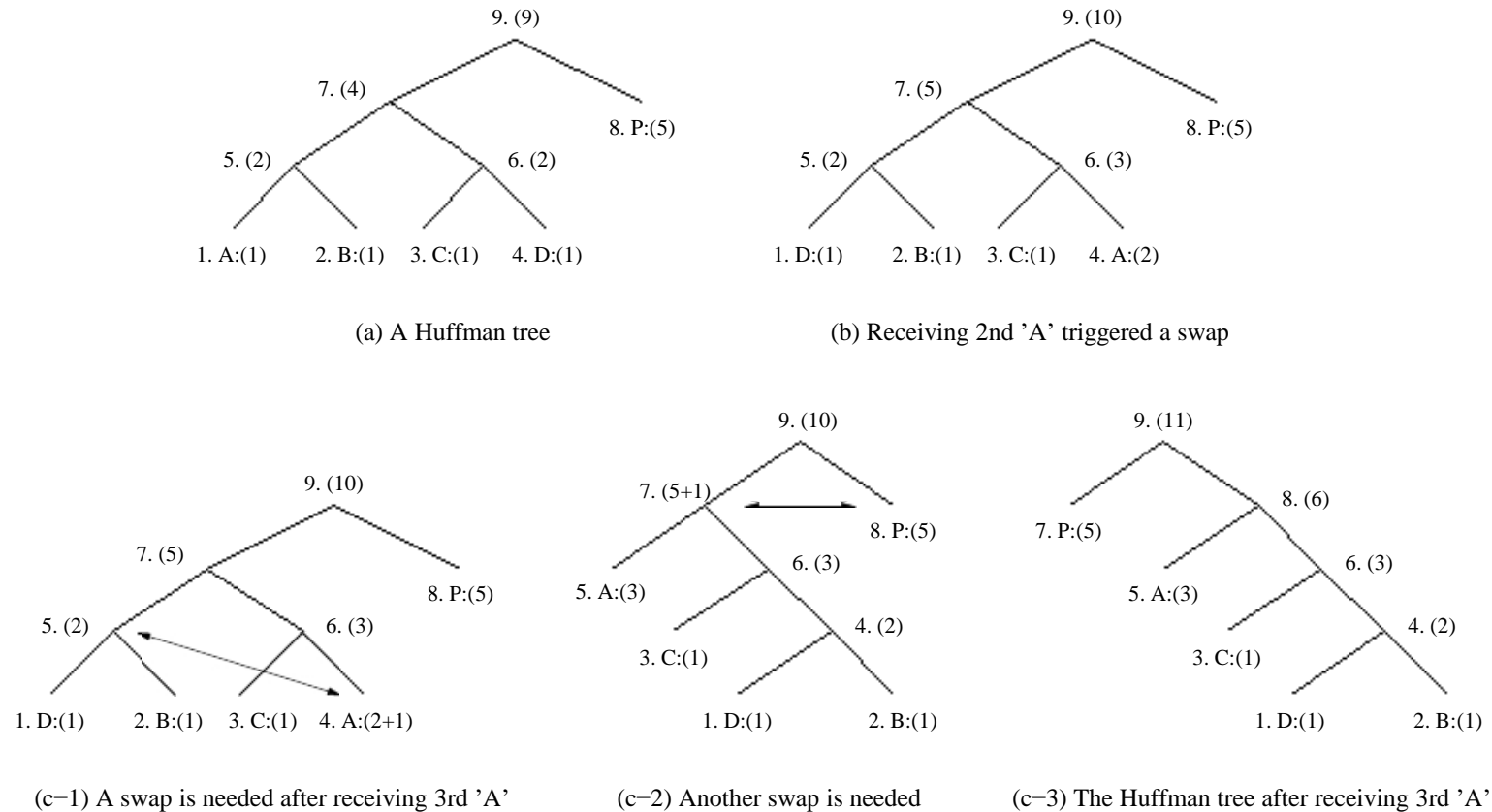


Fig. 7.6: Node Swapping for Updating an Adaptive Huffman Tree

## Another Example: Adaptive Huffman Coding

- This is to clearly illustrate more implementation details. We show exactly what *bits* are sent, as opposed to simply stating how the tree is updated.
- An additional rule: if any character/symbol is to be sent the first time, it must be preceded by a special symbol, NEW. The initial code for NEW is 0. The *count* for NEW is always kept as 0 (the count is never increased); hence it is always denoted as NEW:(0) in Fig. 7.7.



