UNIT-III

Coding:

Coding a Sequence : In order to distinguish a sequence of symbols from another sequence of symbols we need to tag it with a unique identifier. One possible set of tags for representing sequences of symbols are the numbers in the unit interval [0,1). Because the number of numbers in the unit interval is infinite, it should be possible to assign a unique tag to each distinct sequence of symbols. In order to do this, we need a function that will map sequences of symbols into the unit interval. A function that maps random variables, and sequences of random variables, into the unit interval is the cumulative distribution function (cdf) of the random variable associated with the source. This is the function we will use in developing the arithmetic code The use of the cumulative distribution function to generate a binary code for a sequence has a rather interesting history. Shannon mentioned an approach using the cumulative distribution function when describing what is now known as the Shannon-Fano code. Peter Elias, another member of Fano's first information theory class at MIT (this class also included Huffman), came up with a recursive implementation for this idea. However, he never published it, and we only know about it through a mention in a 1963 book on information theory by Abramson. Abramson described this coding approach in a note to a chapter. In another book on information theory by Jelinek in 1968, the idea of arithmetic coding is further developed, this time in an appendix, as an example of variable-length coding. Modern arithmetic coding owes its birth to the independent discoveries in 1976 of Pasco and Rissanen who solved the problem of finite precision. Finally, several papers appeared that provided practical arithmetic coding algorithms, the most well known of which is the paper by Rissanen and Langdon. Before we begin our development of the arithmetic code, we need to establish some nota-tion. Recall that a random variable maps the outcomes, or sets of outcomes, of an experiment to values on the real number line. For example, in a coin-tossing experiment, the random variable could map a head to zero and a tail to one (or it could map a head to 2367.5 and a tail to −192). To use this technique, we need to map the source symbols or letters to numbers. For convenience, in the discussion in this chapter we will use the mapping

$$X(a_i) = i \quad a_i \in A \tag{1}$$

where $A = \{a_1, a_2, \ldots, a_m\}$ is the alphabet for a discrete source and X is a random variable. This mapping means that given a probability model P for the source, we also have a probability density function for the random variable

$$P(X = i) = P(a_i)$$

and the cumulative density function can be defined as

$$F_X(i) = \sum_{k=1}^{i} P(X = k)$$

Notice that for each symbol a_i with a nonzero probability, we have a distinct value of $F_X(i)$.We will use this fact in what follows to develop the arithmetic code. Our development may be more detailed than what you are looking for, at least on the first reading. If so, skip or skim Sections 4.3.1–4.4.1 and go directly to Section

Generating a Tag: The procedure for generating the tag works by reducing the size of the interval in which the tag resides as more and more elements of the sequence are received.

We start out by first dividing the unit interval into subintervals of the form $[F_X (i - 1), F_X (i))$, $i = 1, \ldots, m$. Because the minimum value of the cdf is zero and the maximum value is one, this exactly partitions the unit interval. We associate the subinterval $[F_X (i - 1), F_X (i)$ with the symbol a_i . The appearance of the first symbol in the sequence restricts the interval containing the tag to one of these subintervals. Suppose the first symbol was a_k . Then the interval containing the tag value will be the subinterval $[F_X (k - 1), F_X (k))$. This subinterval is now partitioned in exactly the same proportions as the original interval. That is, the jth interval corresponding to the symbol a_j is given by $[F_X (k - 1) + F_X (j - 1) \times (F_X (k) - F_X (k - 1), F_X (k - 1) + F_X (j) \times (F_X (k) - F_X (k - 1))]$. So if the second symbol in the sequence is a_j , then the interval containing the tag value becomes $[F_X (k - 1) + F_X (j - 1) \times F_X (k) - F_X (k - 1), F_X (k - 1) + F_X (j) \times (F_X (k) - F_X (k - 1)))]$. Each succeeding symbol causes the tag to be restricted to a subinterval that is further partitioned in the same proportions. This process can be more clearly understood through an example.

Consider a simple dice-throwing experiment with a fair die. The outcomes of a roll of the die can be mapped into the numbers $\{1, 2, \ldots, 6\}$. For a fair die

$$P(X = k) = \frac{1}{6} \quad \text{for } k = 1, 2, \ldots, 6$$

Therefore, using (3) we can find the tag for X = 2 as

$$\overline{T}_X(2) = P(X = 1) + \frac{1}{2} P(X = 2) = 1 + \frac{1}{2} = 0.25$$

and the tag for $X = 5$ as

$$\overline{T}_X(5) = \sum_{k=1}^{4} P(X = k) + \frac{1}{2} P(X = 5) = 0.75$$

The tags for all other outcomes are shown in Table 4.3 where the overbar denotes repeating decimal values.

TABLE 4.3 Toss for outcomes in a dice-throwing experment.

Outcome	Tag
1	$0.08\overline{33}$
3	$0.41\overline{66}$
4	$0.58\overline{33}$
6	$0.91\overline{66}$

As we can see from the example above, giving a unique tag to a sequence of length one is an easy task. This approach can be extended to longer sequences by imposing an order on the sequences. We need to impose an order on the sequences because we will assign a tag to a particular sequence $\mathbf{x}_i$ as

$$\overline{T}_{\mathbf{X}}^{(m)}(\mathbf{x}_i) = \sum_{\mathbf{y} < \mathbf{x}_i} P(\mathbf{y}) + \frac{1}{2} P(\mathbf{x}_i) \qquad (4)$$

where $\mathbf{y} < \mathbf{x}$ means that $\mathbf{y}$ precedes $\mathbf{x}$ in the ordering, and the superscript denotes the length of the sequence.An easy ordering to use is lexicographic ordering. In lexicographic ordering, the ordering of letters in an alphabet induces an ordering on the words constructed from this alphabet. The ordering of words in a dictionary is a good (maybe the original) example of

lexicographic ordering. Dictionary order is sometimes used as a synonym for lexicographic order.

E x a m p l e : We can extend Example so that the sequence consists of two rolls of a die. Using the ordering scheme described above, the outcomes (in order) would be 11, 12, 13, . . . , 66. The tags can then be generated using Equation(4). For example, the tag for the sequence 1 3 would be

$$\bar{T}_X(13) = P(\mathbf{x} = 11) + P(\mathbf{x} = 12) + 1/2P(\mathbf{x} = 13) \quad (5)$$
$$= 1/36 + 1/36 + 1/2(1/36) \quad (6)$$
$$= 5/72 \quad (7)$$

Notice that to generate the tag for the sequence 1 3 we did not have to generate a tag for every other possible message. However, based on Equation (4) and Example , we need to know the probability of every sequence that is "less than" the sequence for which the tag is being generated. The requirement that the probability of all sequences of a given length be explicitly calculated can be as prohibitive as the requirement that we have codewords for all sequences of a given length. Fortunately, we shall see that to compute a tag for a given sequence of symbols, all we need is the probability of individual symbols, or the probability model.

Generating a Binary Code:- Using the algorithm described in the previous section, we can obtain a tag for a given sequence **x**. However, the binary code for the sequence is what we really want to know. We want to find a binary code that will represent the sequence **x** in a unique and efficient manner.

We have said that the tag forms a unique representation for the sequence. This means that the binary representation of the tag forms a unique binary code for the sequence. However, we have placed no restrictions on what values in the unit interval the tag can take. The binary representation of some of these values would be infinitely long, in which case, although the code is unique, it may not be efficient. To make the code efficient, the binary representation has to be truncated. But if we truncate the representation, is the resulting code still unique? Finally, is the resulting code efficient? How far or how close is the average number of bits per symbol from the entropy.

Comparison of Huffman and Arithmetic Coding:- We have described a new coding scheme that, although more complicated than Huffman coding, allows us to code *sequences* of symbols. How well this coding scheme works depends on how it is used. Let's first try to use this code for encoding sources for which we know the Huffman code.

Looking at Example 4.4.1, the average length for this code is

$$l = 2 \times 0.5 + 3 \times 0.25 + 4 \times 0.125 + 4\ 0.125 \quad (83)$$
$$= 2.75 \text{ bits/symbol} \quad (84)$$

T A B L E Arithmetic code for two-symbol sequences.

Message	$P(x)$	$\bar{T}_X(x)$	$\bar{T}_X(x)$ in Binary	$\lfloor \log \frac{1}{P(x)} \rfloor + 1$	Code
11	.25	.125	.001	3	001
12	.125	.3125	.0101	4	0101
13	.0625	.40625	.01101	5	01101
14	.0625	.46875	.01111	5	01111
21	.125	.5625	.1001	4	1001

22	.0625	.65625	.10101	5	10101
23	.03125	.703125	.101101	6	101101
24	.03125	.734375	.101111	6	101111
31	.0625	.78125	.11001	5	11001
32	.03125	.828125	.110101	6	110101
33	.015625	.8515625	.1101101	7	1101101
34	.015625	.8671875	.1101111	7	1101111
41	.0625	.90625	.11101	5	11101
42	.03125	.953125	.111101	6	111101
43	.015625	.9765625	.1111101	7	1111101
44	.015625	.984375	.1111111	7	1111111

Recall from Section that the entropy of this source was 1.75 bits/symbol and the Huffman code achieved this entropy. Obviously, arithmetic coding is not a good idea if you are going to encode your message one symbol at a time. Let's repeat the example with messages consisting of two symbols.

Dictionary Techniques:-

Introduction:- In many applications, the output of the source consists of recurring patterns. A classic example is a text source in which certain patterns or words recur constantly. Also, there are certain patterns that simply do not occur, or if they do occur, it is with great rarity. For example, sgiomlaireached might be an annoying habit but the word probably occurs in a very small fraction of the text sources in existence.A very reasonable approach to encoding such sources is to keep a list, or dictionary, of frequently occurring patterns. When these patterns appear in the source output, they are encoded with a reference to the dictionary. If the pattern does not appear in the dictionary, then it can be encoded using some other less efficient method. In effect, we are splitting the input into two classes: frequently occurring patterns and infrequently occurring patterns. For this technique to be effective, the class of frequently occurring patterns, and hence the size of the dictionary, must be much smaller than the number of all possible patterns.

Suppose we have a particular text that consists of four-character words, three characters from the 26 lowercase letters of the English alphabet followed by a punctuation mark. Sup-pose our source alphabet consists of the 26 lowercase letters of the English alphabet and the punctuation marks comma, period, exclamation mark, question mark, semicolon, and colon. In other words, the size of the input alphabet is 32. If we were to encode the text source one character at a time, treating each character as an equally likely event, we would need 5 bits per character. Treating all $32^4 (= 2^{20} = 1,048,576)$ four-character patterns as equally likely, we have a code that assigns 20 bits to each four-character pattern. Let us now put the 256 most likely four-character patterns into a dictionary. The transmission scheme works as follows. Whenever we want to send a pattern that exists in the dictionary, we will send a 1-bit flag, say, a 0, followed by an 8-bit index corresponding to the entry in the dictionary. If the pattern is not in the dictionary, we will send a 1 followed by the 20-bit encoding of the pattern. If the pattern we encounter is not in the dictionary, we will actually use more bits than in the original scheme, 21 instead of 20. But if it is in the dictionary, we will send only 9 bits. The utility of our scheme will depend on the percentage of the words we encounter that are in the dictionary. We can get an idea about the utility of our scheme by calculating the average number of bits per pattern. If the probability of encountering a pattern from the dictionary is p, then the average number of bits per pattern R is given by

$$R = 9\,p + 21(1 - p) = 21 - 12\,p$$

For our scheme to be useful, R should have a value less than 20. This happens when p 0.084. This does not seem like a very large number. However, note that if all patterns were occurring in an

equally likely manner, the probability of encountering a pattern from the dictionary would be less than 0.00025. We do not simply want a coding scheme that performs slightly better than the simple-minded approach of coding each pattern as equally likely; we would like to improve the performance as much as possible. In order for this to happen, p should be as large as possible. This means that we should carefully select patterns that are most likely to occur as entries in the dictionary. To do this, we have to have a pretty good idea about the structure of the source output. If we do not have information of this sort available to us prior to the encoding of a particular source output, we need to acquire this information somehow when we are encoding. If we feel we have sufficient prior knowledge, we can use a static approach; if not, we can take an adaptive approach.

