TCOM 370 NOTES 99-10

DATA COMPRESSION

1. Introduction

Consider a message sequence of binary digits, or more generally a sequence of message symbols from a finite-size alphabet (e.g. English text), which is to be transmitted (or stored). The more general sequence of symbols may be converted into a bit sequence by a simple symbol-to-character mapping such as provided by the ASCII code.

- Often we can find a ***more efficient representation*** of the sequence, which requires fewer bits to specify it than contained in the original sequence. This is because there is often much ***redundancy*** in message sequences in typical situations. We can seek ways of ***compressing*** the message into a shorter sequence ***without losing any information,*** that is in a reversible or ***loss-less way,*** so that the original sequence can be reconstructed upon receipt of its compressed version.

For example, in typewritten English text, individual letters may be mis-typed or simply dropped at a fairly high rate and the text usually remains quite meaningful. This is because there is a lot of redundant information in much of English text. In fact most of the vowels can be dropped and it is still possible to make out many of the original words within the context of the message. As a simple example of a sequence of symbols that can be compressed without any loss of information, the text string *"aaa...a"* in which there are 1000 *a*'s may be equivalently stated as "sequence of one thousand letter *a*". This requires 33 characters (including spaces) as opposed to 1000. This type of compression can be made even more efficient, and is an example of ***run-length coding*** in which repeated characters are replaced by an indication of the character and a repetition count. The use of the scientific notation 10^6 is another simple instance of this.

What makes compression valuable is that we can use it to transmit an original message at a higher ***effective*** rate than is provided by a physical link.

Note: The term "coding" is used to describe schemes for error control and *also* to describe schemes to compress data. Do not get confused! The context makes it very clear what type of code is being referred to. Error control coding works by **adding** redundancy in special ways (e.g. repetition codes) so that transmission errors do not necessarily destroy the integrity of a message. In coding for compression we **remove** redundancy thus making the message "leaner (and meaner)". As you can expect, if compressed messages then suffer from any errors in transmission, they may become completely meaningless because there is no redundancy in them to help recovery from error. In fact the common procedure in many transmission schemes is to first remove redundancy from (compress) the message stream, and then add redundancy in a controlled and specially designed way to provide best protection against transmission errors for the specific type of channel that is being used. The original redundancy in the message is often not in a form that is best suited for error control in transmission channels.

2. Lossy Compression - A Digression

It is generally possible to achieve even more impressive compression if we are willing to lose some information permanently, that is if the compression is ***non-reversible***. Many speech and video compression schemes are ***"lossy"*** or non-reversible and the compressed information does not have the fidelity of the original. Nonetheless, some loss of information may be quite acceptable if the corresponding compression factor is very significant.

For example, instead of using 8-bit PCM for **telephone speech digitization** an alternative standard specifies differential PCM or **DPCM** that results in half the bit rate [32 Kbps] or less, instead of the 64 Kbps PCM rate, with an almost imperceptible loss of speech quality. In DPCM we transmit a quantized version of the ***difference*** between the current speech sample and the immediately previous reconstructed speech sample. The scheme starts with the first sample being quantized with the full 8-bit resolution of standard PCM. The first quantized sample is then subtracted from the next incoming sample and the difference is quantized with a 4-bit resolution. Because adjacent speech samples are correlated with each other, the differences tend to have a dynamic range that is smaller than the range of any one sample (differences tend to be smaller), and therefore a 4-bit quantization gives adequate representation of the differences. To reconstruct, we start with the first sample and then add to it the difference to get the second sample

reconstruction, etc. Only the differences (after the first sample) are transmitted, thereby achieving the compression.