**Table 7.3: Initial code assignment for AADCCDD using adaptive Huffman coding.**

| <i>Initial Code</i> |       |
|---------------------|-------|
| NEW:                | 0     |
| A:                  | 00001 |
| B:                  | 00010 |
| C:                  | 00011 |
| D:                  | 00100 |
| .                   | .     |
| .                   | .     |
| .                   | .     |

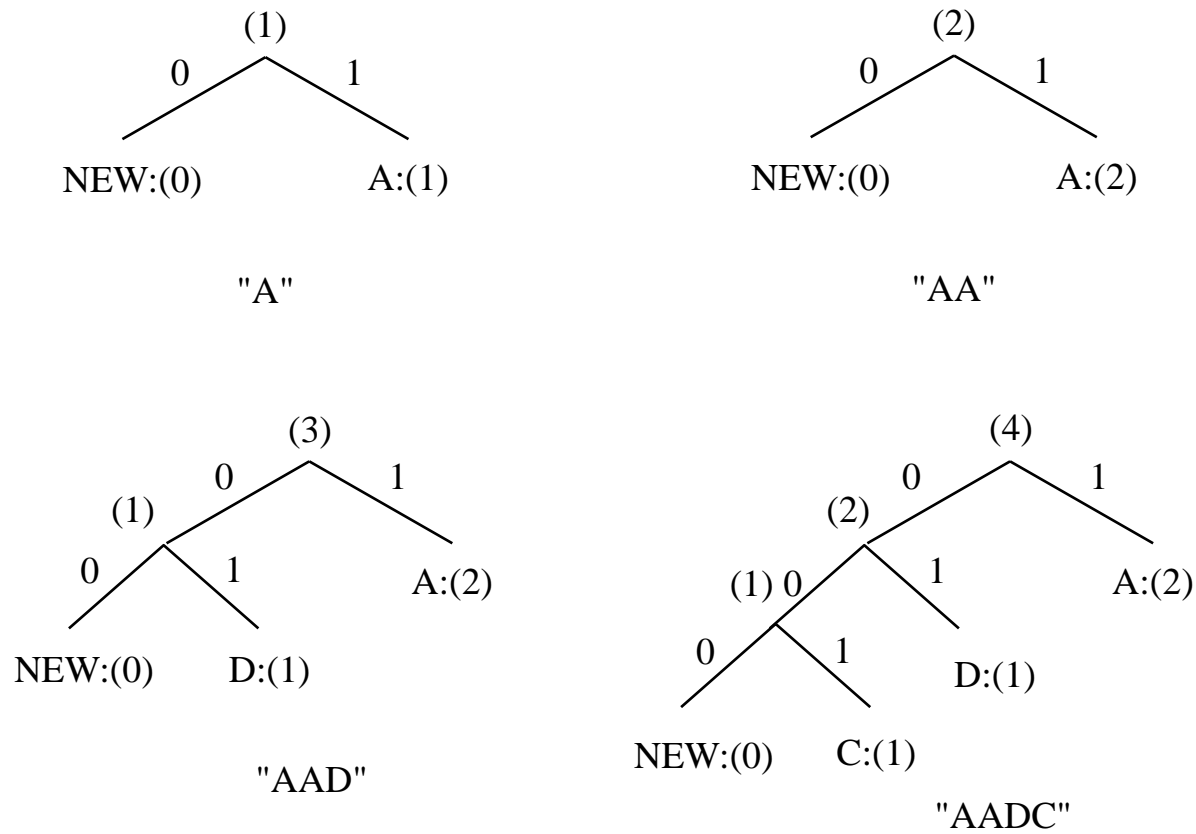
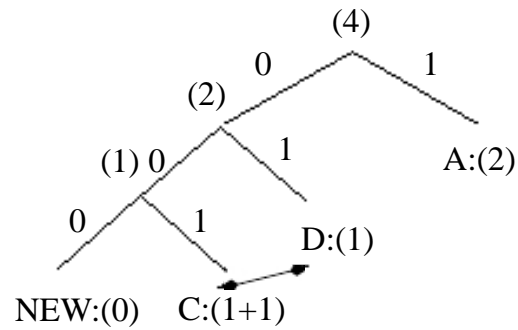
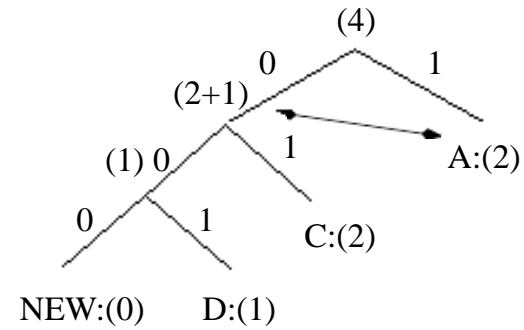


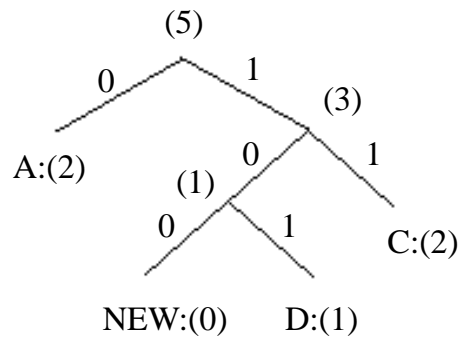
Fig. 7.7 Adaptive Huffman tree for AADCCDD.