Static Dictionary:- Choosing a static dictionary technique is most appropriate when considerable prior knowl-edge about the source is available. This technique is especially suitable for use in specific applications. For example, if the task were to compress the student records at a university, a static dictionary approach may be the best. This is because we know ahead of time that certain words and phrases such as "Name" and "Student ID" are going to appear in almost all of the records. Other words such as "Sophomore," "credits," and so on will occur quite often. Depending on the location of the university, certain digits in Social Security numbers are more likely to occur. For example, in Nebraska most student ID numbers used to begin with the digits 505. In fact, most entries will be of a recurring nature. In this situation, it is highly efficient to design a compression scheme based on a static dictionary containing the recurring patterns. Similarly, there could be a number of other situations in which an application-specific or data-specific static-dictionary-based coding scheme would be the most efficient. It should be noted that these schemes would work well only for the applications and data they were designed for. If these schemes were to be used with different applications, they may cause an expansion of the data instead of compression.
A static dictionary technique that is less specific to a single application is digram coding.

Digram Coding:- One of the more common forms of static dictionary coding is digram coding. In this form of coding, the dictionary consists of all letters of the source alphabet followed by as many pairs of letters, called *digrams*, as can be accommodated by the dictionary. For example, suppose we were to construct a dictionary of size 256 for digram coding of all printable ASCII characters. The first 95 entries of the dictionary would be the 95 printable ASCII characters. The remaining 161 entries would be the most frequently used pairs of characters.
The digram encoder reads a two-character input and searches the dictionary to see if this input exists in the dictionary. If it does, the corresponding index is encoded and transmitted. If it does not, the first character of the pair is encoded. The second character in the pair then becomes the first character of the next digram. The encoder reads another character to complete the digram, and the search procedure is repeated.
Example:- Suppose we have a source with a five-letter alphabet $A = \{a, b, c, d, r\}$. Based on knowledge about the source, we build the dictionary shown in Table 5.1.

T A B L E 5.1 A sample dictionary.

Code	Entry	Code	Entry
000	*a*	100	*r*
001	*b*	101	*ab*
010	*c*	110	*ac*
011	*d*	111	*ad*

T A B L E 5.2: Thirty most frequently occurring pairs of character in a 41,364-character-long LaTeX document.

Pair	Count	Pair	Count
eb/	1128	*ar*	314
bt/	838	*at*	313
b/b/	823	*b/w*	309
th	817	*te*	296
he	712	*bs/*	295
in	512	*db/*	272
sb/	494	*bo/*	266
er	433	*io*	257
ba/	425	*co*	256
tb/	401	*re*	247
en	392	*b/$*	246
on	385	*rb/*	239
nb/	353	*di*	230
ti	322	*ic*	229
bi/	317	*ct*	226

Suppose we wish to encode the sequence

abracadabra

The encoder reads the first two characters *ab* and checks to see if this pair of letters exists in the dictionary. It does and is encoded using the codeword 101. The encoder then reads the next two characters *ra* and checks to see if this pair occurs in the dictionary. It does not, so the encoder sends out the code for *r* , which is 100, then reads in one more character, *c*, to make the two-character pattern *ac*. This does exist in the dictionary and is encoded as 110. Continuing in this fashion, the remainder of the sequence is coded. The output string for the given input sequence is 101100110111101100000.

A list of the 30 most frequently occurring pairs of characters in an earlier version of this chapter is shown in Table 5.2. For comparison, the 30 most frequently occurring pairs of characters in a set of C programs is shown in Table 5.3.

In these tables, *b/* corresponds to a space and *nl* corresponds to a new line. Notice how different the two tables are. It is easy to see that a dictionary designed for compressing LaTeX documents would not work very well when compressing C programs. However, we generally want techniques that will be able to compress a variety of source outputs. If we want to compress computer files, we do not want to change techniques based on the content of the file. Rather, we would like the technique to *adapt* to the characteristics of the source output. We discuss adaptive-dictionary-based techniques in the next section.

T A B L E 5 . 3 Thirty most frequently occurring pairs characters in a collection of C programs containing 64,983 characters.

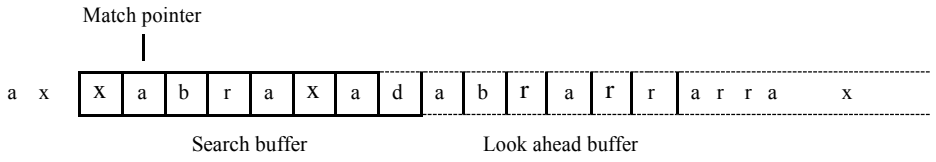
Pair	Count	Pair	Count
b/b/	5728	*st*	442
nlb/	1471	*le*	440
; nl	1133	*ut*	440
in	985	*f (*	416
nt	739	*ar*	381
= b/	687	*or*	374
bi/	662	*rb/*	373

tb/	615	*en*	371
b/ =	612	*er*	358
);	558	*ri*	357
, b/	554	*at*	352
nlnl	506	*pr*	351
b/ f	505	*te*	349
eb/	500	*an*	348
*b/**	444	*lo*	347

Adaptive Dictionary:- Most adaptive-dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 [52] and 1978 [53]. These papers provide two different approaches to adaptively building dictionaries, and each approach has given rise to a number of variations. The approaches based on the 1977 paper are said to belong to the LZ77 family (also known as LZ1), while the approaches based on the 1978 paper are said to belong to the LZ78, or LZ2, family. The transposition of the initials is a historical accident and is a convention we will observe in this book. In the following sections, we first describe an implementation of each of the adaptive-dictionary-based compression approaches followed by some of the more well-known variations.

LZ77 Approach:- In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window, as shown in Figure 5.1. The window consists of two parts, a search buffer that contains a portion of the recently encoded sequence and a lookahead buffer that contains the next portion of the sequence to be encoded. In Figure 5.1, the search buffer contains eight symbols, while the lookahead buffer contains seven symbols. In practice, the size of the buffers is significantly larger; however, for the purpose of explanation, we will keep the buffer sizes small.

To encode the sequence in the lookahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the lookahead



F I G U R E 5 . 1 **Encoding using the LZ77 approach.**

buffer. The distance of the pointer from the lookahead buffer is called the *offset*. The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the lookahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the lookahead buffer, starting with the first symbol, is called the length of the match. The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triple *_o, l, c _*, where *o* is the offset, *l* is the length of the match, and *c* is the codeword corresponding to the symbol in the lookahead buffer that follows the match. For example, in Figure 5.1 the pointer is pointing to the beginning of the longest match. The offset *o* in this case is 7, the length of the match *l* is 4, and the symbol in the lookahead buffer following the match is *r* .The reason for sending the third element in the triple is to take care of the situation where no match for the symbol in the lookahead buffer can be found in the search buffer. In this case, the offset and match-length values are set to 0, and the third element of the triple is the code for the symbol itself.

Example: The LZ77 Approach

Suppose the sequence to be encoded is

. . . **cabracadabrarrarrad** . . .

Suppose the length of the window is 13, the size of the lookahead buffer is 6, and the current condition is

cabraca | dabrar

with dabrar in the lookahead buffer. We look back in the already encoded portion of the window to find a match for d. As we can see, there is no match, so we transmit the triple _0, 0, *C(d)*_. The first two elements of the triple show that there is no match to *d* in the search buffer, while *C(d)* is the code for the character *d*. This seems like a wasteful way to encode a single character, and we will have more to say about this later.

For now, let's continue with the encoding process. As we have encoded a single character, we move the window by one character. Now the contents of the buffer are

abracad | abrarr

with *abrarr* in the lookahead buffer. Looking back from the current location, we find a match to *a* at an offset of two. The length of this match is one. Looking further back, we have another match for *a* at an offset of four; again the length of the match is one. Looking back even further in the window, we have a third match for *a* at an offset of seven. However, this time the length of the match is four (see Figure 5.2). So we encode the string *abra* with the triple _7, 4, *C(r)*_, and move the window forward by five characters. The window now contains the following characters:

adabrar | rarrad

Now the lookahead buffer contains the string *rarrad*. Looking back in the window, we find a match for *r* at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.

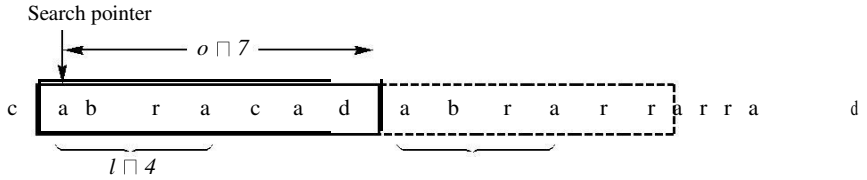
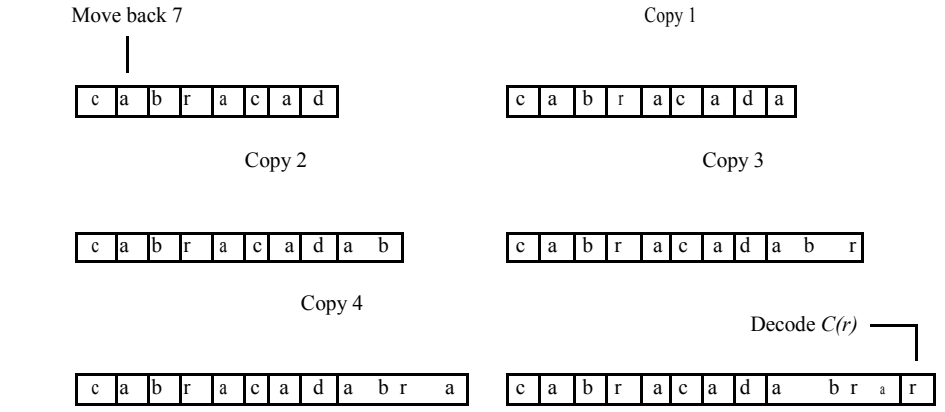


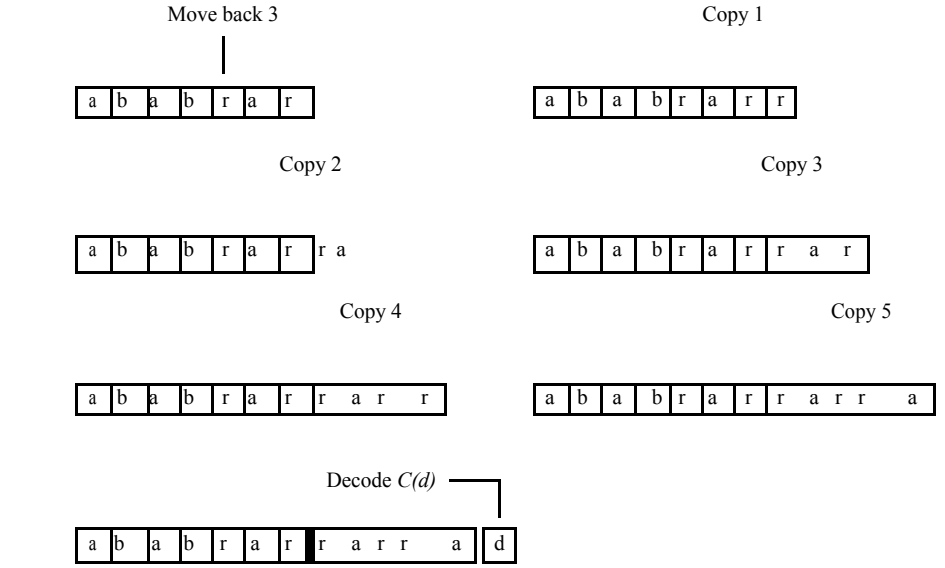
FIGURE 5.2 The encoding process.

