

Basic Principles:

We now consider yet another popular lossless compression scheme, which is originally called Lempel-Ziv coding, and also referred to as Lempel-Ziv-Welch (LZW) coding, following the modifications of Welch for image compression.

The basic principles of this encoding scheme are:

- a) It assigns a fixed length codeword to a variable length of symbols.
- b) Unlike Huffman coding and arithmetic coding, this coding scheme does not require a priori knowledge of the probabilities of the source symbols.
- c) The coding is based on a “dictionary” or “codebook” containing the source symbols to be encoded. The coding starts with an initial dictionary, which is enlarged with the arrival of new symbol sequences.
- d) There is no need to transmit the dictionary from the encoder to the decoder. A Lempel-Ziv decoder builds an identical dictionary during the decoding process



Example Initialization of the dictionary: (1) *a*
(2) *b*
(3) *c*

Encode *baca caba baba caba*.

Read	Produce	Write
		(1) <i>a</i>
		(2) <i>b</i>
		(3) <i>c</i>
<i>b</i>		
<i>a</i>	(2)	(4) <i>ba</i>
<i>c</i>	(1)	(5) <i>ac</i>
<i>a</i>	(3)	(6) <i>ca</i>
<i>c</i>		
<i>a</i>	(5)	(7) <i>aca</i>
<i>b</i>	(1)	(8) <i>ab</i>
<i>a</i>		
<i>b</i>	(4)	(9) <i>bab</i>
<i>a</i>		
<i>b</i>		
<i>a</i>	(9)	(10) <i>baba</i>
<i>c</i>		
<i>a</i>		
<i>b</i>	(7)	(11) <i>acab</i>
<i>a</i>	(4)	

remove the last character

Look, once more, how the source stream is transformed into the code stream:

b a c ac a ba bab aca ba


(2) (1) (3) (5) (1) (4) (9) (7) (4)

1.2.2 The LZW Decoder

A First Approach

The principal goal of the decoder is the *reconstruction of the dictionary* of the encoder. It has to correctly interpret the stream of code words (pointers) that it receives. The down-to-earth **decoding** (the identification of the code words) is a part of this task.

The *current string* s , candidate for admission to the dictionary, will still remain in the centre of the algorithm. Let us begin immediately with an example.

Example Initialized dictionary: (1) a
(2) b
(3) c
Decode  (3)(1)(2)(5)(1)(4)(6)(6).

Read	Produce	Write
		(1) a
		(2) b
		(3) c
(3)	c	
(1)	a	(4) ca
(2)	b	(5) ab
(5)	ab	(6) ba
(1)	a	(7) aba
(4)	ca	(8) ac
(6)	ba	(9) cab
(6)	ba	(10) bab

take the first character only
not remove the last characters

Example:

EXAMPLE LZW Compression for String ABABBABCABABBA

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

code	string
1	A
2	B
3	C

Now if the input string is ABABBABCABABBA, the LZW compression algorithm works as follows:

output	code	string
	1	A
	2	B
	3	C
1	4	AB
2	5	BA
4	6	ABB
5	7	BAB
2	8	BC
3	9	CA
4	10	ABA
6	11	ABBA
1		

EXAMPLE LZW decompression for string ABABBABCABABBA

Input codes to the decoder are 124523461. The initial string table is identical to what is used by the encoder.

The LZW decompression algorithm then works as follows:

entry/output	code	string
	1	A
	2	B
	3	C

A		
B	4	AB
AB	5	BA
BA	6	ABB
B	7	BAB
C	8	BC
AB	9	CA
ABB	10	ABA
A	11	ABBA

Apparently the output string is ABABB ABC ABABB A — a truly lossless result!

▶ Principal of Lampel ZIV algorithm

The encoding in this algorithm is accomplished by parsing the source data stream into segments that are the shortest substances not encountered previously.