A similar principle can be used to transmit **video sequences.** A video source will typically produces 30 frames (still images) per second. Each frame is sampled to get a digital representation of the frame. A sample is called a picture element (pixel), and there could easily be 800x600 pixels per frame. Each pixel of a color image generally requires 8 bits for each of the three colors (R, G, B). This works out to a very high bit rate of 30x800x600x24 =345.6 Mbps! Even for grayscale (intensity only, no color) video we need more than 100 Mbps for transmission. Differential encoding can be very useful here also. Successive frames of the video are often very similar to each other, and pixel differences for pixels in corresponding positions in two frames can be encoded instead of pixel values. Furthermore, the effect of motion is generally to displace a block of pixels in one frame to a different location in the next frame. Video coders can make use of this by taking blocks (say of size 16x16) of each frame and comparing them to blocks in the previous frame to see if and where there is a close match. Upon finding a good match the displacement between the matching blocks is encoded, together with the differences between corresponding pixels in the current and matching previous block. This type of scheme (including other refinements not discussed here) can dramatically reduce the bit rate with small loss of quality. For example, a compression factor of 10 may be achieved easily with very good picture quality.

In the rest of these notes we will focus on loss-less compression.

3. Simple Loss-less Compression Schemes

These are of an *ad hoc* nature. Some examples:

(a) Packed Decimal or Half-Byte Compression for Numeric Data.

Suppose we are sending only numeric data, for which we need the characters 0-9, a character for the decimal point, and a delimiter character to separate individual numerical records. Instead of using ASCII 7-bit characters for this, consider using only that subset of the ASCII characters that start with the sequence "011". These include the symbols 0-9 and other symbols including the ":" and ";" characters. These latter two can be used for the decimal point and the delimiter, respectively. Since all characters used start with 011, this prefix can be dropped and only four

bits are sent for each symbol needed to transmit the numerical data. This achieves a *compression factor* of 7/4.

(b) Character Suppression and Run Length Coding

In character oriented transmission, a string of identical characters occurring in the message can be replaced by an indication of the character value and a second character containing its repeat count (binary encoded integer), preceded by a reserved control character to indicate that character suppression is being used at that point in the message. This is a version of run-length coding. It requires three bytes to encode a string of repeated characters (and should be used for a string of three or more identical characters!). The old KERMIT personal computer file transfer protocol uses this scheme for compression.

4. Information and Entropy

Important ideas in this section:

- *Quantitative definition of information*
- *entropy and its computation*
- *entropy as a bound for source compression*
- *the notion of a variable length code for compression, prefix condition codes, Huffman coding*
- *the possibility of significant compression for a source with non-equal symbol probabilities*

Consider a source putting out a sequence $\{X_i , i=1, 2, ... \}$ of symbols from its alphabet set. In general, the source may have an alphabet of N symbols {**s**1, **s**2, ..., **s**N}. It emits some random sequence of these symbols as far as the destination is concerned. The output X_i at time i is one of the symbols in the source alphabet. A binary source has the symbol alphabet {0, 1}. The purpose of a communication system is to transmit the source sequence faithfully to a remote destination.

- A fundamental question may be asked -- how much "**information**" is the source putting out each time it produces a symbol?

To see that this is not a vacuous question, consider the binary source and suppose it produces

symbol **s**1="0" with probability p

and

symbol **s**2="1" with probability 1-p.

Consider the special case in which p is very close to 1, say p=0.99. This means that with high probability the source just produces symbol "0". If the receiver knew this probability distribution of symbols, the source would be producing very little useful information as far as the receiver is concerned! This is because even if the source symbols were never transmitted to the receiver, the receiver would be able to reconstruct the source symbols to a high degree of accuracy by simply always assuming that the source symbol was "0". If the source was picking symbols with some more uniform probability distribution (for example, equal probability for each symbol in the source alphabet), the receiver would be getting more "information" each time a symbol got transmitted to it. This is because without the transmission the receiver would always be fairly uncertain as to the value of the symbol, and the transmission ***resolves this uncertainty*** by virtue of the information it has conveyed.

Information conveyed by a source may therefore be quantified as ***a degree of uncertainty*** that one has about the outputs from a source before they are observed. How do we quantify uncertainty? We realize from the above that it has to do with the probability distribution of symbols. Consider a particular symbol **s** that gets produced with probability p. If p is small, the occurrence of **s** is very "informative"; we would not have expected that **s** would occur, so its occurrence conveys a lot of information. If p=1 the occurrence of **s** conveys no information since we already expect it to be observed.