Why this is so will become clearer when we decode the sequence. To see how the decoding works, let us assume that we have decoded the sequence cabraca and we receive the triples _0, 0, *C(d)*_,7, 4, *C(r)*_, & 3, 5, *C(d)*_. The first triple is easy to decode; there was no match within the previously decoded string, and the next symbol is *d*. The decoded string is now cabracad. The first element of the next triple tells the decoder to move the copy pointer back seven characters, and copy four characters from that point. The decoding process works as shown in Figure 5.3.

Finally, let's see how the triple _3, 5, *C(d)*_ gets decoded. We move back three characters and start copying. The first three characters we copy are *rar* . The copy pointer moves once again, as shown in Figure 5.4, to copy the recently copied character *r* . Similarly, we copy the next character *a*. Even though we started copying only three characters back, we end up decoding five characters. Notice that the match only has to *start* in the search buffer; it can extend into the lookahead buffer. In fact, if the last character in the lookahead buffer had been *r* instead of d, followed by several more repetitions of rar , the entire sequence of repeated *rar* s could have been encoded with a single triple As we can see, the LZ77 scheme is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. The authors of this algorithm showed that the performance of this algorithm asymptotically approached the best that could be obtained by using a scheme that had full knowledge about the statistics of the source. While this may be true asymptotically, in practice there are a number of ways of improving the performance of the LZ77 algorithm as described here. Furthermore, by using the recent portions of the sequence, there is an assumption of sorts being used here—that is, that patterns recur "close" together. As we shall see, the authors removed this "assumption" and came up with an entirely different adaptive-dictionary-based scheme in LZ78. Before we get to that, let us look at the different variations of the LZ77 algorithm



F I G U R E 5 . 3 **Decoding the triple 7, 4, C(r) .**



F I G U R E 5 . 4 **Decoding the triple 3, 5, C(d) .**

LZ78 Approach:-

The LZ77 approach implicitly assumes that like patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding. However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured. The worst-case situation would be where the sequence to be encoded was periodic with a period longer than the search buffer. Consider Figure 5.5.

This is a periodic sequence with a period of nine. If the search buffer had been just one symbol longer, this sequence could have been significantly compressed. As it stands, none of the new symbols will have a match in the search buffer and will have to be represented by separate codewords. As this involves sending along overhead (a 1-bit flag for LZSS and a triple for the original LZ77 algorithm), the net result will be an expansion rather than a compression.

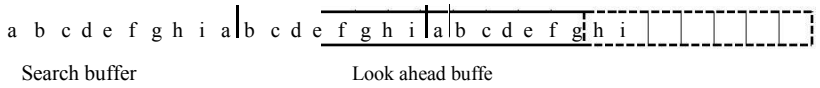


TABLE 5.4 The initial dictionary.

Index	Entry
1	*w*
2	*a*
3	*b*

Although this is an extreme situation, there are less drastic circumstances in which the finite view of the past would be a drawback. The LZ78 algorithm solves this problem by dropping the reliance on the search buffer and keeping an explicit dictionary. This dictionary has to be built at both the encoder and decoder, and care must be taken that the dictionaries are built in an identical manner. The inputs are coded as a double _*i, c*_, with *i* being an index corresponding to the dictionary entry that was the longest match to the input and *c* being the code for the character in the input following the matched portion of the input. As in the case of LZ77, the index value of 0 is used in the case of no match. This double then becomes the newest entry in the dictionary. Thus, each new entry into the dictionary is one new symbol concatenated with an existing dictionary entry. To see how the LZ78 algorithm works, consider the following example.

Example : The LZ78 Approach

Let us encode the following sequence using the LZ78 approach

wabbab/wabbab/wabbab/wabbab/woob/woob/woo.

where *b/* stands for space. Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to 0. The first three encoder outputs are _0*, C(w)*_, _0*, C(a)*_, and _0*, C(b)*_, and the dictionary looks like Table 5.4.The fourth symbol is a *b*, which is the third entry in the dictionary. If we append the next symbol, we would get the pattern *ba*, which is not in the dictionary, so we encode these two symbols as _3*, C(a)*_ and add the pattern *ba* as the fourth entry in the dictionary. Continuing in this fashion, the encoder output and the dictionary develop as in Table 5.5. Notice that the entries in the dictionary generally keep getting longer, and if this particular sentence was repeated often, as it is in the song, after a while the entire sentence would be an entry in the dictionary.While the LZ78 algorithm has the ability to capture patterns and hold them indefinitely, it also has a rather serious drawback. As seen from the example, the dictionary keeps growing without bound. In a practical situation, we would have to stop the growth of the dictionary at some stage and then either prune it back or treat the encoding as a fixed dictionary scheme. We will discuss some possible approaches when we study applications of dictionary coding.

TABLE 5.5 Development of the dictionary.

	Dictionary	
Encoder Output	Index	Entry
_0, *C(w)*_	01	*w*
_0, *C(a)*_	02	*a*
_0, *C(b)*_	03	*b*
_3, *C(a)*_	04	*ba*
_0, *C(/b)*_	05	*b/*
_1, *C(a)*_	06	*wa*
_3, *C(b)*_	07	*bb*
_2, *C(/b)*_	08	*ab/*
_6, *C(b)*_	09	*wab*
_4, *C(/b)*_	10	*bab/*
_9, *C(b)*_	11	*wabb*
_8, *C(w)*_	12	*ab/w*
_0, *C(o)*_	13	*o*
_13, *C(/b)*_	14	*ob/*
_1, *C(o)*_	15	*wo*
_14, *C(w)*_	16	*ob/w*
_13, *C(o)*_	17	*oo*

Example : The LZW Algorithm Encoding

We will use the sequence previously used to demonstrate the LZ78 algorithm as our input sequence:

wabbab/wabbab/wabbab/wabbab/woob/woob/woo

Assuming that the alphabet for the source is {*b/, a, b, o, w*}, the LZW dictionary initially looks like Table 5.6.

TABLE 5.6 Initial LZW dictionary.

Index	Entry
1	*b/*
2	*a*
3	*b*
4	*o*
5	*w*

TABLE 5.7 Constructing the 12th entry of the LZW dictionary.

Index	Entry
01	*b/*
02	*a*
03	*b*
04	*o*
05	*w*
06	*wa*
07	*ab*
08	*bb*
09	*ba*
10	*a/b/*
11	*b/*w
12	*w…*

The encoder first encounters the letter *w*. This "pattern" is in the dictionary so we concate-nate the next letter to it, forming the pattern *wa*. This pattern is not in the dictionary, so we encode *w* with its dictionary index 5, add the pattern *wa* to the dictionary as the sixth element of the dictionary, and begin a new pattern starting with the letter *a*. As *a* is in the dictionary, we concatenate the next element *b* to form the pattern *ab*. This pattern is not in the dictionary, so we encode *a* with its dictionary index value 2, add the pattern *ab* to the dictionary as the seventh element of the dictionary, and start constructing a new pattern with the letter *b*. We continue in this manner, constructing two-letter patterns, until we reach the letter *w* in the sec-ond *wabba*. At this point, the output of the encoder consists entirely of indices from the initial dictionary: 5 2 3 3 2 1. The dictionary looks like Table 5.7. (The 12th entry in the dictionary is still under construction.) The next symbol in the sequence is *a*. Concatenating this to *w*, we get the pattern *wa*. This pattern already exists in the dictionary (item 6), so we read the next symbol, which is *b*. Concatenating this to *wa*, we get the pattern *wab*. This pattern does not exist in the dictionary, so we include it as the 12th entry in the dictionary and start a new pattern with the symbol *b*. We also encode *wa* with its index value of 6. Notice that after a series of two-letter entries, we now have a three-letter entry. As the encoding progresses, the length of the entries keeps increasing. The longer entries in the dictionary indicate that the dictionary is capturing more of the structure in the sequence. The dictionary at the end encoding process is shown in Table 5.8. Notice that the 12th through the 19th entries are all either three or four letters in length. Then we encounter the pattern *woo* for the first time and we drop back to two-letter patterns for three more entries, after which we go back to entries of increasing length.

The encoder output sequence is 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4.

T A B L E 5 . 8 The LZW dictionary for encoding wabba*b*/wabba*b*/wabba*b*/wabba*b*/ woo*b*/woo*b*/woo.

Index	Entry	Index	Entry
01	*b/*	14	*ab/ w*
02	*a*	15	*wabb*
03	*b*	16	*bab/*
04	*o*	17	*b/ wa*
05	*w*	18	*abb*
06	*wa*	19	*bab/ w*
07	*ab*	20	*wo*
08	*bb*	21	*oo*
09	*ba*	22	*ob/*
10	*ab/*	23	*b/ wo*
11	*b/ w*	24	*oob/*
12	*wab*	25	*b/ woo*
13	*bba*		

Example : The LZW Algorithm Decoding:- In this example we will take the encoder output from the previous example and decode it using the LZW algorithm. The encoder output sequence in the previous example was

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

This becomes the decoder input sequence. The decoder starts with the same initial dictionary as the encoder (Table 5.6).

The index value 5 corresponds to the letter *w*, so we decode *w* as the first element of our sequence. At the same time, we begin construction of the next element of the dictionary in order to mimic the dictionary construction procedure of the encoder. We start with the letter *w*. This pattern exists in the dictionary, so we do not add it to the dictionary and continue with the decoding process. The next decoder input is 2, which is the index corresponding to the letter *a*. We decode an *a* and concatenate it with our current pattern to form the pattern *wa*. As this does not exist in the dictionary, we add it as the sixth element of the dictionary and start a new pattern beginning with the letter *a*. The next four inputs 3 3 2 1 correspond to the letters *bbab/* and generate the dictionary entries *ab*, *bb*, *ba*, and *ab/*. The dictionary now looks like Table 5.9, where the 11th entry is under construction. The next input is 6, which is the index of the pattern *wa*. Therefore, we decode a *w* and an *a*. We first concatenate *w* to the existing pattern, which is *b/* and form the pattern *b/w*. As *b/w*

T A B L E 5 . 9 Constructing the 11th entry of the LZW dictionary while decoding.

Index	Entry
01	*b/*
02	*a*
03	*b*
04	*o*
05	*w*
06	*wa*
07	*ab*
08	*bb*
09	*ba*
10	*ab/*
11	*b/. . .*

does not exist in the dictionary, it becomes the 11th entry. The new pattern now starts with the letter *w*. We had previously decoded the letter *a*, which we now concatenate to *w* to obtain the pattern *wa*. This pattern is contained in the dictionary, so we decode the next input, which is 8. This corresponds to the entry *bb* in the dictionary. We decode the first *b* and concatenate it to the pattern *wa* to get the pattern *wab*. This pattern does not exist in the dictionary, so we add it as the 12th entry in the dictionary and start a new pattern with the letter *b*. Decoding the second *b* and concatenating it to the new pattern, we get the pattern *bb*. This pattern exists in the dictionary, so we decode the next element in the sequence of encoder outputs. Continuing in this fashion, we can decode the entire sequence. Notice that the dictionary being constructed by the decoder is identical to that constructed by the encoder.