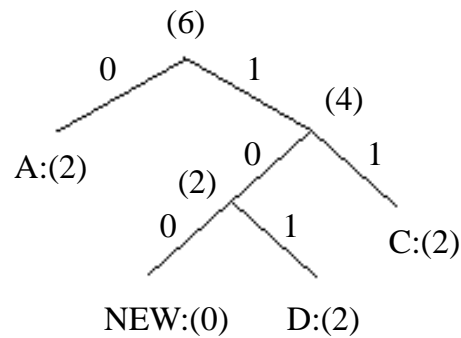
"AADCC" Step 1



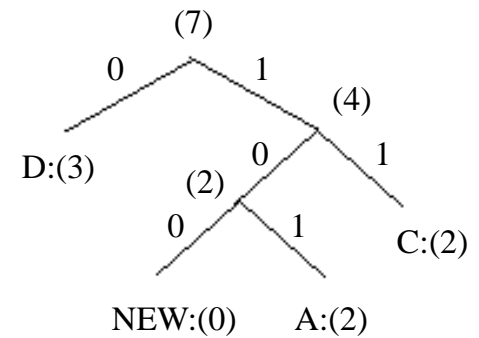
"AADCC" Step 2



"AADCC" Step 3



"AADCCD"



"AADCCDD"

Fig. 7.7 (cont'd) Adaptive Huffman tree for AADCCDD.

**Table 7.4      Sequence of symbols and codes sent to the decoder**

|        |     |       |   |     |       |     |       |     |     |     |
|--------|-----|-------|---|-----|-------|-----|-------|-----|-----|-----|
| Symbol | NEW | A     | A | NEW | D     | NEW | C     | C   | D   | D   |
| Code   | 0   | 00001 | 1 | 0   | 00100 | 00  | 00011 | 001 | 101 | 101 |

- It is important to emphasize that the code for a particular symbol changes during the adaptive Huffman coding process.

For example, after AADCCDD, when the character D overtakes A as the most frequent symbol, its code changes from 101 to 0.

- The "Squeeze Page" on this book's web site provides a Java applet for adaptive Huffman coding.

## 7.5 Dictionary-Based Coding

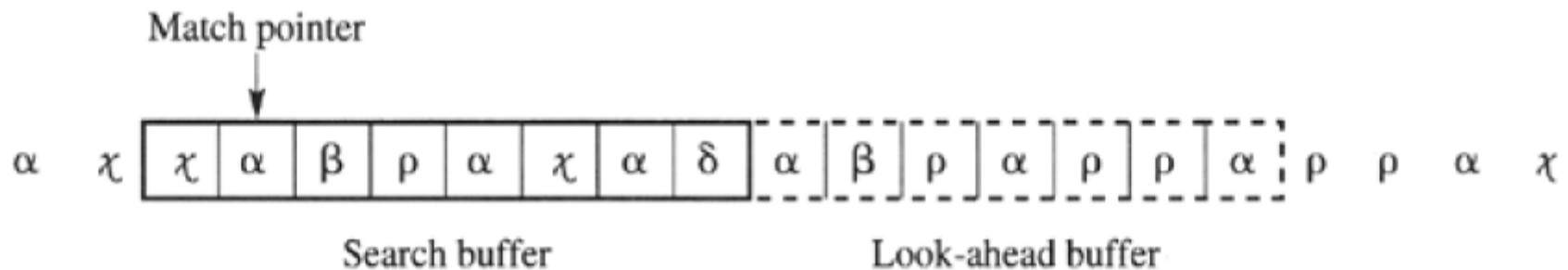
- The Lempel-Ziv-Welch (LZW) algorithm employs an adaptive, dictionary-based compression technique.
- Unlike variable-length coding, in which the lengths of the codewords are different, LZW uses fixed-length codewords to represent variable length strings of symbols/characters that commonly occur together, such as words in English text.
- As in the other adaptive compression techniques, the LZW encoder and decoder builds up the same dictionary dynamically while receiving the data—the encoder and the decoder both develop the same dictionary.

## 7.5 Dictionary-Based Coding

- LZW proceeds by placing longer and longer repeated entries into a dictionary, then emitting (sending) the *code for an element rather than the string itself, if the element has already been placed in the dictionary.*
- Remember, the LZW is an adaptive algorithm, in which the encoder and decoder independently build their own string tables. Hence, there is no overhead involving transmitting the string table.
- LZW is used in many applications, such as UNIX compress, GIF for images, WinZip, and others.