We want therefore a measure of information conveyed that will vary in inverse relation to the probability p of occurrence. A measure that varies inversely as the probability p is $\log(1/p) = -\log(p)$. This quantity is always non-negative and decreases from a value of $+\infty$ at p=0, to 0 at p=1. We choose this definition for the information associated with the occurrence of a particular output **s** which has a probability p for the following reason. *We would like to have the property that a sequence of two symbols produced independently of each other from the same source should have an amount of information that is the sum of the individual information for each of the two symbols.* Since the probability of the sequence of two symbols is the product of the individual probabilities (product rule for joint occurrence of independent events), and the log of a product is the sum of the logs, we have the desired property.

- **The information associated with occurrence of a symbol "s" with *a priori* probability "p" of occurring is $-\log_2(p)$**

This leads to the following result for the **entropy, or average amount of information per symbol, H,** for a source of alphabet size N for which the individual symbols are produced with probabilities p_n, n=1, 2, ..., N :

$$\mathbf{H} = -\sum_{n=1}^{N} p_n \log_2(p_n)$$

The **entropy** is simply the **expected value of the information** $-\log_2(p)$ associated with each output from the source. Notice that we have chosen to use the base 2 in the logarithm. With this choice of the log base, the **units of H are "bits per symbol".** The term "bit" usually stands for *"**bi**nary dig**it**"*, that is a "1" or a "0". Here the unit of entropy is defined as the "bit" when we use log base 2 in the definition; this is a unit of "information". It turns out that there is a close relationship between the entropy of a source in bits per source symbol and the minimum number of binary digits we need to represent each source symbol.

Numerical Example:

(a) Consider a **binary source** with $p_1 = p = 1/8$ and $p_2 = 1-p = 7/8$. Then entropy H for this source is, using the fact that $\log_2(a/b)=\log_2(a) - \log_2(b)$ and $\log_2 8 = 3$,

$H = -(1/8)\log_2(1/8) - (7/8)\log_2(7/8) = (3/8) + (21/8) - (7/8)\log_2(7) = 0.54$ bits/symbol.

If the source had $p=1-p=\frac{1}{2}$ (**equally likely symbols**) then

$H = -(1/2)\log_2(1/2) - (1/2)\log_2(1/2) = 1$ bits/symbol

(b) More generally, for a **source alphabet of size N and equally likely symbols** we will have **$H=\log_2 N$ bits/symbol.**

- We can also show that **H ≥ 0 always** and that the **maximum value for H** for a source with symbol alphabet of N symbols is **$\log_2 N$ bits/symbol**

H is never negative (reasonable, given it is the expected value of -log(p), non-negative "information") and the largest average information is associated with a source with equally likely symbols. This is in agreement with our intuition, since for such a source it is most difficult to predict (we are most uncertain about) the value of the symbol it will emit.

5. Statistical Encoding for Compression

There is a fundamental set of results on compression of source sequences, and we will state a result in simple form here, without proof, that indicates the central importance of the concept of **entropy.**

Before we proceed note that if we have a source with symbols from an alphabet of size N and we do not utilize any statistical information about the source, then we would need a number of bits (binary digits) to represent it that is equal to the least integer upper bounding $\log_2 N$. For example, suppose we have a source producing one of N=8 symbols each time. We could use a 3-bit code, since $2^3=8$ and $\log_2 8 =3$. If N=10 we would need a 4-bit code, since $\log_2 10 =3.32$ (We can produce $2^4=16$ combinations of 4 binary digits, hence we can assign a unique 4-bit sequence to each of the 10 symbols.)

A Source Compression Theorem:

> "Consider a source producing **independent symbols from a finite alphabet** according to a specific probability distribution (e.g. a binary source producing a sequence of independent, identically distributed, binary digits). Let the **source entropy be H bits/symbol.** Then it is always possible to **compress** the source **without losing information** and get arbitrarily close to the **minimum possible average transmission rate of H binary digits (bits) per source symbol.**"

This is a celebrated result of information/communication theory.