There is one particular situation in which the method of decoding the LZW algorithm described above breaks down. Suppose we had a source with an alphabet $A = \{a, b\}$, and we were to encode the sequence beginning with *abababab* . . .. The encoding process is still the same. We begin with the initial dictionary shown in Table 5.10 and end up with the final dictionary shown in Table 5.11.

The transmitted sequence is 1235 . . .. This looks like a relatively straightforward sequence to decode. However, when we try to do so, we run into a snag. Let us go through the decoding process and see what happens.

TABLE 5.10 **Initial dictionary for abababab.**

Index	Entry
1	*a*
2	*b*

TABLE 5.11 **Final dictionary for abababab.**

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*aba*
6	*abab*
7	*b . . .*

TABLE 5.12 **Constructing the fourth entry of the dictionary while decoding.**

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*b . . .*

TABLE 5.13 **Constructing the fifth entry (stage one).**

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*a . . .*

We begin with the same initial dictionary as the encoder (Table 5.10). The first two elements in the received sequence 1235 . . . are decoded as *a* and *b*, giving rise to the third dictionary entry *ab*, and the beginning of the next pattern to be entered in the dictionary, *b*. The dictionary at this point is shown in Table 5.12.

The next input to the decoder is 3. This corresponds to the dictionary entry *ab*. Decoding each in turn, we first concatenate *a* to the pattern under construction to get *ba*. This pattern is not contained in the dictionary, so we add this to the dictionary (keep in mind, we have not used the *b* from *ab* yet), which now looks like Table 5.13.

The new entry starts with the letter *a*. We have only used the first letter from the pair *ab*. Therefore, we now concatenate *b* to *a* to obtain the pattern *ab*. This pattern is contained in the dictionary, so we continue with the decoding process. The dictionary at this stage looks like Table 5.14.

TABLE 5.14 Constructing the fifth entry (stage two).

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*ab . . .*

TABLE 5.15 Completion of the fifth entry.

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*aba*
6	*a . . .*

The first four entries in the dictionary are complete, while the fifth entry is still under construction. However, the very next input to the decoder is 5, which corresponds to the incomplete entry. How do we decode an index for which we do not as yet have a complete dictionary entry?

The situation is actually not as bad as it looks. While we may not have a fifth entry for the dictionary, we do have the beginnings of the fifth entry, which is *ab* . . .. Let us, for the moment, pretend that we do indeed have the fifth entry and continue with the decoding process. If we had a fifth entry, the first two letters of the entry would be *a* and *b*. Concatenating *a* to the partial new entry we get the pattern *aba*. This pattern is not contained in the dictionary, so we add this to our dictionary, which now looks like Table 5.15. Notice that we now have the fifth entry in the dictionary, which is *aba*. We have already decoded the *ab* portion of *aba*. We can now decode the last letter *a* and continue on our merry way.

This means that the LZW decoder has to contain an exception handler to handle the special case of decoding an index that does not have a corresponding complete entry in the decoder dictionary.

Applications:-Since the publication of Terry Welch's article, there has been a steadily increasing number of applications that use some variant of the LZ78 algorithm. Among the LZ78 variants, by far the most popular is the LZW algorithm. In this section, we describe two applications of LZW: GIF and V.42 bis. While the LZW algorithm was initially the algorithm of choice, patent concerns led to increasing use of the LZ77 algorithm. The most popular implementation of the LZ77

algorithm is the deflate algorithm initially designed by Phil Katz. It is part of the popular zlib library developed by Jean-Loup Gailly and Mark Adler. Jean-Loup Gailly also used deflate in the widely used gzip algorithm. The deflate algorithm is also used in PNG, which we describe below.

File Compression UNIX compress:-The UNIX compress command is one of the earlier applications of LZW. The size of the dictionary is adaptive. We start with a dictionary of size 512. This means that the transmitted codewords are 9 bits long. Once the dictionary has filled up, the size of the dictionary is doubled to 1024 entries. The codewords transmitted at this point have 10 bits. The size of the dictionary is progressively doubled as it fills up. In this way, during the earlier part of the coding process when the strings in the dictionary are not very long, the codewords used to encode them also have fewer bits. The maximum size of the codeword, b_{max}, can be set by the user to between 9 and 16, with 16 bits being the default. Once the dictionary contains 2^{bmax} entries, compress becomes a static dictionary coding technique. At this point the algorithm monitors the compression ratio. If the compression ratio falls below a threshold, the dictionary is flushed, and the dictionary building process is restarted. This way, the dictionary always reflects the local characteristics of the source.

Image Compressio The Graphics Interchange Format (GIF):-The Graphics Interchange Format (GIF) was developed by CompuServe Information Service to encode graphical images. It is another implementation of the LZW algorithm and is very similar to the compress command. The compressed image is stored with the first byte being the minimum number of bits b per pixel in the original image. For the images we have been using as examples, this would be eight. The binary number 2^b is defined to be the clear code. This code is used to reset all compression and decompression parameters to a start-up state. The initial size of the dictionary is 2^{b+1}. When this fills up, the dictionary size is doubled, as was done in the compress algorithm, until the maximum dictionary size of 4096 is reached. At this point, the compression algorithm behaves like a static dictionary algorithm. The codewords from the LZW algorithm are stored in blocks of characters. The characters are 8 bits long, and the maximum block size is 255. Each block is preceded by a header that contains the block size. The block is terminated by a block terminator consisting of eight 0s. The end of the compressed image is denoted by an end-of-information code with a value of $2^b + 1$. This codeword should appear before the block terminator.GIF has become quite popular for encoding all kinds of images, both computer-generated and "natural" images. While GIF works well with computer-generated graphical images and pseudocolor or color-mapped images, it is generally not the most efficient way to losslessly compress images of natural scenes, photographs, satellite images, and so on. In Table 5.16, we give the file sizes for the GIF-encoded test images. For comparison, we also include the file sizes for arithmetic coding the original images and arithmetic coding the differences

TABLE 5.16 Comparison of GIF with arithmetic coding.

Image	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	51,085	53,431	31,847
Sensin	60,649	58,306	37,126
Earth	34,276	38,248	32,137
Omaha	61,580	56,061	51,393

Notice that even if we account for the extra overhead in the GIF files, for these images GIF barely holds its own even with simple arithmetic coding of the original pixels. While this might seem odd

at first, if we examine the image on a pixel level, we see that there are very few repetitive patterns compared to a text source. Some images, like the Earth image, contain large regions of constant values. In the dictionary coding approach, these regions become single entries in the dictionary. Therefore, for images like these, the straightforward dictionary coding approach does hold its own. However, for most other images, it would probably be preferable to perform some preprocessing to obtain a sequence more amenable to dictionary coding. The Portable Network Graphics (PNG) standard described next develops an appropriate preprocessor to take advantage of the fact that in natural images the pixel-to-pixel variation is generally small.

Compression over Modems V.42 bits:-The ITU-T Recommendation V.42 bis is a compression standard devised for use over a tele-phone network along with error-correcting procedures described in CCITT Recommendation V.42. This algorithm is used in modems connecting computers to remote users. The algorithm described in this recommendation operates in two modes, a transparent mode and a compressed mode. In the transparent mode, the data are transmitted in uncompressed form, while in the compressed mode an LZW algorithm is used to provide compression.
The reason for the existence of two modes is that at times the data being transmitted do not have repetitive structure and, therefore, cannot be compressed using the LZW algorithm. In this case, the use of a compression algorithm may even result in expansion. In these situations, it is better to send the data in an uncompressed form. A random data stream would cause the dictionary to grow without any long patterns as elements of the dictionary. This means that most of the time the transmitted codeword would represent a single letter from the source alphabet. As the dictionary size is much larger than the source alphabet size, the number of bits required to represent an element in the dictionary is much more than the number of bits required to represent a source letter. Therefore, if we tried to compress a sequence that does not contain repeating patterns, we would end up with more bits to transmit than if we had not performed any compression. Data without repetitive structure are often encountered when a previously compressed file is transferred over the telephone lines.

Prediction with Partial Match (ppm):- The best-known context-based algorithm is the ppm algorithm, first proposed by Cleary and Witten in 1984. It has not been as popular as the various Ziv-Lempel-based algorithms mainly because of the faster execution speeds of the latter algorithms. Lately, with the devel-opment of more efficient variants, ppm-based algorithms are becoming increasingly popular.

The idea of the ppm algorithm is elegantly simple. We would like to use large contexts to determine the probability of the symbol being encoded. However, the use of large contexts would require us to estimate and store an extremely large number of conditional probabilities, which might not be feasible. Instead of estimating these probabilities ahead of time, we can reduce the burden by estimating the probabilities as the coding proceeds. This way we only need to store those contexts that have occurred in the sequence being encoded. This is a much smaller number than the number of all possible contexts. While this mitigates the problem of storage, it also means that, especially at the beginning of an encoding, we will need to code letters that have not occurred previously in this context. In order to handle this situation, the source coder alphabet always contains an escape symbol, which is used to signal that the letter to be encoded has not been seen in this context.

(1) The Basic Algorithm:- The basic algorithm initially attempts to use the largest context. The size of the largest context is predetermined. If the symbol to be encoded has not previously been encountered in this context, an escape symbol is encoded and the algorithm attempts to use the next smaller context. If the symbol has not occurred in this context either, the size of the context is

further reduced. This process continues until either we obtain a context that has previously been encountered with this symbol or we arrive at the conclusion that the symbol has not been encountered previously in *any* context. In this case, we use probability of 1/M to encode the symbol, where M is the size of the source alphabet. For example, when coding the *a* of probability,

T A B L E 6 . 1 Count array for −1 order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	1	3
s	1	4
e	1	5
b/	1	6
Total_Count		6

we would first attempt to see if the string *proba* has previously occurred that is, if *a* had previously occurred in the context of *prob*. If not, we would encode an escape and see if *a* had occurred in the context of *rob*. If the string *roba* had not occurred previously, we would again send an escape symbol and try the context *ob*. Continuing in this manner, we would try the context *b*, and failing that, we would see if the letter *a* (with a zero-order context) had occurred previously. If *a* was being encountered for the first time, we would use a model in which all letters occur with equal probability to encode *a*. This equiprobable model is sometimes referred to as the context of order −1. As the development of the probabilities with respect to each context is an adaptive process, the count corresponding to a symbol is updated each time the symbol is encountered. The number of counts to be assigned to the escape symbol is not obvious, and a number of different approaches have been used. One approach used by Cleary and Witten is to give the escape symbol a count of one, thus inflating the total count by one. Cleary and Witten call this method of assigning counts Method A and the resulting algorithm *ppma*. We will describe some of the other ways of assigning counts to the escape symbol later in this section.

Before we delve into some of the details, let's work through an example to see how all this works together. As we will be using arithmetic coding to encode the symbols, you might wish to refresh your memory of the arithmetic coding algorithms.