# Lempel-Ziv (LZ77)

- ❖ Algorithm for compression of **character sequences**
- ❖ Assumption: Sequences of characters are repeated
- ❖ Idea: Replace a character sequence by **a reference to an earlier occurrence**
  - ❖ 1. Define a: search buffer = (portion) of recently encoded data  
look-ahead buffer = not yet encoded data
  - ❖ 2. Find the longest match between  
the first characters of the look ahead buffer  
and an arbitrary character sequence in the search buffer
  - ❖ 3. Produces output <offset, length, next\_character>  
offset + length = reference to earlier occurrence  
next\_character = the first character following the match in the look ahead buffer





# Lempel-Ziv-Welch (LZW)

- ❖ Drops the **search buffer** and keeps an explicit **dictionary**
- ❖ Produces only output <index>
- ❖ Used by unix "compress", "GIF", "V24.bis", "TIFF"
- ❖ Example: wabbapwabbapwabbapwabbapwoopwoopwoo

Progress clip at 12<sup>th</sup> entry

**Initial LZW dictionary.**

| Index | Entry    |
|-------|----------|
| 1     | <i>⌀</i> |
| 2     | <i>a</i> |
| 3     | <i>b</i> |
| 4     | <i>o</i> |
| 5     | <i>w</i> |

Encoder output sequence so far: 5 2 3 3 2 1

**Constructing the 12<sup>th</sup> entry of the LZW dictionary.**

| Index | Entry       |
|-------|-------------|
| 1     | <i>⌀</i>    |
| 2     | <i>a</i>    |
| 3     | <i>b</i>    |
| 4     | <i>o</i>    |
| 5     | <i>w</i>    |
| 6     | <i>wa</i>   |
| 7     | <i>ab</i>   |
| 8     | <i>bb</i>   |
| 9     | <i>ba</i>   |
| 10    | <i>a⌀</i>   |
| 11    | <i>⌀w</i>   |
| 12    | <i>w...</i> |

## ALGORITHM 7.2 LZW Compression

BEGIN

  s = next input character;

  while not EOF

    { c = next input character;

      if s + c exists in the dictionary

        s = s + c;

      else

        { output the code for s;

          add string s + c to the dictionary with a new code;

          s = c;

        }

    }

  output the code for s;

END

# Lempel-Ziv-Welch (LZW)

❖ **Example: wabbapwabbapwabbapwabbapwoopwoopwoo**

Progress clip at the end of above example

**Initial LZW dictionary.**

| Index | Entry    |
|-------|----------|
| 1     | <i>b</i> |
| 2     | <i>a</i> |
| 3     | <i>b</i> |
| 4     | <i>o</i> |
| 5     | <i>w</i> |

**The LZW dictionary for encoding  
wabbapwabbapwabbapwabbapwoopwoopwoo.**

| Index | Entry      | Index | Entry       |
|-------|------------|-------|-------------|
| 1     | <i>b</i>   | 14    | <i>abw</i>  |
| 2     | <i>a</i>   | 15    | <i>wabb</i> |
| 3     | <i>b</i>   | 16    | <i>bab</i>  |
| 4     | <i>o</i>   | 17    | <i>bwa</i>  |
| 5     | <i>w</i>   | 18    | <i>abb</i>  |
| 6     | <i>wa</i>  | 19    | <i>babw</i> |
| 7     | <i>ab</i>  | 20    | <i>wo</i>   |
| 8     | <i>bb</i>  | 21    | <i>oo</i>   |
| 9     | <i>ba</i>  | 22    | <i>ob</i>   |
| 10    | <i>ab</i>  | 23    | <i>bwo</i>  |
| 11    | <i>bw</i>  | 24    | <i>oob</i>  |
| 12    | <i>wab</i> | 25    | <i>bwoo</i> |
| 13    | <i>bba</i> |       |             |

Encoder output sequence: 5 2 3 3 2 1  
6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

### **Example 7.2 LZW compression for string “ABABBAB-CABABBA”**

- Let's start with a very simple dictionary (also referred to as a “string table”), initially containing only 3 characters, with codes as follows:

| code  | string |
|-------|--------|
| ----- |        |
| 1     | A      |
| 2     | B      |
| 3     | C      |

- Now if the input string is “ABABBAB-CABABBA”, the LZW compression algorithm works as follows:

| s     | c   | output | code | string |
|-------|-----|--------|------|--------|
| <hr/> |     |        |      |        |
|       |     |        | 1    | A      |
|       |     |        | 2    | B      |
|       |     |        | 3    | C      |
| <hr/> |     |        |      |        |
| A     | B   | 1      | 4    | AB     |
| B     | A   | 2      | 5    | BA     |
| A     | B   |        |      |        |
| AB    | B   | 4      | 6    | ABB    |
| B     | A   |        |      |        |
| BA    | B   | 5      | 7    | BAB    |
| B     | C   | 2      | 8    | BC     |
| C     | A   | 3      | 9    | CA     |
| A     | B   |        |      |        |
| AB    | A   | 4      | 10   | ABA    |
| A     | B   |        |      |        |
| AB    | B   |        |      |        |
| ABB   | A   | 6      | 11   | ABBA   |
| A     | EOF | 1      |      |        |

- The output codes are: 1 2 4 5 2 3 4 6 1. Instead of sending 14 characters, only 9 codes need to be sent (compression ratio =  $14/9 = 1.56$ ).

### ALGORITHM 7.3 LZW Decompression (simple version)

```
BEGIN
  s = NIL;
  while not EOF
  {
    k = next input code;
    entry = dictionary entry for k;
    output entry;
    if (s != NIL)
      add string s + entry[0] to dictionary with a new code;
    s = entry;
  }
END
```

**Example 7.3:** LZW decompression for string "ABABBABCABABBA".

Input codes to the decoder are 1 2 4 5 2 3 4 6 1.