Note that H binary digits per source symbol becomes the lowest achievable compression for such a source! Since H can be much smaller than $\log_2(N)$ for a

source of alphabet size N, for example, this suggests the possibility of significant compression.

It turns out that if the source produces symbols that are statistically *dependent* than it is possible to achieve even lower rates. On the other hand, for independent symbols it is not possible to compress to less than H bits per source symbol on the average.

Example of Statistical Compression:

Consider a source producing independent symbols, from the alphabet {A, B, C, D}, each time with the following probabilities:

A with probability $\frac{3}{4}$

B with probability $\frac{1}{8}$

C with probability $\frac{1}{16}$

D with probability $\frac{1}{16}$

The source entropy is

$$-(3/4)\log_2(3/4) - (1/8)\log_2(1/8) - (1/16)\log_2(1/16) - (1/16)\log_2(1/16)$$

$$= 1.186 \text{ bits/symbol}$$

For 4 symbols we can use a simple two-bit code (00, 01, 10, and 11) and this would require 2 bits/symbol to represent the source outputs. How can we transmit the source using less than 2 bits/symbol on the average?

Notice that if the letter A is encoded using n_A bits, the letter B using n_B bits, etc., then the average number of bits per letter is $\frac{3}{4}\ n_A + \frac{1}{8}\ n_B + \frac{1}{16}\ n_C + \frac{1}{16}\ n_D$. The simple code above used 2 bits for each letter. Suppose we could find a code that uses ***fewer bits for the more probable symbols,*** that is make n_A smaller than 2 and n_C and n_D larger than 2. Then we could expect a smaller average number of bits per symbol. In fact, consider the following code:

A	sent as	0	(1 bit)
B	sent as	10	(2 bits)
C	sent as	110	(3 bits)
D	sent as	111	(3 bits)

This is a **variable-length code,** assigning codewords of different lengths to different symbols. Now the average length of the codeword, that is **the average number of bits per symbol, is**

$\frac{3}{4}\ 1 + \frac{1}{8}\ 2 + \frac{1}{16}\ 3 + \frac{1}{16}\ 3 =$ **1.375** Compare this to the entropy H which is 1.186 bits/symbol. According to the theorem above we could do even better than this! We will see later how.

We also notice that it was the **non-uniform** nature of the source symbol probability distribution that worked in our favor for this example. If the source symbols had been equally likely, produced with probability $\frac{1}{4}$ each, then H would have been 2 bits/symbol and the simple 2-bit code would have been the best possible. Thus we have the potential of obtaining significant savings for sources producing symbols that are not equally likely.

One might question how we would **decode** the bit stream produced by the variable length code that we defined in this example. After all, since the codewords do not have fixed sizes, how do we know the codeword boundaries in a transmitted bit stream? If you think about the particular code we used to illustrate the idea of variable length coding, you notice that ***no codeword was the prefix*** of any other codeword! Thus the codeword boundaries in the stream of transmitted bits can be obtained simply by looking for the first complete codeword that can be formed following start of transmission or after the end of the last complete codeword identified. The code defined satisfied the ***"prefix condition"*** that no codeword be the prefix of any other valid codeword.

6. Huffman Coding

This is a **variable length coding** technique that creates a **prefix-condition** code given the non-uniform probability distribution of any finite alphabet. It provides a **systematic approach** to designing a variable length code which is **best** for a given finite-alphabet source (that is, a given probability distribution for its symbols). The example above is one instance of a code designed by this general technique. The sense in which this procedure results in a "best" code is that ***no other code can have a lower average number of bits per source symbol when individual source symbols are encoded one by one as they are produced.*** (It is possible to do better by processing source symbols in ***blocks*** of 2 or more consecutive outputs form the source, as we shall see)

The mechanics of producing the Huffman code for a given probability distribution are quite simple. To illustrate, we will consider an example.

Example of Huffman Coding:

Suppose we have a source with alphabet size N=7. Let the source symbols be from the alphabet {A, B, C, D, E, F, G} with respective probabilities {0.3, 0.2, 0.15, 0.15, 0.08, 0.07, 0.05}. Trace the steps below in the figure on the next page.