Example : Let's encode the sequence ***thisbis/bthe/btithe/***

Assuming we have already encoded the initial seven characters *thisbis/*, the various counts and *Cum_Count* arrays to be used in the arithmetic coding of the symbols are shown in Tables 6.1–6.4. In this example, we are assuming that the longest context length is two. This is a rather small value and is used here to keep the size of the example reasonably small. A more common value for the longest context length is five.

We will assume that the word length for arithmetic coding is six. Thus, $l = 000000$ and $u = 111111$. As *thisbis/* has already been encoded, the next letter to be encoded is *b/*. The second-order context for this letter is *is*. Looking at Table 6.4, we can see that the letter *b/* is the first letter in this context with a *Cum_Count* value of 1. As the *T otal_Count* in this case is 2, the update equations for the lower and upper limits are

$$l = 0 + (63 - 0 + 1) \times \frac{0}{2} = 0 = 000000$$

$$u = 0 + (63 - 0 + 1) \times \frac{1}{2} - 1 = 31 = 011111$$

As the MSBs of both *l* and *u* are the same, we shift that bit out and shift a 0 into the LSB of *l* and a 1

into the LSB of u. The transmitted sequence, lower limit, and upper limit after the update are

TABLE 6.2 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b/	1	7
*_Esc_*	1	8
Total_Count		8

Transmitted sequence : 0

l : 000000 u : 111111

We also update the counts in Tables 6.2–6.4.

The next letter to be encoded in the sequence is *t*. The second-order context is *sb/*. Looking at Table 6.4, we can see that *t* has not appeared before in this context. We therefore encode an escape symbol. Using the counts listed in Table 6.4, we update the lower and upper limits:

$$l = 0 + (63 - 0 + 1) \times \frac{1}{2} = 32 = 100000$$
$$u = 0 + (63 - 0 + 1) \times 2 - 1 = 63 = 111111$$

Again, the MSBs of l and u are the same, so we shift the bit out and shift 0 into the LSB of l and 1 into u, restoring l to a value of 0 and u to a value of 63. The transmitted sequence is now 01. After transmitting the escape, we look at the first-order context of *t*, which is *b/*. Looking at Table 6.3, we can see that *t* has not previously occurred in this context. To let the decoder know this, we transmit another escape. Updating the limits, we get

$$u = 0 + (63 - 0 + 1) \times 2 - 1 = 63/2 = 111111$$

TABLE 6.3 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	1	1
	*_Esc_*	1	2
	Total_Count		2
h	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
i	*s*	2	2
	*_Esc_*	1	3
	Total_Count		3
b/	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
s	*b/*	1	1
	*_Esc_*	1	2
	Total_Count		2

Table 6.4 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
hi	*s*	1	1
	*_Esc_*	1	2
	Total_Count		2
is	*b/*	1	1
	*_Esc_*	1	2
	Total_Count		2
sb/	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
bi/	*s*	1	1
	*_Esc_*	1	2
	Total_Count		2

TABLE 6.5 Updated count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b/	2	8
*_Esc_*	1	9
Total_Count		9

As the MSBs of l and u are the same, we shift the MSB out and shift 0 into the LSB of l and 1 into the LSB of u. The transmitted sequence is now 011. Having escaped out of the first-order contexts, we examine Table 6.5, the updated version of Table 6.2, to see if we can encode t using a zero-order context. Indeed we can, and using the *Cum_Count* array, we can update l and u:

$$l = 0 + (63 - 0 + 1) \times \frac{0}{9} = 0 = 000000$$

$$u = 0 + (63 - 0 + 1) \times \frac{1}{9} - 1 = 6 = 000110$$

The three most significant bits of both l and u are the same, so we shift them out. After the update we get

Transmitted sequence: 011000 l : 000000 u : 110111

The next letter to be encoded is h. The second-order context *bt/* has not occurred previously, so we move directly to the first-order context t. The letter h has occurred previously in this context, so we update l and u and obtain

Transmitted sequence : 0110000

l : 000000

u : 110101

The method of encoding should now be clear. At this point the various counts are as shown in Tables 6.6–6.8.Now that we have an idea of how the *ppm* algorithm works, let's examine some of the variations.

TABLE 6.6 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	2	2
h	2	4
i	2	6
s	2	8
b/	2	10
*_Esc_*	1	11
Total_Count		11

TABLE 6.7 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	2	2
	*_Esc_*	1	3
	Total_Count		3
h	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
i	*s*	2	2
	*_Esc_*	1	3
	Total_Count		3
b/	*i*	1	1
	t	1	2
	*_Esc_*	1	3
	Total_Count		3
s	*b/*	2	2
	*_Esc_*	1	3
	Total_Count		3

(2) The Escape Symbol : - In our example we used a count of one for the escape symbol, thus inflating the total count in each context by one. Cleary and Witten call this Method A, and the corresponding algorithm is referred to as *ppma*. There is really no obvious justification for assigning a count of one to the escape symbol. For that matter, there is no obvious method of assigning counts to the escape symbol. There have been various methods reported in the literature.

Another method described by Cleary and Witten is to reduce the counts of each symbol by one and assign these counts to the escape symbol. For example, suppose in a given sequence *a* occurs 10 times in the context of *prob*, *l* occurs 9 times, and *o* occurs 3 times in the same

TABLE 6.8 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	*_Esc_*	1	2
	Total_Count		2
hi	*s*	1	1
	*_Esc_*	1	2
	Total_Count		2
is	*b/*	2	2
	*_Esc_*	1	3

	Total_Count		3
sb/	*i*	1	1
	t	1	2
	*_Esc_*	1	3
	Total_Count		3
bi/	*s*	1	1
	*_Esc_*	1	2
	Total_Count		2
bt/	*h*	1	1
	*_Esc_*	1	2
	Total_Count		2

TABLE 6.9 Counts using Method A.

Context	Symbol	Count
prob	*a*	10
	l	09
	o	03
	*_Esc_*	01
	Total_Count	23

context (e.g., *problem, proboscis*, etc.). In Method A, we assign a count of one to the escape symbol, resulting in a total count of 23, which is one more than the number of times *prob* has occurred. This situation is shown in Table 6.9.

In this second method, known as Method B, we reduce the count of each of the symbols *a, l*, and *o* by one and give the escape symbol a count of three, resulting in the counts shown in Table 6.10.

The reasoning behind this approach is that if in a particular context more symbols can occur, there is a greater likelihood that there is a symbol in this context that has not occurred

TABLE 6.10 Counts using Method B.

Context	Symbol	Count
prob	*a*	09
	l	08
	o	02
	*_Esc_*	03
	Total_Count	22

TABLE 6.11 Counts using Method C.

Context	Symbol	Count
prob	*a*	10
	l	09
	o	03
	*_Esc_*	03
	Total_Count	25

before. This increases the likelihood that the escape symbol will be used. Therefore, we should assign a higher probability to the escape symbol.

A variant of Method B, appropriately named Method C, was proposed by Moffat. In Method C, the count assigned to the escape symbol is the number of symbols that have occurred in that context. In this respect, Method C is similar to Method B. The difference comes in the fact that, instead of "robbing" this from the counts of individual symbols, the total count is inflated by this amount. This situation is shown in Table 6.11. While there is some variation in performance depending on the characteristics of the data being encoded, Method C, on average, seems to provide the best

performance of the three methods for assigning counts to the escape symbol.

(3) Length of Context : - It would seem that as far as the maximum length of the contexts is concerned, more is better. However, this is not necessarily true. A longer maximum length will usually result in a higher probability if the symbol to be encoded has a non zero count with respect to that context. How-ever, a long maximum length also means a higher probability of long sequences of escapes, which in turn can increase the number of bits used to encode the sequence. If we plot the compression performance versus maximum context length, we see an initial sharp increase in performance until some value of the maximum length, followed by a steady drop as the maximum length is further increased. The value at which we see a downturn in performance changes depending on the characteristics of the source sequence.

(4) The Exclusion Principle : - The basic idea behind arithmetic coding is the division of the unit interval into subintervals, each of which represents a particular letter. The smaller the subinterval, the more bits are required to distinguish it from other subintervals. If we can reduce the number of symbols to be represented, the number of subintervals goes down as well. This in turn means that the sizes of the subintervals increase, leading to a reduction in the number of bits required for encoding. The exclusion principle used in *ppm* provides this kind of reduction in rate. Suppose we have been compressing a text sequence and come upon the sequence *proba*, and suppose we are trying to encode the letter *a*. Suppose also that the state of the two-letter context *ob* and the one-letter context *b* are as shown in Table 6.12.

First, we attempt to encode *a* with the two-letter context. As *a* does not occur in this context, we issue an escape symbol and reduce the size of the context. Looking at the table for the one-letter context *b*, we see that *a* does occur in this context with a count of 4 out of a total possible count of 21. Notice that other letters in this context include *l* and *o*. However, by sending the escape symbol in the context of *ob*, we have already signalled to the decoder that the symbol being encoded is not any of the letters that have previously been encountered in the context of *ob*. Therefore, we can increase the size of the subinterval corresponding to *a* by temporarily removing *l* and *o* from the table. Instead of using Table 6.12, we use Table 6.13 to encode *a*. This exclusion of symbols from contexts on a temporary basis can result in cumulatively significant savings in terms of rate.

TABLE 6.12 Counts for exclusion example.

Context	Symbol	Count
ob	l	10
	o	03
	*_Esc_*	02
Total_Count		15
b	l	05
	o	03
	a	04
	r	02
	e	02
	*_Esc_*	05
Total_Count		21

TABLE 6.13 Modified table used for exclusion example.

Context	Symbol	Count
b	*a*	4
	r	2
	e	2
	*_Esc_*	3
Total_Count		11

You may have noticed that we keep talking about small but significant savings. In lossless compression schemes, there is usually a basic principle, such as the idea of prediction with partial match, followed by a host of relatively small modifications. The importance of these modifications should not be underestimated, because together they often provide the margin of compression that makes a particular scheme competitive.

The Burrows Wheeler Transform : - The Burrows-Wheeler transform (BWT) algorithm also uses the context of the symbol being encoded, but in a very different way, for lossless compression. The transform that is a major part of this algorithm was developed by Wheeler in 1983. However, the BWT compression algorithm, which uses this transform, saw the light of day in 1994 [77]. Unlike most of the previous algorithms we have looked at, the BWT algorithm requires that the entire sequence to be coded be available to the encoder before the coding takes place. Also, unlike most of the previous algorithms, the decoding procedure is not immediately evident once we know the encoding procedure. We will first describe the encoding procedure. If it is not clear how this particular encoding can be reversed, bear with us and we will get to it.

The algorithm can be summarized as follows. Given a sequence of length N , we create $N - 1$ other sequences where each of these $N - 1$ sequences is a cyclic shift of the original sequence. These N sequences are arranged in lexicographic order. The encoder then transmits the sequence of length N created by taking the last letter of each sorted, cyclically shifted, sequence. This sequence of last letters L, and the position of the original sequence in the sorted list, are coded and sent to the decoder. As we shall see, this information is sufficient to recover the original sequence.

We start with a sequence of length N and end with a representation that contains $N + 1$ elements. However, this sequence has a structure that makes it highly amenable to compression. In particular we will use a method of coding called move to front (*mtf*), which is particularly effective on the type of structure exhibited by the sequence L.

Before we describe the *mtf* approach, let us work through an example to generate the L sequence.

Example : Let's encode the sequence

thisbis/bthe/

We start with all the cyclic permutations of this sequence. As there are a total of 11 characters, there are 11 permutations, shown in Table 6.14.