The initial string table is identical to what is used by the encoder.

The LZW decompression algorithm then works as follows:

| s     | k   | entry/output | code | string |
|-------|-----|--------------|------|--------|
| <hr/> |     |              |      |        |
|       |     |              | 1    | A      |
|       |     |              | 2    | B      |
|       |     |              | 3    | C      |
| <hr/> |     |              |      |        |
| NIL   | 1   | A            |      |        |
| A     | 2   | B            | 4    | AB     |
| B     | 4   | AB           | 5    | BA     |
| AB    | 5   | BA           | 6    | ABB    |
| BA    | 2   | B            | 7    | BAB    |
| B     | 3   | C            | 8    | BC     |
| C     | 4   | AB           | 9    | CA     |
| AB    | 6   | ABB          | 10   | ABA    |
| ABB   | 1   | A            | 11   | ABBA   |
| A     | EOF |              |      |        |

Apparently, the output string is "ABABBABCABABBA", a truly lossless result!



## **ALGORITHM 7.4 LZW Decompression (modified)**

BEGIN

    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 string s + entry[0] to dictionary with a new code;

        s = entry;

    }

END

## LZW Coding (cont'd)

- In real applications, the code length  $l$  is kept in the range of  $[l_0, l_{max}]$ . The dictionary initially has a size of  $2^{l_0}$ . When it is filled up, the code length will be increased by 1; this is allowed to repeat until  $l = l_{max}$ .
- When  $l_{max}$  is reached and the dictionary is filled up, it needs to be flushed (as in Unix *compress*, or to have the LRU (least recently used) entries removed).

## 7.6 Arithmetic Coding

- Arithmetic coding is a more modern coding method that usually out-performs Huffman coding.
- Huffman coding assigns each symbol a codeword which has an integral bit length. Arithmetic coding can treat the whole message as one unit.
- A message is represented by a half-open interval  $[a, b)$  where  $a$  and  $b$  are real numbers between 0 and 1. Initially, the interval is  $[0, 1)$ . When the message becomes longer, the length of the interval shortens and the number of bits needed to represent the interval increases.

## Arithmetic Coding

- ◆ Encodes the **block of symbols** into a single number, a fraction  $n$  where  $(0.0 \leq n < 1.0)$ .
- ◆ **Step 1:** Divide interval  $[0,1)$  into subintervals based on probability of the symbols in the current context → **Dividing Model**.
- ◆ **Step 2:** Divide interval corresponds to the current symbol into subintervals based on dividing model of step 1.
- ◆ **Step 3:** Repeat Step 2 for all symbols in the block of symbols.
- ◆ **Step 4:** Encode the block of symbols with a single number in the final resulting range. Use the corresponding binary number in this range with the smallest number of bits.

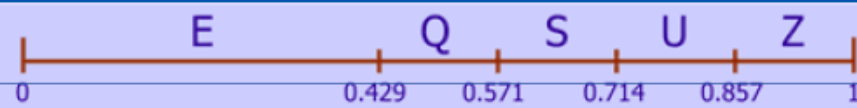
See the encoding and decoding examples in the following slides

# Arithmetic Coding, Encoding

## Example: SQUEEZE

Using FLC: 3 bits per symbol  $\rightarrow 7 \times 3 = 21$  bits

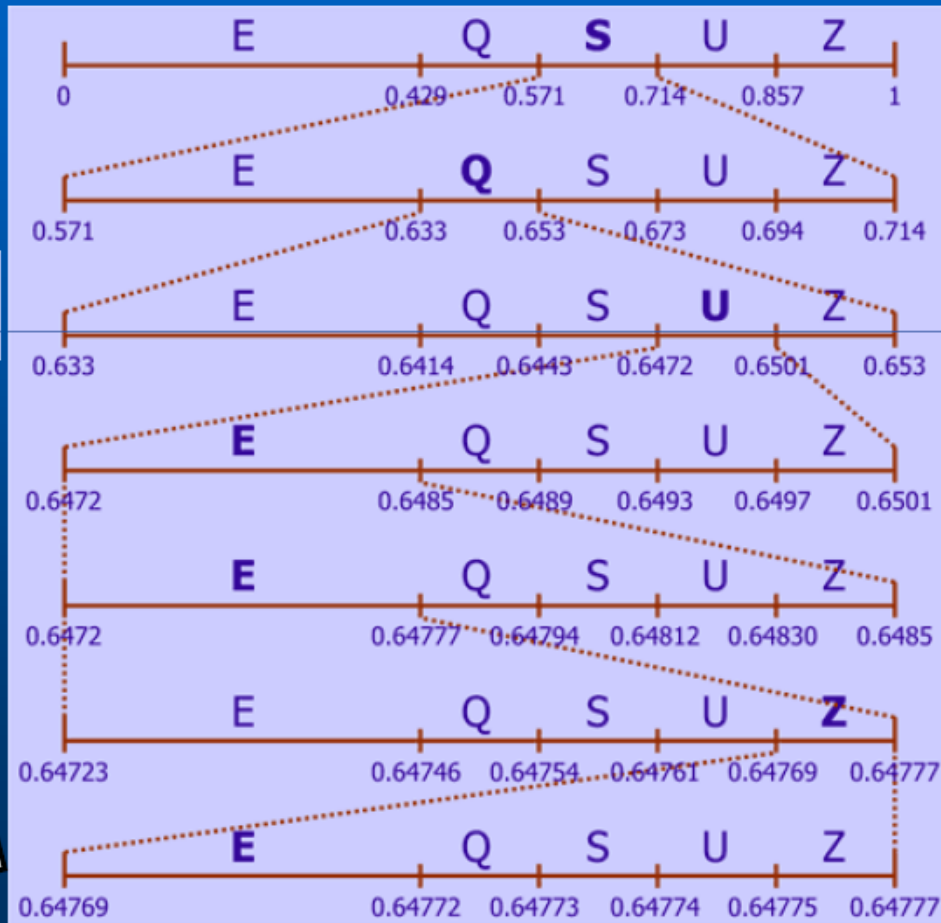
$P('E') = 3/7$  Prob. 'S' 'Q' 'U' 'Z':  $1/7$