- ▶ To illustrate this principle let us consider the example of an input binary sequence specified as :

000101110010

- ▶ We assume that the binary symbols 0 and 1 are already stored in this order in the code book. Hence we write,

subsequences stored : 0, 1

Data to be parsed : 000101110010

- ▶ Now examine the data in above equation from LHS and find the shortest subsequence which is not encountered previously. It is 00. so we include 00 as the next entry in the subsequence and move 00 from data to subsequence as follow :

Subsequences stored : 0, 1, 00

Data to be parsed : 010110010

- ▶ The next shortest Subsequences which is not previously repeated is 01. In above equation Note that we are examining from LHS. Hence we write,

Subsequences stored : 0, 1, 00, 01

Data to be parsed : 01110010

- ▶ The next shortest Subsequences which is previously not encountered is 011. so we write,

Subsequences stored : 0, 1, 00, 01, 011

Data to be parsed : 10010

- ▶ Similarly we can continue until the data stream has been completely parsed. The code book of binary Subsequences gets ready as shown in figure

Numerical Position	1	2	3	4	5	6	7
Subsequences	0	1	00	01	001	10	010

Code book of Sequence

- ▶ The first row in the codebook shows the numerical position of various subsequence in the codebook.
- ▶ **Numerical representation :**
 - Let us now add third row to figure. This row is called as numerical representation as shown in figure

Numerical Position	1	2	3	4	5	6	7
Subsequences	0	1	00	01	011	10	010
Numerical representation			11	12	42	21	41

- ▶ The sequences 0 and 1 are originally stored. So consider the third Subsequences i.e. 00. this is the first Subsequences in the data stream and it is made up of concatenation of the first Subsequences i.e. 0 with itself.
- ▶ Hence it is represented by 11 in the row of numerical representation in above figure
- ▶ Similarly, subsequences 01 obtained by concatenation of first and second subsequences so we enter 12 below that.
- ▶ The remaining subsequences are treated accordingly.
- ▶ **Binary Encoded Representation :**
- ▶ The last (4th) row added as shown in figure, is the binary encoded representation of each subsequence.

Numerical Position	1	2	3	4	5	6	7
Subsequences	0	1	00	01	011	10	010
Numerical representation			11	12	42	21	41
Binary encoded blocks			0010	0011	1001	0100	1000

- ▶ The question is how to obtain binary encoded blocks.
- ▶ the last symbol of each subsequence in the second row of above figure (called as codebook) is called as an innovation symbol.
- ▶ So the last bit in each binary encoded block (4th row) is the innovation symbol of the corresponding subsequence,
- ▶ The remaining bits provide the equivalent binary representation of the “pointer” to the “root subsequence” that matches the one in question except for the innovation symbol.
- ▶ This can be explained as follow.
 1. Consider Numerical position 3 in figure. The binary encoded block is 0010.
 2. Consider Numerical position 5 in figure. It is partially reproduced below.

▶ Row 1: Numerical position 3 →

▶ Row 2: Subsequence →

0	0
---	---

 → Innovation number

This is the first subsequence

Take as it is

▶ Row 4: Binary encoded Block →

001	0
-----	---

Binary equivalent of 1 (this is called pointer)

▶ Row 1: Numerical position → 5

▶ Row 2: Subsequence →

01	1
----	---

 → Innovation number

This is the 4th subsequence

Take as it is

▶ Row 4: Binary encoded Block →

100	1
-----	---

Binary equivalent of 4. (this is called pointer)

- ▶ Consider the numerical position 6 in figure. It is partially reproduced below.

- ▶ Row 1: Numerical position → 6

- ▶ Row 2: Subsequence →

1	0
---	---

 → Innovation number

This is the 2nd subsequence

Take as it is

- ▶ Row 4: Binary encoded Block →

010	0
-----	---

Binary equivalent of 2.
(this is called pointer)

- ▶ Similarly the other entries in the fourth row are made.

RLE for text data

The process involves going through the text and counting the number of consecutive occurrences of each character (called "a run"). The number of occurrences of the character and the character itself are then stored in pairs.

Example:

aaaabbbbbbcdddd

There are 16 characters in the example so 16 bytes (assuming Extended ASCII is being used) are needed to store these characters in an uncompressed format:

Example as ASCII:

97 97 97 97 98 98 98 98 98 98 99 100 100 100 100 100

RLE can be used to store that same data using fewer bytes. There are four consecutive occurrences of the character 'a' followed by six consecutive occurrences of 'b', one 'c' and finally five consecutive occurrences of 'd'. This could be written as the following sequence of count-character pairs:

Run-length encoding for the above example:

(4, a) (6, b) (1, c) (5, d)

So, if the text aaaabbbbbbcdddd is compressed using RLE, storing each count in binary in a single byte, we would end up with:

✓ 04 97 06 98 01 99 05 100

As we can see, this compressed version only requires 8 bytes – a reduction from the original 16 bytes.