Now let's sort these sequences in lexicographic (dictionary) order (Table 6.15). The se-quence of the last letters L in this case is

L : *sshtthbii/be/*

Notice how like letters have come together. If we had a longer sequence of letters, the *runs* of like letters would have been even longer. The *mtf* algorithm, which we will describe later, takes advantage of these runs.

The original sequence appears as sequence number 10 in the sorted list, so the encoding of the sequence consists of the sequence L and the index value 10.

Now that we have an encoding of the sequence, let's see how we can decode the original sequence by using the sequence L and the index to the original sequence in the sorted list. The important thing to note is that all the elements of the initial sequence are contained in L. We just need to figure out the permutation that will let us recover the original sequence.

The first step in obtaining the permutation is to generate the sequence F consisting of the first element of each row. That is simple to do because we lexicographically ordered the sequences. Therefore, the sequence F is simply the sequence L in lexicographic order. In our example this means that F is given as

F : *b/behhiisstt/*

We can use *L* and *F* to generate the original sequence. Look at Table 6.15 containing the cyclically shifted sequences sorted in lexicographic order. Because each row is a cyclical shift, the letter in the first column of any row is the letter appearing after the last column in the row in the original sequence. If we know that the original sequence is in the *k*th row, then we can begin unraveling the original sequence starting with the *k*th element of *F*.

T A B L E 6 . 14 Permutations of this/bis/bthe.

00	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*
01	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*
02	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*
03	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*
04	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*
05	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*
06	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*
07	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*
08	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*
09	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*
10	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*

T A B L E 6 . 15 Sequences sorted into lexicographic order.

0	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*
1	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*
2	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*
3	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*
4	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*
5	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*
6	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*
7	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*
8	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*
9	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*
10	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*

the original sequence is sequence number 10, so the first letter in the original sequence is $F[10] = t$. To find the letter following *t*, we look for *t* in the array *L*. There are two *t*'s in *L*. Which should we use? The *t* in *F* that we are working with is the lower of two *t*'s, so we pick the lower of two *t*'s in *L*. This is *L*[4]. Therefore, the next letter in our reconstructed sequence is $F[4] = h$. The reconstructed sequence to this point is *th*. To find the next letter, we look for *h* in the *L* array. Again there are two *h*'s. The *h* at *F*[4] is the lower of two *h*'s in *F*, so we pick the lower of the two *h*'s in *L*. This is the fifth element of *L*, so the next element in our decoded sequence is $F[5] = i$.
The decoded sequence to this point is *thi*. The process continues as depicted in Figure 6.1 to generate the original sequence.

Why go through all this trouble? After all, we are going from a sequence of length *N* to another sequence of length *N* plus an index value. It appears that we are actually causing expansion instead of compression.

The answer is that the sequence *L* can be compressed much more efficiently than the original sequence. Even in our small example, we have runs of like symbols. This will happen a lot more

when N is large. Consider a large sample of text that has been cyclically shifted and sorted in a list A. Consider all the rows of A beginning with

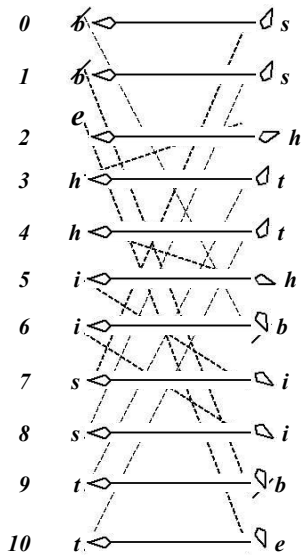


FIGURE 6.1 Decoding process.

heb/. With high probability, *heb/* would be preceded by *t*. Therefore, in L we would get a long run of $\bar{t}$s.

Move- to- Front Coding:- A coding scheme that takes advantage of long runs of identical symbols is move-to-front (*mtf*) coding. In this coding scheme, we start with some initial listing of the source alphabet. The symbol at the top of the list is assigned the number 0, the next one is assigned the number 1, and so on. The first time a particular symbol occurs, the number corresponding to its place in the list is transmitted. Then it is moved to the top of the list. If we have a run of this symbol, we transmit a sequence of 0s. This way, long runs of different symbols get transformed to a large number of 0s. Applying this technique to our example does not produce very impressive results due to the small size of the sequence, but we can see how the technique functions.

Example :

Let's encode L = *sshtthbii/be/*. Let's assume that the source alphabet is given by

$$A = \{b/, e, h, i, s, t\}$$

We start out with the assignment

0	1	2	3	4	5
b/	*e*	*h*	*i*	*s*	*t*

The first element of L is *s*, which gets encoded as a 4. We then move *s* to the top of the list, which gives us

0	1	2	3	4	5
s	*b/*	*e*	*h*	*i*	*t*

The next *s* is encoded as 0. Because *s* is already at the top of the list, we do not need to make any changes. The next letter is *h*, which we encode as 3. We then move *h* to the top of the list:

0	1	2	3	4	5
h	*s*	*b/*	*e*	*i*	*t*

The next letter is *t*, which gets encoded as 5. Moving *t* to the top of the list, we get

0	1	2	3	4	5
t	*h*	*s*	*b/*	*e*	*i*

The next letter is also a *t*, so that gets encoded as a 0. Continuing in this fashion, we get the sequence

4 0 3 5 0 1 3 5 0 1 5

As we warned, the results are not too impressive with this small sequence, but we can see how we would get large numbers of 0s and small values if the sequence to be encoded was longer. These long sequences of 0s could then be efficiently encoded using schemes specifically designed for encoding long repetitions of the same symbol, such as run-length coding.

CALIC:- The Context Adaptive Lossless Image Compression (CALIC) scheme, which came into being in response to a call for proposals for a new lossless image compression scheme in 1994, uses both context and prediction of the pixel values. The CALIC scheme actually functions in two modes, one for grayscale images and another for bi-level images. In this section, we will concentrate on the compression of grayscale images.

In an image, a given pixel generally has a value close to one of its neighbors. Which neighbor has the closest value depends on the local structure of the image. Depending on whether there is a horizontal or vertical edge in the neighborhood of the pixel being encoded, the pixel above, or the pixel to the left, or some weighted average of neighboring pixels may give the best prediction. How close the prediction is to the pixel being encoded depends on the surrounding texture. In a region of the image with a great deal of variability, the prediction is likely to be further from the pixel being encoded than in the regions with less variability.

In order to take into account all of these factors, the algorithm has to make a determina-tion about the environment of the pixel to be encoded. The information used to make this determination has to be available to both the encoder and decoder.

Let's take up the question of the presence of vertical or horizontal edges in the neighborhood of the pixel being encoded. To help our discussion, we will refer to Figure 7.1. In this figure, the pixel to be encoded has been marked with an *X*. The pixel above is called the north pixel, the pixel to the left is the west pixel, and so on. Note that when pixel *X* is being encoded, all other marked pixels (*N, W, NW, NE, WW, NN, NE,* and *NNE*) are available to both the encoder and decoder.

		NN	*NNE*
	NW	*N*	*NE*
WW	*W*	*X*	

We can get an idea of what kinds of boundaries may or may not be in the neighborhood of *X* by computing

$$d_h = |W - WW| + |N - NW| + |NE - N|$$
$$d_v = |W - NW| + |N - NN| + |NE - NNE|.$$

The relative values of d_h and d_v are used to obtain the initial prediction of the pixel *X*. This initial prediction is then refined by taking other factors into account. If the value of d_h is much higher than the value of d_v, this will mean there is a large amount of horizontal variation, and it would be better to pick *N* to be the initial prediction. If, on the other hand, d_v is much larger than d_h, this would mean that there is a large amount of vertical variation, and the initial prediction is

taken to be W. If the differences are more moderate or smaller, the predicted value is a weighted average of the neighboring pixels.

The exact algorithm used by CALIC to form the initial prediction is given by the following pseudocode:

```
if d_h − d_V > 80
  X ← N
else if d_V − d_h > 80
  X ← W
else
{
    X ← (N + W)/2 + (NE − NW)/4
    if d_h − d_V > 32
                        X ← (X + N)/2
                    else if d_V − d_h > 32
                        X ← (X + W)/2
                    else if d_h − d_V > 8
      X ← (3X + N)/4
    else if d_V − d_h > 8
      X ← (3X + W)/4
}
```

Using the information about whether the pixel values are changing by large or small amounts in the vertical or horizontal direction in the neighborhood of the pixel being encoded results in a good initial prediction. In order to refine this prediction, we need some information about the interrelationships of the pixels in the neighborhood. Using this information, we can generate an offset or refinement to our initial prediction. We quantify the information about the neighborhood by first forming the vector

$$[\ N,\ W,\ NW,\ NE,\ NN,\ WW,\ 2N - NN,\ 2W - WW\ ]$$

We then compare each component of this vector with our initial prediction X. If the value of the component is less than the prediction, we replace the value with a 1; otherwise we replace it with a 0. Thus, we end up with an eight-component binary vector. If each component of the binary vector was independent, we would end up with 256 possible vectors. However, because of the dependence of various components, we actually have 144 possible configurations. We also compute a quantity that incorporates the vertical and horizontal variations and the previous error in prediction by

$$\delta = d_h + d_v + \frac{e}{2} N - N \qquad (8)$$

where N is the predicted value of N. This range of values of δ is divided into four intervals, each represented by 2 bits. These four possibilities, along with the 144 texture descriptors, create $144 \times 4 = 576$ contexts for X. As the encoding proceeds, we keep track of how much prediction error is generated in each context and offset our initial prediction by that amount. This results in the final predicted value.

Once the prediction is obtained, the difference between the pixel value and the prediction (the prediction error or residual) has to be encoded. While the prediction process outlined above removes a lot of the structure that was in the original sequence, there is still some structure left in the residual sequence. We can take advantage of some of this structure by coding the residual in terms of its context. The context of the residual is taken to be the value of δ defined in Eq. (8). In order to reduce the complexity of the encoding, rather than using the actual value as the context, CALIC uses the range of values in which δ lies as the context. Thus:

$0 \quad \delta < q_1 \Rightarrow$ Context 1

$q_1 \quad \delta < q_2 \Rightarrow$ Context 2

$q_2 \quad \delta < q_3 \Rightarrow$ Context 3

q_3 $\delta < q_4 \Rightarrow$ Context 4
q_4 $\delta < q_5 \Rightarrow$ Context 5
q_5 $\delta < q_6 \Rightarrow$ Context 6
q_6 $\delta < q_7 \Rightarrow$ Context 7
q_7 $\delta < q_8 \Rightarrow$ Context 8

The values of q_1–q_8 can be prescribed by the user.