Dividing Model

We can encode the word SQUEEZE with a single number in  $[0.64769-0.64772)$  range.

The binary number in this range with the smallest number of bits is 0.101001011101, which corresponds to 0.647705 decimal. The '0.' prefix does not have to be transmitted because every arithmetic coded message starts with this prefix. So we only need to transmit the sequence 101001011101, which is only 12 bits.



## ALGORITHM 7.5 Arithmetic Coding Encoder

```
BEGIN
    low = 0.0;    high = 1.0;    range = 1.0;

    while (symbol != terminator)
    {
        get (symbol);
        low = low + range * Range_low(symbol);
        high = low + range * Range_high(symbol);
        range = high - low;
    }

    output a code so that low <= code < high;
END
```

## Example: Encoding in Arithmetic Coding

| Symbol | Probability | Range        |
|--------|-------------|--------------|
| A      | 0.2         | [0, 0.2)     |
| B      | 0.1         | [0.2, 0.3)   |
| C      | 0.2         | [0.3, 0.5)   |
| D      | 0.05        | [0.5, 0.55)  |
| E      | 0.3         | [0.55, 0.85) |
| F      | 0.05        | [0.85, 0.9)  |
| \$     | 0.1         | [0.9, 1.0)   |

(a) Probability distribution of symbols.

Fig. 7.8: Arithmetic Coding: Encode Symbols "CAEE\$"