Step 1

List the source symbols in **decreasing** probability order. (The symbols A through G for the example are already labeled in decreasing probability order).

Step 2

Combine the two least-probability symbols into a single entity (call it "fg" in Stage 2 for the example), associating with it the **sum of the probabilities** of the individual symbols in the combination. Assign to each of the symbols that were combined one of the binary digits, "0" for upper symbol , "1" for the lower one (F →"0", G →"1" in the example).

Step 3

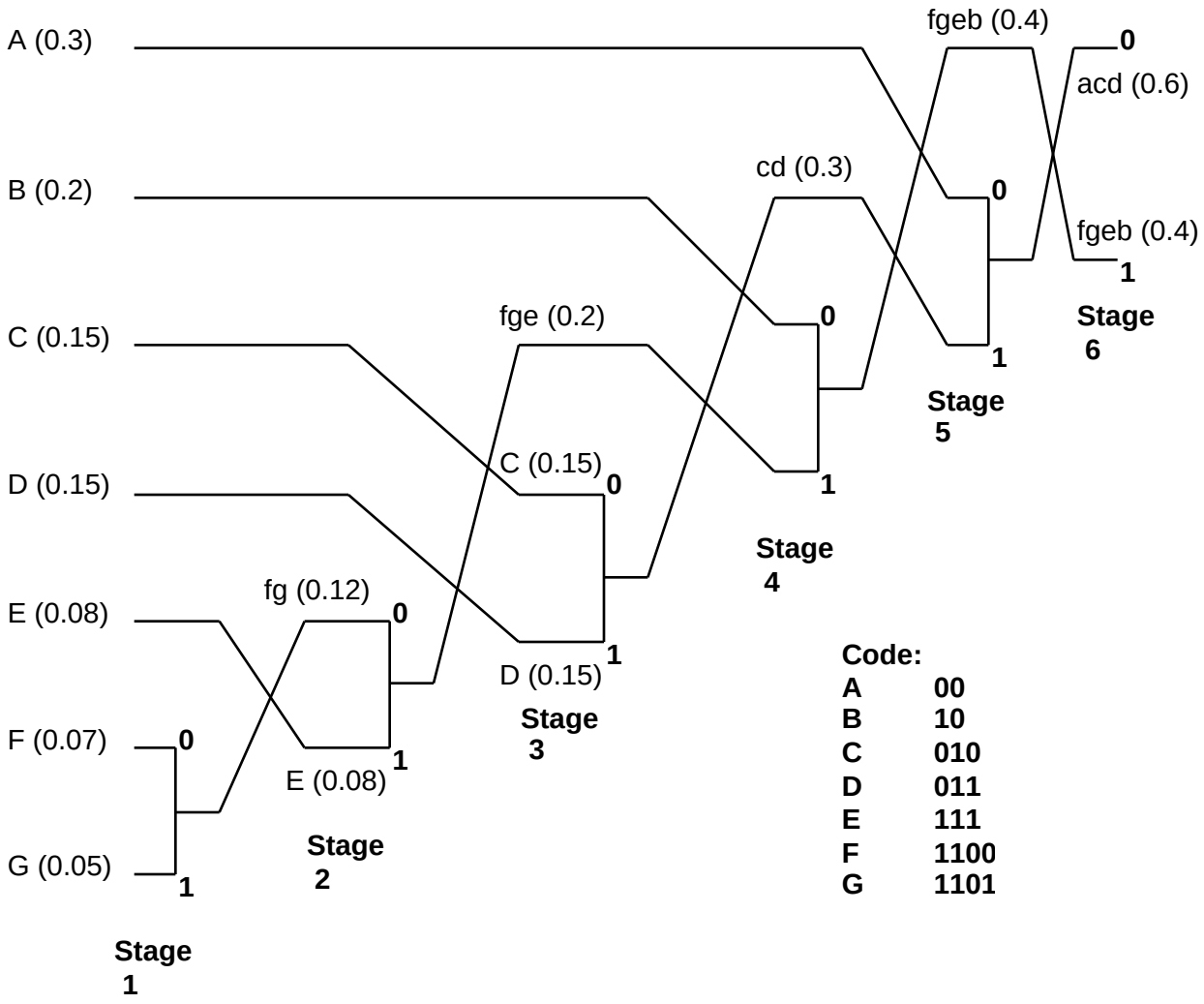
Re-order, if necessary, the resulting symbol list in decreasing probability order, treating any combined symbols from the previous list as a single symbol. If the list now has only 2 symbols, assign to each of them one of the binary digits (upper symbol →0, lower one →1) and go to Step 4. Otherwise go to Step 2.