If the original pixel values lie between 0 and $M - 1$, the differences or prediction residuals will lie between $-(M - 1)$ and $M - 1$. Even though most of the differences will have a magnitude close to zero, for arithmetic coding we still have to assign a count to all possible symbols. This means a reduction in the size of the intervals assigned to values that do occur, which in turn means using a larger number of bits to represent these values. The CALIC algorithm attempts to resolve this problem in a number of ways. Let's describe these using an example.Consider the sequence

$$x_n : 0, 7, 4, 3, 5, 2, 1, 7$$

We can see that all of the numbers lie between 0 and 7, a range of values that would require 3 bits to represent. Now suppose we predict a sequence element by the previous element in the sequence. The sequence of differences

$$r_n = x_n - x_{n-1}$$

is given by

$$r_n : 0, 7, -3, -1, 2, -3, -1, 6$$

If we were given this sequence, we could easily recover the original sequence by using

$$x_n = x_{n-1} + r_n .$$

However, the prediction residual values r_n lie in the $[-7, 7]$ range. That is, the alphabet required to represent these values is almost twice the size of the original alphabet. However, if we look closely we can see that the value of r_n actually lies between $-x_{n-1}$ and $7 - x_{n-1}$. The smallest value that r_n can take on occurs when x_n has a value of 0, in which case r_n will have a value of $-x_{n-1}$. The largest value that r_n can take on occurs when x_n is 7, in which case r_n has a value of $7 - x_{n-1}$. In other words, given a particular value for x_{n-1}, the number of different values that r_n can take on is the same as the number of values that x_n can take on. Generalizing from this, we can see that if a pixel takes on values between 0 and $M - 1$, then given a predicted value X , the difference $X - \hat{X}$ will take on values in the range $-X$ to $M - 1 - X$. We can use this fact to map the difference values into the range $[0, M - 1]$, using the following mapping:

Another approach used by CALIC to reduce the size of its alphabet is to use a modification of a technique called *recursive indexing*] described in Chapter 9. Recursive indexing is a technique for representing a large range of numbers using only a small set. It is easiest to explain using an example. Suppose we want to represent positive integers using only the integers between 0 and 7—that is, a representation alphabet of size 8. Recursive indexing works as follows. If the number to be represented lies between 0 and 6, we simply represent it by that number. If the number to be represented is greater than or equal to 7, we first send the number 7, subtract 7 from the original number, and repeat the process. We keep repeating the process until the remainder is a number between 0 and 6. Thus, for example, 9 would be represented by 7 followed by a 2, and 17 would be represented by two 7s followed by a 3. The decoder, when it sees a number between 0 and 6, will decode it at its face value, and when it sees 7, it will keep accumulating the values until a value between 0 and 6 is received. This method of representation followed by entropy coding has been shown to be optimal for sequences that follow a geometric distribution [86].

In CALIC, the representation alphabet is different for different coding contexts. For each coding context k, we use an alphabet $A_k = \{0, 1, \ldots, N_k\}$. Furthermore, if the residual occurs in context k, then the first number that is transmitted is coded with respect to context k; if further recursion is needed, we use the $k + 1$ context.

We can summarize the CALIC algorithm as follows:

1. Find the initial prediction X .
2. Compute the prediction context.
3. Refine the prediction by removing the estimate of the bias in that context.
4. Update the bias estimate.
5. Obtain the residual and remap it so the residual values lie between 0 and $M - 1$, where M is the size of the initial alphabet.
6. Find the coding context k.
7. Code the residual using the coding context.

All these components working together have kept CALIC as the state of the art in lossless image compression. However, we can get almost as good a performance if we simplify some of the more involved aspects of CALIC.

JPEG-LS:-

The JPEG-LS standard looks more like CALIC than the old JPEG standard. When the initial proposals for the new lossless compression standard were compared, CALIC was rated first in six of the seven categories of images tested. Motivated by some aspects of CALIC, a team from Hewlett-Packard (HP) proposed a much simpler predictive coder, under the name LOCO-I (for low complexity), that still performed close to CALIC.

As in CALIC, the standard has both a lossless and a lossy mode. We will not describe the lossy coding procedures.

The initial prediction is obtained using the following algorithm:

```
if N W max(W, N )
X = max(W, N )
else
{
    if N W min(W, N )
    X = min(W, N )
    else
    X = W + N − N W
 }
```

This prediction approach is a variation of Median Adaptive Prediction [88], in which the predicted value is the median of the N , W , and NW pixels. The initial prediction is then refined using the average value of the prediction error in that particular context.

The contexts in JPEG-LS also reflect the local variations in pixel values. However, they are computed differently from CALIC. First, measures of differences D_1, D_2, and D_3 are computed as follows:

$$D_1 = N\,E - N$$
$$D_2 = N - N\,W$$
$$D_3 = N\,W - W.$$

The values of these differences define a three-component context vector **Q**. The components of **Q** (Q_1, Q_2, and Q_3) are defined by the following mappings:

$$D_i \quad -T_3 \Rightarrow \quad Q_i = -4$$
$$-T_3 < D_i - T_2 \Rightarrow \quad Q_i = -3$$

$$
\begin{aligned}
-T_2 < D_i - T_1 &\Rightarrow Q_i = -2 \\
-T_1 < D_i 0 &\Rightarrow Q_i = -1 \\
D_i = 0 &\Rightarrow Q_i = 0 \\
0 < D_i \quad T_1 &\Rightarrow Q_i = 1 \\
T_1 < D_i \quad T_2 &\Rightarrow Q_i = 2 \\
T_2 < D_i \quad T_3 &\Rightarrow Q_i = 3 \\
T_3 < D_i &\Rightarrow Q_i = 4
\end{aligned} \qquad (9)
$$

where T_1, T_2, and T_3 are positive coefficients that can be defined by the user. Given nine possible values for each component of the context vector, this results in $9 \times 9 \times 9 = 729$ possible contexts. In order to simplify the coding process, the number of contexts is reduced by replacing any context vector **Q** whose first nonzero element is negative by $-\mathbf{Q}$. Whenever this happens, a variable *SIGN* is also set to −1; otherwise, it is set to +1. This reduces the number of contexts to 365. The vector **Q** is then mapped into a number between 0 and 364. (The standard does not specify the particular mapping to use.)

The variable *SIGN* is used in the prediction refinement step. The correction is first multi-plied by *SIGN* and then added to the initial prediction.

The prediction error r_n is mapped into an interval that is the same size as the range occupied by the original pixel values. The mapping used in JPEG-LS is as follows:

$$r_n < -\frac{M}{} \Rightarrow r_n \leftarrow r_n + M$$

$$r_n > \frac{M}{} \Rightarrow r_n \leftarrow r_n - M$$

Finally, the prediction errors are encoded using adaptively selected codes based on Golomb codes, which have also been shown to be optimal for sequences with a geometric distribution. In Table 7.3, we compare the performance of the old and new JPEG standards and CALIC. The results for the new JPEG scheme were obtained using a software implementation courtesy of HP. We can see that for most of the images, the new JPEG standard performs very close to CALIC and outperforms the old standard by 6% to 18%. The only case where the performance is not as good is for the Omaha image. While the performance improvement in these examples.

TABLE 7.3 Comparison of the file sizes obtained using the new and old JPEG lossless compression standards and CALIC.

Image	Old JPEG	New JPEG	CALIC
Sena	31,055	27,339	26,433
Sensin	32,429	30,344	29,213
Earth	32,137	26,088	25,280
Omaha	48,818	50,765	48,249

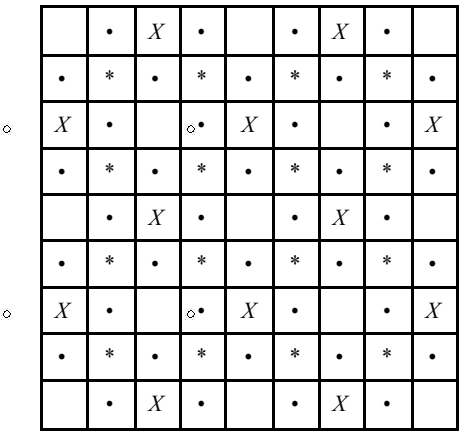
may not be very impressive, we should keep in mind that we are picking the best result out of eight for the old JPEG. In practice, this would mean trying all eight JPEG predictors and picking the best. On the other hand, both CALIC and the new JPEG standard are single-pass algorithms. Furthermore, because of the ability of both CALIC and the new standard to function in multiple modes, both perform very well on compound documents, which may contain images along with text.

Multiresolution Approaches:- Our final predictive image compression scheme is perhaps not as competitive as the other schemes. However, it is an interesting algorithm because it approaches the problem from a slightly different point of view.

Multiresolution models generate representations of an image with varying spatial resolu-tion. This usually results in a pyramid-like representation of the image, with each layer of the pyramid serving as a prediction model for the layer immediately below.

One of the more popular of these techniques is known as HINT (Hierarchical INTerpola-tion). The specific steps involved in HINT are as follows. First, residuals corresponding to the pixels labeled in Figure 7.2 are obtained using linear prediction, and they are trans-mitted. Then, the intermediate pixels *(◦)* are estimated by linear interpolation, and the error in estimation is then transmitted. Then, the pixels *X* are estimated from and ◦, and the estimation error is transmitted. Finally, the pixels labeled ⋆ and then • are estimated from known neighbors, and the errors are transmitted. The reconstruction process proceeds in a similar manner. One use of a multiresolution approach is in progressive image transmission.

Progressive Image Transmission:- The last few years have seen a very rapid increase in the amount of information stored as images, especially remotely sensed images (such as images from weather and other satellites) and medical images (such as computerized axial tomography (CAT) scans, magnetic resonance



F I G U R E 7 . 2 The HINT scheme for hierarchical prediction.

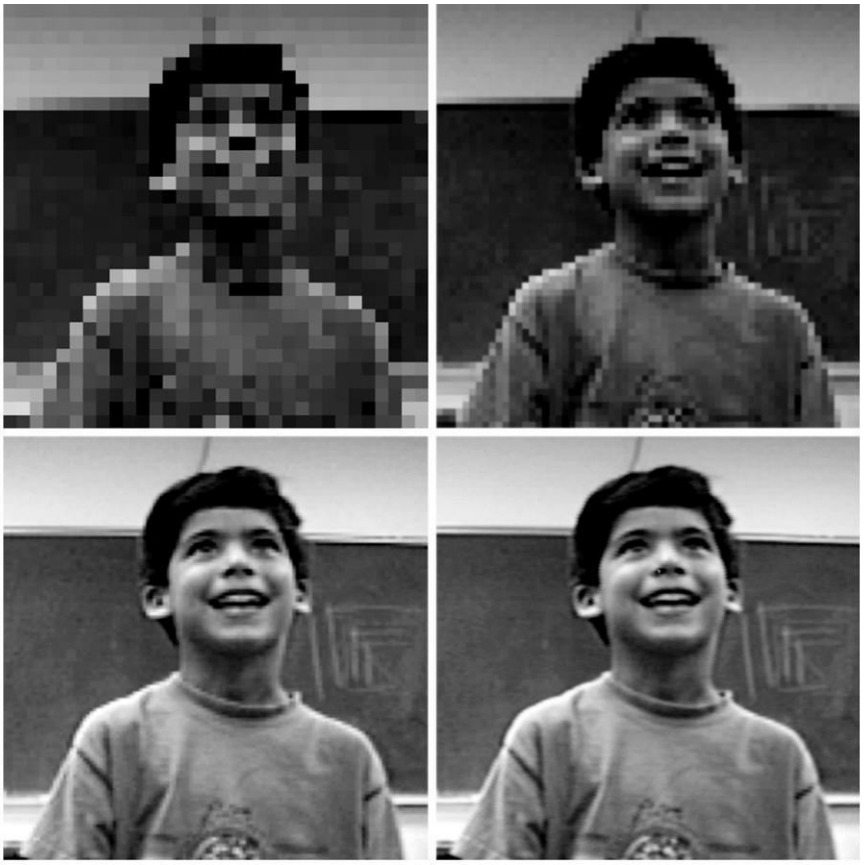
images (MRI), and mammograms). It is not enough to have information. We also need to make these images accessible to individuals who can make use of them. There are many issues involved with making large amounts of information accessible to a large number of people. In this section, we will look at one particular issue—transmitting these images to remote users. (For a more general look at the problem of managing large amounts of information.