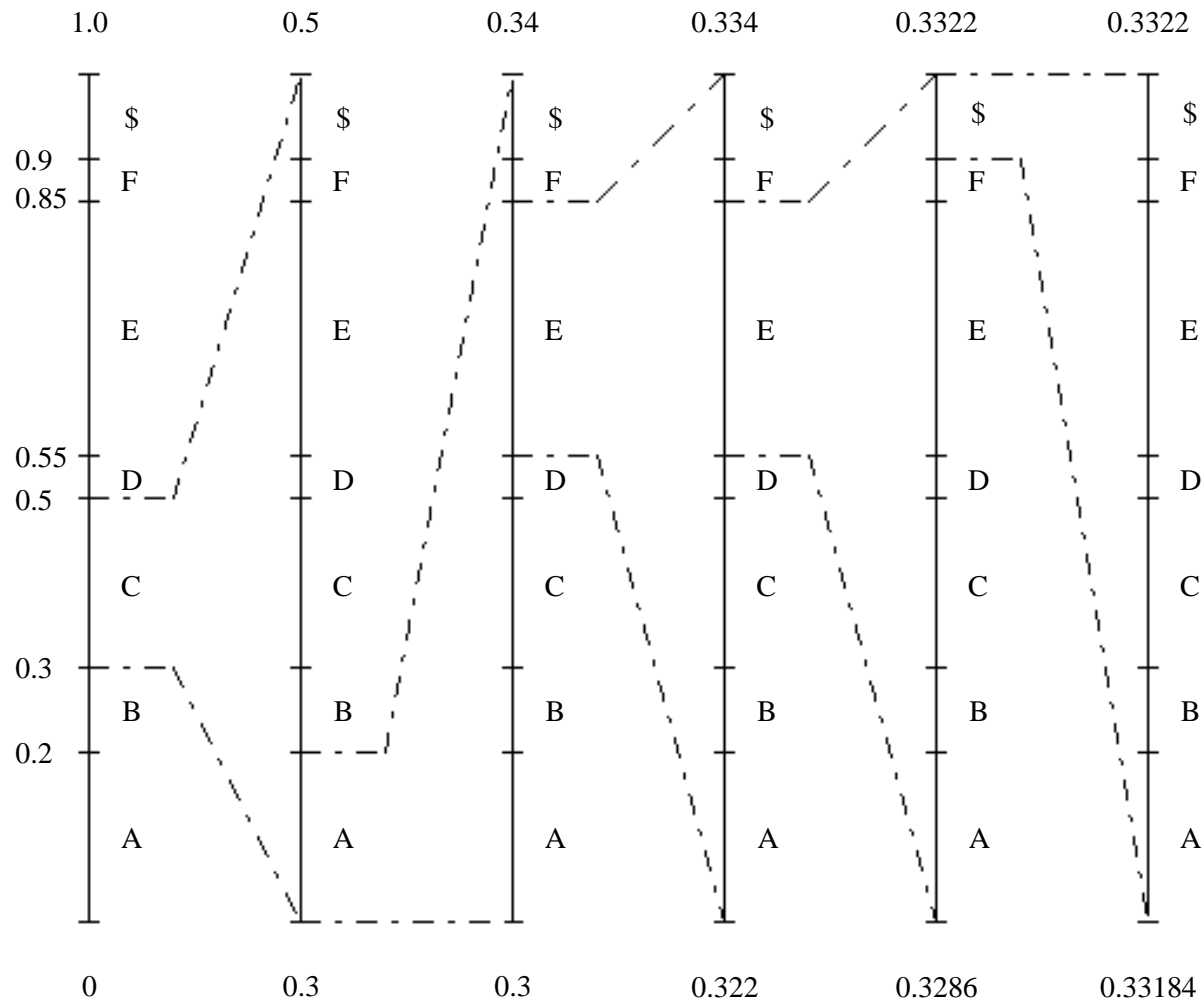


Fig. 7.8(b) Graphical display of shrinking ranges.

| Symbol | low     | high    | range   |
|--------|---------|---------|---------|
|        | 0       | 1.0     | 1.0     |
| C      | 0.3     | 0.5     | 0.2     |
| A      | 0.30    | 0.34    | 0.04    |
| E      | 0.322   | 0.334   | 0.012   |
| E      | 0.3286  | 0.3322  | 0.0036  |
| \$     | 0.33184 | 0.33220 | 0.00036 |

(c) New *low*, *high*, and *range* generated.

Fig. 7.8 (cont'd): Arithmetic Coding: Encode Symbols "CAEE\$"