Text that contains characters with a high frequency of repetition can be compressed efficiently. Text with few repetitions, ie more random text, will not compress well and can even result in the compressed version using *more* storage space than the uncompressed version.

A possible solution to this is to use a special byte value that 'flags' when a run is going to occur. This does mean though that any run of bytes automatically has an additional byte added to it. When there is no flag, the next byte(s) are taken as their face value and with a run of 1.

Example

Uncompressed text

aaaaaaaaaabbbbbbececececececdccccccccccccccdec b (46 bytes)

RLE count-character pairs

(10, a) (6, b) (1, e) (1, c) (1, e) (1, c) (1, e) (1, c) (1, e)
(1, c) (1, e) (1, c) (1, e) (1, c) (15, d) (1, e) (1, c) (1, b)

No Flag RLE

10 97 06 98 01 101 01 99 01 101 01 99 01 101 01 99 01 101 01
99 01 101 01 99 01 101 01 99 15 100 01 101 01 99 01 98 (36 bytes)

Flag (value 255) RLE

255 10 97 255 06 98 101 99 101 99 101 99 101 99 101 99 101 99
255 15 100 101 99 98 (24 bytes)

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

000000000000000011111111000000011111111111

40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:

15 0s, then 7 1s, then 7 0s, then 11 1s.

1111011101111011 ← 16 bits (instead of 40)
15 7 7 11

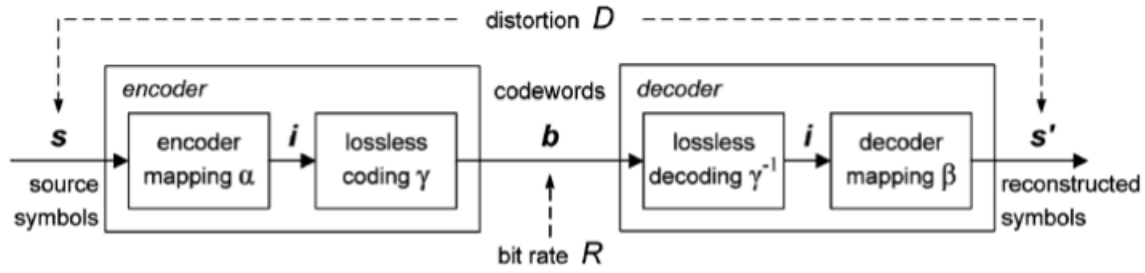


Fig. 4.1 Block diagram for a typical lossy source coding system.

4.1 The Operational Rate Distortion Function

A lossy source coding system as illustrated in Figure 4.1 consists of an encoder and a decoder. Given a sequence of source symbols s , the encoder generates a sequence of codewords b . The decoder converts the sequence of codewords b into a sequence of reconstructed symbols s' .

The encoder operation can be decomposed into an irreversible encoder mapping α , which maps a sequence of input samples s onto a sequence of indexes i , and a lossless mapping γ , which converts the sequence of indexes i into a sequence of codewords b . The encoder mapping α can represent any deterministic mapping that produces a sequence of indexes i of a countable alphabet. This includes the methods of scalar quantization, vector quantization, predictive coding, and transform coding, which will be discussed in the following sections. The lossless mapping γ can represent any lossless source coding technique, including the techniques that we discussed in Section 3. The decoder operation consists of a lossless mapping γ^{-1} , which represents the inverse of the lossless mapping γ and converts the sequence of codewords b into the sequence of indexes i , and a deterministic decoder mapping β , which maps the sequence of indexes i to a sequence of reconstructed symbols s' . A lossy source coding system Q is characterized by the mappings α , β , and γ . The triple $Q = (\alpha, \beta, \gamma)$ is also referred to as *source code* or simply as *code* throughout this monograph.