Suppose a user wants to browse through a number of images in a remote database. The user is connected to the database via a 1 Mbps link. Suppose the size of the images is 2048 × 1536, and on average, users have to look through 30 images before finding the image for which they are looking. If these images were monochrome with 8 bits per pixel, this process would take close to 15 minutes, which is not very practical. Even if we compressed these images before transmission, lossless compression, on average, gives us about a 2:1 compression. This would only cut the transmission in half, which still makes the approach cumbersome. A better alternative is to send an approximation of each image first, which does not require too many bits but is still sufficiently accurate to give users an idea of what the image looks like. If users find the image to be of interest, they can request a further refinement of the approximation or the complete image. This approach is called *progressive image transmission*.

E x a m p l e : A simple progressive transmission scheme is to divide the image into blocks and then send a representative pixel for the block. The receiver replaces each pixel in the block with the representative value. In this example, the representative value is the value of the pixel in the top-left

corner. Depending on the size of the block, the amount of data that would need to be transmitted could be substantially reduced. For example, to transmit a 2048 × 1536 image at 8 bits per pixel over a 1 Mbps link takes about 25 seconds. Using a block size of 8 × 8 and using the top-left pixel in each block as the representative value, we can approximate the 1024× 1024 image with a 128 × 128 subsampled image. Using 8 bits per pixel and a 1 Mbps link, the time required to transmit this approximation to the image takes less than a third of a second.

Assuming that this approximation was sufficient to let the user decide whether a particular image was the desired image, the time required now to look through 30 images becomes on the order of seconds instead of the 15 minutes mentioned earlier. If the approximation using a block size of 8 × 8 does not provide enough resolution to make a decision, the user can ask for a refinement. The transmitter can then divide the 8 × 8 block into four 4 × 4 blocks. The pixel at the upper-left corner of the upper-left block was already transmitted as the representative pixel for the 8 × 8 block, so we need to send three more pixels for the other three 4 × 4 blocks. This takes a little more than a second, so even if the user had to request a finer approximation every third image, this would only increase the total search time by a very small amount. To see what these approximations look like, we have taken the Sena image and encoded it using different block sizes. The results are shown in Figure 7.3. The lowest-resolution image, shown in the top left, is a 32 × 32 image. The top-right image is a 64 × 64 image. The bottom-left image is a 128 × 128 image, and the bottom-right image is the 256 × 256 original.



F I G U R E Sena image coded using different block sizes for progressive trans-mission. Top row: block size 8 × 8 and block size 4 × 4. Bottom row: block size 2 × 2 and original image.

Notice that even with a block size of 8, the image is clearly recognizable as a person. Therefore, if the user was looking for a house, they would probably skip over this image after seeing the

first approximation. If the user was looking for a picture of a person, they could still make decisions based on the second approximation.

Finally, when an image is built line by line, the eye tends to follow the scan line. With the progressive transmission approach, the user gets a more global view of the image very early in the image formation process. Consider the images in Figure 7.4. The images on the left are the 8 × 8, 4 × 4, and 2 × 2 approximations of the Sena image. On the right, we show how much of the image we would see in the same amount of time if we used the standard line-by-line raster scan order. We would like the first approximations that we transmit to use as few bits as possible yet be accurate enough to allow the user to make a decision to accept or reject the image with a certain degree of confidence. As these approximations are lossy, many progressive transmission schemes use well-known lossy compression schemes in the first pass.

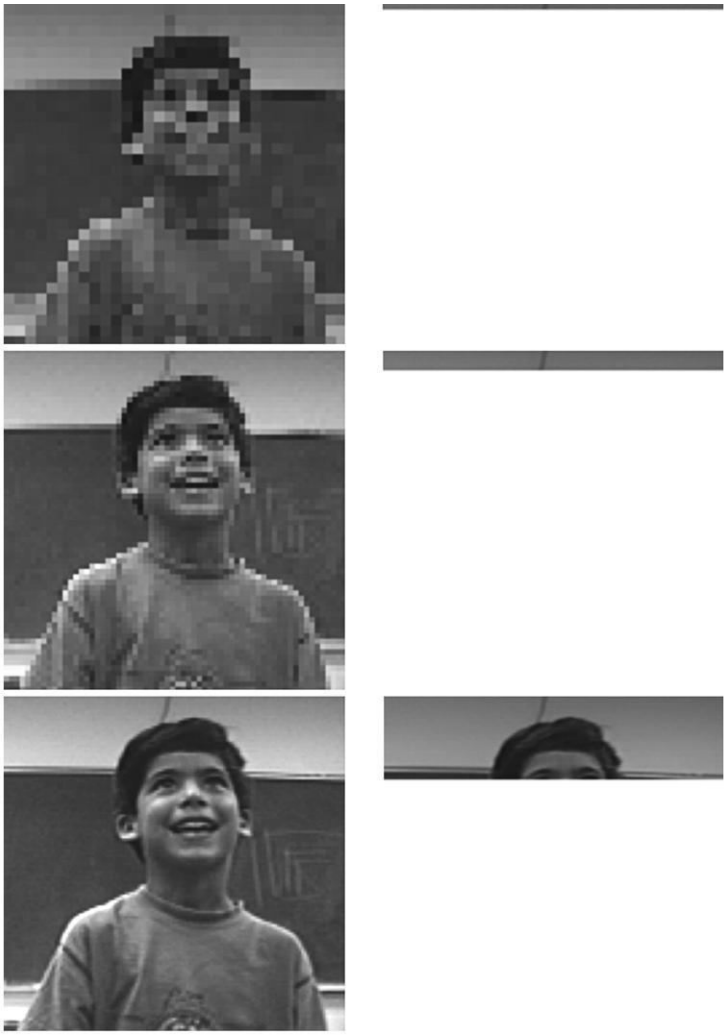
The more popular lossy compression schemes, such as transform coding, tend to require a significant amount of computation. As the decoders for most progressive transmission schemes have to function on a wide variety of platforms, they are generally implemented in software and need to be simple and fast. This requirement has led to the development of a number of progressive transmission schemes that do not use lossy compression for their initial approximations. Most of these schemes have a form similar to the one described in Example 7.6.1, and they are generally referred to as *pyramid schemes* because of the manner in which the approximations are generated and the image is reconstructed.

When we use the pyramid form, we still have a number of ways to generate the approximations. One of the problems with the simple approach described in Example 7.6.1 is that if the pixel values vary a lot within a block, the "representative" value may not be very representa-tive. To prevent this from happening, we could represent the block by some sort of an average or composite value. For example, suppose we start out with a 512 × 512 image. We first divide the image into 2 × 2 blocks and compute the integer value of the average of each block]. The integer values of the averages would constitute the penultimate approximation. The approximation to be transmitted prior to that can be obtained by taking the average of 2 × 2 averages and so on, as shown in Figure 7.5.

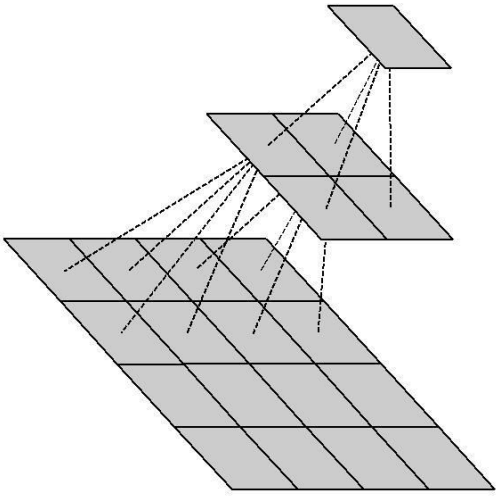
Using the simple technique in Example 7.6.1, we ended up transmitting the same number of values as the original number of pixels. However, when we use the mean of the pixels as our approximation, after we have transmitted the mean values at each level, we still have to transmit the actual pixel values. The reason is that when we take the integer part of the average, we end up throwing away information that cannot be retrieved. To avoid this problem of data expansion, we can transmit the sum of the values in the 2 × 2 block. Then we only need to transmit three more values to recover the original four values. With this approach, although we would be transmitting the same number of values as the number of pixels in the image, we might still end up sending more bits because representing all possible values of the sum would require transmitting 2 more bits than was required for the original value.

For example, if the pixels in the image can take on values between 0 and 255, which can be represented by 8 bits, their sum will take on values between 0 and 1024, which would require 10 bits. If we are allowed to use entropy coding, we can remove the problem of data expansion by using the fact that the neighboring values in each approximation are heavily correlated, as are values in different levels of the pyramid.

This means that differences between these values can be efficiently encoded using entropy coding. By doing so, we end up getting compression instead of expansion.



F I G U R E 7 . 4 Comparison between the received image using progressive transmis-sion and using the standard raster scan order.



F I G U R E 7 . 5 The pyramid structure for progressive transmission.

Instead of taking the arithmetic average, we could also form some sort of weighted average. The general procedure would be similar to that described above. (For one of the more well-known weighted average techniques).

The representative value does not have to be an average. We could use the pixel values in the approximation at the lower levels of the pyramid as indices into a lookup table. The lookup table could be designed to preserve important information such as edges. The problem with this approach would be the size of the lookup table. If we were using 2×2 blocks of 8-bit values, the lookup table would have 2^{32} values, which is too large for most applications. The size of the table could be reduced if the number of bits per pixel was lower or if, instead of taking 2×2 blocks, we used rectangular blocks of size 2×1 and 1×2 .

Finally, we do not have to build the pyramid one layer at a time. After sending the lowest-resolution approximations, we can use some measure of information contained in a block to decide whether it should be transmitted . One possible measure could be the difference between the largest and smallest intensity values in the block. Another might be to look at the maximum number of similar pixels in a block. Using an information measure to guide the progressive transmission of images allows the user to see portions of the image first that are visually more significant.

Facsimile Encoding:- One of the earliest applications of lossless compression in the modern era has been the com-pression of facsimile, or fax. In facsimile transmission, a page is scanned and converted into a sequence of black or white pixels. The requirements of how fast the facsimile of an A4 document (210 × 297 mm) must be transmitted have changed over the last two decades. The issued a number of recommendations based on the speed requirements at a given time. The CCITT classifies the apparatus for facsimile transmission into four groups. Although several considerations are used in this classification, if we only consider the time to transmit an A4-size document over phone lines, the four groups can be described as follows:

Group 1: This apparatus is capable of transmitting an A4-size document in about 6 minutes over phone lines using an analog scheme. The apparatus is standardized in recommendation T.2.

Group 2: This apparatus is capable of transmitting an A4-size document over phone lines in about 3 minutes. A Group 2 apparatus also uses an analog scheme and, therefore, does not use data compression. The apparatus is standardized in recommendation T.3.

Group 3: This apparatus uses a digitized binary representation of the facsimile. Because it is a digital scheme, it can and does use data compression and is capable of transmitting an A4-size document in about a minute. The apparatus is standardized in recommendation T.4.

Group 4: This apparatus has the same speed requirement as Group 3. The apparatus is standardized in recommendations T.6, T.503, T.521, and T.563.

With the arrival of the Internet, facsimile transmission has changed as well. Given the wide range of rates and “apparatus” used for digital communication, it makes sense to focus more on protocols than on apparatus. The newer recommendations from the ITU provide standards for compression that are more or less independent of apparatus.

Later in this chapter, we will look at the compression schemes described in ITU-T recommendations T.4, T.6, T.82 (JBIG), T.88 (JBIG2), and T.44 (MRC). We begin with a look at an earlier technique for facsimile called *run-length coding*, which still survives as part of the T.4 recommendation.