## PROCEDURE 7.2 Generating Codeword 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
```

- The final step in Arithmetic encoding calls for the generation of a number that falls within the range [*low*, *high*). The above algorithm will ensure that the shortest binary codeword is found.

## ALGORITHM 7.6 Arithmetic Coding Decoder

BEGIN

  get binary code and convert to  
  decimal value = value(code);

  Do

    { find a symbol  $s$  so that  
       $\text{Range\_low}(s) \leq \text{value} < \text{Range\_high}(s)$ ;  
      output  $s$ ;  
       $\text{low} = \text{Rang\_low}(s)$ ;  
       $\text{high} = \text{Range\_high}(s)$ ;  
       $\text{range} = \text{high} - \text{low}$ ;  
       $\text{value} = [\text{value} - \text{low}] / \text{range}$ ;  
    }

  Until symbol  $s$  is a terminator

END

**Table 7.5 Arithmetic coding: decode symbols “CAEE\$”**

| value      | Output Symbol | low  | high | range |
|------------|---------------|------|------|-------|
| 0.33203125 | C             | 0.3  | 0.5  | 0.2   |
| 0.16015625 | A             | 0.0  | 0.2  | 0.2   |
| 0.80078125 | E             | 0.55 | 0.85 | 0.3   |
| 0.8359375  | E             | 0.55 | 0.85 | 0.3   |
| 0.953125   | \$            | 0.9  | 1.0  | 0.1   |

## 7.7 Lossless Image Compression

- **Approaches of Differential Coding of Images:**

- Given an original image  $I(x, y)$ , using a simple difference operator we can define a difference image  $d(x, y)$  as follows:

$$d(x, y) = I(x, y) - I(x - 1, y) \quad (7.9)$$

or use the discrete version of the 2-D Laplacian operator to define a difference image  $d(x, y)$  as

$$d(x, y) = 4I(x, y) - I(x, y - 1) - I(x, y + 1) - I(x + 1, y) - I(x - 1, y) \quad (7.10)$$

- Due to *spatial redundancy* existed in normal images  $I$ , the difference image  $d$  will have a narrower histogram and hence a smaller entropy, as shown in Fig. 7.9.

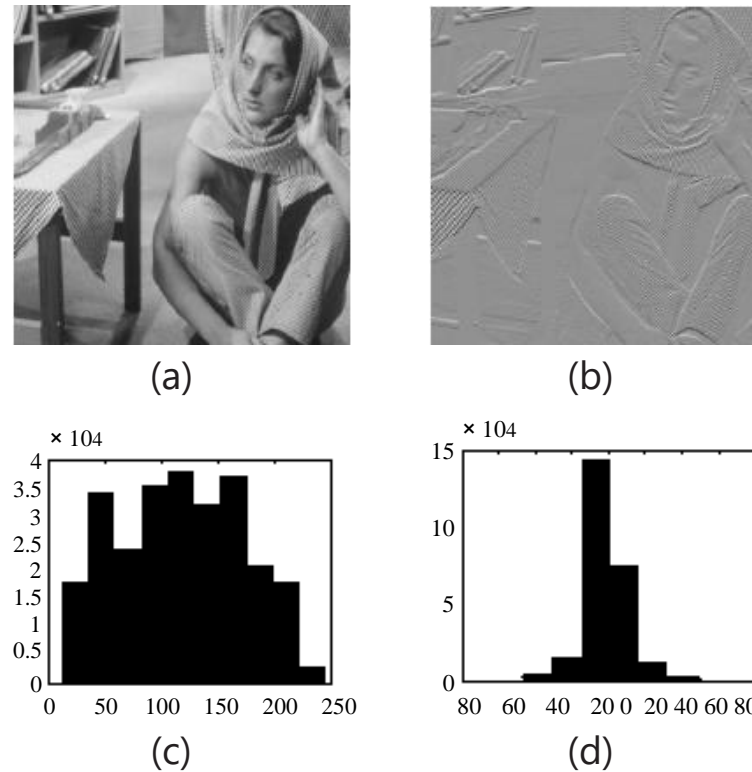


Fig. 7.9: Distributions for Original versus Derivative Images. (a,b): Original gray-level image and its partial derivative image; (c,d): Histograms for original and derivative images.

(This figure uses a commonly employed image called "Barb".)



## Lossless JPEG

- **Lossless JPEG:** A special case of the JPEG image compression.
- **The Predictive method**
  1. **Forming a differential prediction:** A predictor combines the values of up to three neighboring pixels as the predicted value for the current pixel, indicated by 'X' in Fig. 7.10. The predictor can use any one of the seven schemes listed in Table 7.6.
  2. **Encoding:** The encoder compares the prediction with the actual pixel value at the position 'X' and encodes the difference using one of the lossless compression techniques we have discussed, e.g., the Huffman coding scheme.

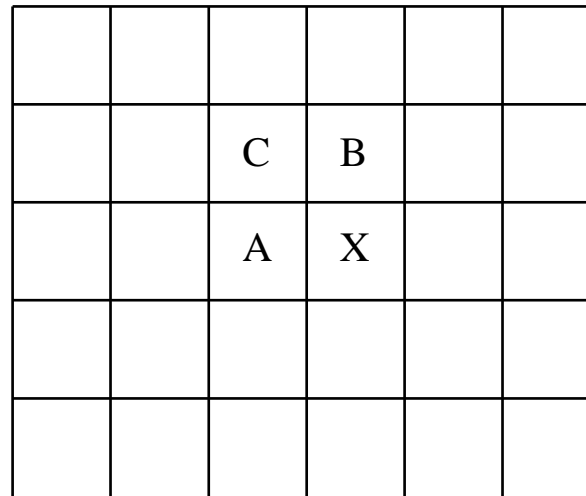


Fig. 7.10: Neighboring Pixels for Predictors in Lossless JPEG.

- **Note:** Any of A, B, or C has already been decoded before it is used in the predictor, on the decoder side of an encode-decode cycle.

**Table 7.6: Predictors for Lossless JPEG**

| Predictor | Prediction        |
|-----------|-------------------|
| P1        | A                 |
| P2        | B                 |
| P3        | C                 |
| P4        | $A + B - C$       |
| P5        | $A + (B - C) / 2$ |
| P6        | $B + (A - C) / 2$ |
| P7        | $(A + B) / 2$     |

**Table 7.7: Comparison with other lossless compression programs**

| Compression Program    | Compression Ratio |          |      |         |
|------------------------|-------------------|----------|------|---------|
|                        | Lena              | football | F-18 | flowers |
| Lossless JPEG          | 1.45              | 1.54     | 2.29 | 1.26    |
| Optimal lossless JPEG  | 1.49              | 1.67     | 2.71 | 1.33    |
| compress (LZW)         | 0.86              | 1.24     | 2.21 | 0.87    |
| gzip (LZ77)            | 1.08              | 1.36     | 3.10 | 1.05    |
| gzip -9 (optimal LZ77) | 1.08              | 1.36     | 3.13 | 1.05    |
| pack (Huffman coding)  | 1.02              | 1.12     | 1.19 | 1.00    |

## 7.8 Further Exploration

- **Text books:**

- *The Data Compression Book* by M. Nelson
- *Introduction to Data Compression* by K. Sayood

- **Web sites:** – [→ Link to Further Exploration for Chapter 7..](#) including:

- An excellent resource for data compression compiled by Mark Nelson.
- The Theory of Data Compression webpage.
- The FAQ for the comp.compression and comp.compression.research groups.
- A set of applets for lossless compression.
- A good introduction to Arithmetic coding
- Grayscale test images f-18.bmp, flowers.bmp, football.bmp, lena.bmp



# End of Chapter 7