Step 4

Read off the binary codewords for the source symbols, from right to left.

(Note: Assignment of code letters "0" and "1" to upper and lower symbols may be done in any way as long as it is consistent throughout. For example, text uses "1" for upper and "0" for lower. If this was done for the example below, all the codewords would have "1" and "0" interchanged.)

It may now be verified that the code for the 4-symbol alphabet of the example in the previous section is a Huffman code. The Huffman code is also always a prefix-condition code, by construction.



Huffman Code Construction for Example Source

Note: It is not always necessary to label combined symbols as we have done above. Probabilty values in parantheses help keep sorted order, however.

Extension of Huffman Coding

Suppose in the example of the 4-symbol source alphabet of **Section 5** the source produces statistically independent symbols according to the given probability distribution
$\{ \frac{3}{4} , \frac{1}{8} , \frac{1}{16} , \frac{1}{16} \}$. If we look at blocks of 2 symbols at a time, we get 16 combinations {AA, AB, AC, AD, BA, ..., DD} and the respective probabilities are the products of the individual symbol probabilities of the symbols making up the 2-symbol blocks. We can apply the Huffman code construction to this extended source alphabet, and work out codewords for each of the 16 pairs of symbols. The average codeword length for the 16-pair extended source may then be found, and dividing this result by 2 gives the average number of binary digits for each original source symbol. It turns out that this procedure gives better performance, i.e. a lower average number of bits per source symbol. The ultimate limit here is still the entropy rate of H bits per symbol.

(Carry out this procedure and verify this statement)

In general, we can get better performance by "blocking" symbols to define extended sources and using the Huffman code construction for the extended source. The complexity grows quickly, however. For example, for the English alphabet and some punctuation characters we might have 36 symbols; for a 2-symbol extended source the alphabet size becomes 36^2 which is more than 1000. On the other hand, the extended source may give very significant further savings especially when the source produces correlated symbols to begin with.

Dynamic Huffman Coding

For Huffman coding, we need to know the probabilities for the individual symbols (and we also need to know the joint probabilities for blocks of symbols in extended Huffman coding). If the probability distributions are not known for a file of characters to be transmitted, they have to be estimated first and the code itself has to be included in the transmission of the coded file. In dynamic Huffman coding a particular probability (frequency of symbols) distribution is assumed at the transmitter and receiver and hence a Huffman code is available to start with. As source symbols come in to be coded the relative frequency of the different symbols is updated at both the transmitter and the receiver, and corresponding to this the

code itself is updated. In this manner the code continuously adapts to the nature of the source distribution, which may change as time progresses and different files are being transferred.

Dynamic Huffman coding is the basis of data compression algorithms used in V series modems for transmission over the PSTN. In particular the MNP (Microcom Networking Protocol) Class 5 protocol commonly found in modems such as the V.32*bis* modem uses both **CRC error detection and dynamic Huffman coding for data compression.**

A modified form of coding together with run-length coding is used in compression for facsimile transmission.

.....

7. The LZW Algorithm

The Lempel-Ziv-Welch algorithm is a scheme for lossless compression of character or byte sequences. It is a very popular technique for compression, and is used in compression of text files for storage in computer systems (e.g. the Unix COMPRESS and GZIP utilities) and also in transmission of data via modems.

The basic idea of the whole class of such algorithms, of which LZW is one particular version, is the following. In sending or storing a string of characters, character patterns that have occurred earlier on in the string need not be sent, only a code referring to the pattern needs to be used. For example, you could create a dictionary of patterns that you encounter as you see new characters of the string, and assign a codeword for each new pattern, or perhaps only to each frequently used pattern. As an example, we could decide that from this point on in this set of notes the word “compression” will be designated by a special codeword say !c, and the word "codeword" will be designated as ?c The procedure for building the code dictionary can be designed such that the receiver (decoder or decompressor) can also construct the same dictionary that the transmitter (encoder or compressor) is constructing as characters come in, so that it is able to recover the original string. This is best illustrated by an example of the operation of the LZW coder/decoder.

This section is for your reference only. One good source for this type of material is the book by K. Sayood, An Intro. to Data Compression (Morgan Kauffman, 1996)

LZW Coding

Assume an alphabet { _ a b o d }of size 5 from which text characters are picked. Suppose the following string of characters is to be transmitted (or stored):

d a b b a _ d a b b a _ d a b b a _ d o o _ d o o_ .....

The coder starts with an initial dictionary consisting of the 5 individual letters of the alphabet and an index assignment, as in the following table:

ENTRY	**INDEX**
_	1
a	2
b	3
o	4
d	5

The first symbol that the encoder gets is the letter **"d"**. This is in the table, so it looks at the next symbol, which is letter **"a"**. The sequence **"da"** is not in the table, so the code for the first symbol **"d"** is sent (Index **5** is sent) and the sequence **"da"** is added to the table with index = **6**. The symbol waiting to be sent is **"a"**

The coder then looks at the next (third) symbol, which is a **"b"**. The sequence **"ab"** is not in the table, so **"a"** is sent as index **2** and **"ab"** is added to the table as index **7**. The symbol waiting to be sent is **"b"**

The next symbol is **"b"**. Sequence **"bb"** is not in the table. Letter **"b"** is sent as index **3**, and **"bb"** is added to the table as index **8**. The symbol waiting to be sent is **"b"**

The next symbol is **"a"**. Sequence **"ba"** is not in the table. Letter **"b"** is sent as index **3**, and **"ba"** is added to the table as index **9**. The symbol waiting to be sent is **"a"**

The next symbol is **"_"**. Sequence **"a_"** is not in the table. Letter **"a"** is sent as index **2**, and **"a_"** is added to the table as index **10**. The symbol waiting to be sent is **"_"**

The next symbol is **"d"**. Sequence **"_d"** is not in the table. Letter **"_"** is sent as index **1**, and **"_d"** is added to the table as index **11**. The symbol waiting to be sent is **"d"**

The next symbol is **"a"**. Sequence **"da"** is in the table. The next symbol is **"b"**. Sequence **"dab"** is not in the table. Sequence **"da"** is sent as index **6**, and **"dab"** is added to the table as index **12**. The symbol waiting to be sent is **"b"**

The next symbol is **"b"**. Sequence **"bb"** is in the table. The next symbol is **"a"**. Sequence **"bba"** is not in the table. **"bb"** is sent as index **8**, and **"bba"** is added to the table as index **13**. The symbol waiting to be sent is **"a"**

The next symbol is **"_".** Sequence **"a_"** is in the table. The next symbol is **"d"**. The sequence **"a_d"** is not in the table. Sequence **"a_"** is sent as index **10**, and **"a_d"** is added to the table as index **14**. The symbol waiting to be sent is **"d"**

The transmitted indices so far are: 5 2 3 3 2 1 6 8 10

The dictionary table at this point looks like this:

ENTRY	INDEX
_	1
a	2
b	3
o	4
d	5
da	6
ab	7
bb	8
ba	9
a_	10
_d	11
dab	12
bba	13
a_d	14

Continuing this way, the transmitter now looks at the next symbols **"abb"** beyond the **"d"** waiting to be sent. It sends out **"dab"** as index **12**, and adds **"dabb"** to the table as entry **15**. The symbol **"b"** is now waiting. The symbols **"a_"** come in, and **"ba"** is sent as **9**, with **"ba_"** added to the table as **16**. The symbol **"_"** is now waiting.

The next symbols **"do"** cause the transmitter to send **"_d"** as **11** and add **"_do"** as index **17** in the table. The next incoming symbol **"o"** causes **"oo"** to be added to the table as **18** and index **4** to be sent out. Symbol **"o"** is waiting.

In response to the next symbols **"_"** and **"doo"** the transmitter adds **"o_"** and **"_doo"** to the table, and sends index codes **4** and **17.** The rest of the procedure depends on the other incoming symbols.

The added table items at this point are:

ENTRY	INDEX
dabb	15
ba_	16
_do	17
oo	18

The transmitted indices to this point are:

5 2 3 3 2 1 6 8 10 12 9 11 4 4 17

Notice, as the table builds up longer sequences appear in the dictionary indicating that compression performance will get better as the coder builds up more information about the nature of the source, i.e. what type of strings it produces.

In practice of course the index values are transmitted as binary words. For example, fixed length 16-bit words may be used to represent the index values as binary coded integers. (With 16 bits, a table of size approx. 64,000 can be built)

LZW Decoding

Consider the decoder operation when the above index string is received. The decoder starts off with the same 5-index single-letter table as does the coder. The initial indices 5 2 3 3 2 1 allow the decoder to obtain the first few symbols (dabba_), and at the same time as the decoder decodes these symbols it expands its table exactly as the coder builds up its table. Thus when the next index 6 comes in the decoder knows that this represents the string "da"

Notice that the coder adds to the dictionary when it sees a symbol that would create a new sequence in the dictionary.

(Special case handling: Can the decoder receive an index code for a sequence not yet in its table? What can happen is that the decoder gets an index for a sequence it has not yet completed for addition to the table. This is because the coder looks one symbol ahead for the current transmission and update to its table, but the decoder does not have current access to the next symbol. Suppose the transmitted sequence was simply "_a_a_a_a_a_a....." Then the coder output will be 1 2 6 8 7 10 ... and the coder table would build up as:

ENTRY	**INDEX**
_	1
a	2
b	3
o	4
d	5
_a	6
a_	7
_a_	8
_a_a	9
a_a	10

At the decoder, the received 1 2 6 will result in the table being built up to index 7, with a partial sequence "_a" waiting to be completed as an entry for index 8. But at this point the index 8 is received, and we do not yet know what it represents! In this case the procedure is simply to use the partial sequence for 8, which is "_a", and note that the waiting "_a" makes a new entry with the first "_" of the partial result for 8. Thus 8 is decoded as "_a_". At this point the "_a_" remains as the beginning of the next index entry (9).

When 7 comes in next, it can be decoded and the sequence "_a_a" is entry 9. At this point "a_" is the sequence waiting to be completed as 10, when 10 comes in. This makes entry 10 "a_a", etc.

One way to avoid this special case is to not allow use of the very last table index, but require transmission of codewords for the constituents of the last entry. This avoids the use at the coder of an index that was added in its immediately previous step, since the decoder does not yet have access to the symbol used in its determination.)

Unix COMPRESS:

This uses an initial dictionary table of size 512 (9 bit codes). When this is filled, the next 512 dictionary indices are encoded as 10 bit words, etc. This way the shorter words are used for the initial shorter dictionary sequences. A maximum of 9 to 16 bits are specified by the user for the codeword size (default 16). When the maximum table size has been reached, the table remains fixed and the scheme becomes a static table-look-up coding scheme. The algorithm now monitors the actual compression ratio attained, and if it falls below a limit the table building is re-started. This allows the scheme to track the nature of the text in the file being compressed.

Modem Compression

The V.42bis modem compression standard uses a form of LZW coding. Some of the particular characteristics include:

- Periodic testing of compression factor; algorithm can switch between compression and transparent mode, depending on compression performance.
- Variable dictionary size, initial size negotiated at start-up. Recommended 2048, minimum 512.
- Special codewords for “enter transparent mode” and “increment dictionary index size by one bit”
- When dictionary size reaches maximum, strings not encountered recently are pruned.
- Maximum string length is 6 to 250, default 6 (Maximum length is imposed to control errors, since for long strings an error in its codeword causes loss of many characters).
- Decoding special case is avoided by not allowing use of latest entry in dictionary at any step.