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 an ordering on the sequences because we will assign a tag to a particular sequence $\mathbf{x}_i$ as

$$\tilde{T}_{\mathbf{X}}^{(m)}(\mathbf{x}_i) = \sum_{\mathbf{y}<\mathbf{x}_i} P(\mathbf{y}) + \frac{1}{2}P(\mathbf{x}_i) \tag{4.4}$$

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.

Example 4.3.3:

We can extend Example 4.3.1 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 (4.4). For example, the tag for the sequence 13 would be

$$\begin{aligned}\tilde{T}_X(13) &= P(\mathbf{x}=11)+P(\mathbf{x}=12)+1/2P(\mathbf{x}=13) &(4.5)\\ &= 1/36+1/36+1/2(1/36) &(4.6)\\ &= 5/72. &(4.7)\end{aligned}$$

♦

Notice that to generate the tag for 13 we did not have to generate a tag for every other possible message. However, based on Equation (4.4) and Example 4.3.3, 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.

Recall that, given our construction, the interval containing the tag value for a given sequence is disjoint from the intervals containing the tag values of all other sequences. This means that any value in this interval would be a unique identifier for x_i. Therefore, to fulfill our initial objective of uniquely identifying each sequence, it would be sufficient to compute the upper and lower limits of the interval containing the tag and select any value in that interval. The upper and lower limits can be computed recursively as shown in the following example.

Example 4.3.4:

We will use the alphabet of Example 4.3.2 and find the upper and lower limits of the interval containing the tag for the sequence 322. Assume that we are observing 3 2 2 in a sequential manner; that is, first we see 3, then 2, and then 2 again. After each observation we will compute the upper and lower limits of the interval containing the tag of the sequence observed to that point. We will denote the upper limit by $u^{(n)}$ and the lower limit by $l^{(n)}$, where n denotes the length of the sequence.

We first observe 3. Therefore,

$$u^{(1)} = F_X(3), \qquad l^{(1)} = F_X(2).$$

We then observe 2 and the sequence is $\mathbf{x} = 32$. Therefore,

$$u^{(2)} = F_X^{(2)}(32), \qquad l^{(2)} = F_X^{(2)}(31).$$

We can compute these values as follows:

$$\begin{aligned} F_X^{(2)}(32) &= P(\mathbf{x} = 11) + P(\mathbf{x} = 12) + \cdots + P(\mathbf{x} = 16) \\ &\quad + P(\mathbf{x} = 21) + P(\mathbf{x} = 22) + \cdots + P(\mathbf{x} = 26) \\ &\quad + P(\mathbf{x} = 31) + P(\mathbf{x} = 32). \end{aligned}$$

But,

$$\sum_{i=1}^{i=6} P(\mathbf{x} = ki) = \sum_{i=1}^{i=6} P(x_1 = k, x_2 = i) = P(x_1 = k)$$

where $\mathbf{x} = x_1 x_2$. Therefore,

$$\begin{aligned} F_X^{(2)}(32) &= P(x_1 = 1) + P(x_1 = 2) + P(\mathbf{x} = 31) + P(\mathbf{x} = 32) \\ &= F_X(2) + P(\mathbf{x} = 31) + P(\mathbf{x} = 32). \end{aligned}$$

However, assuming each roll of the dice is independent of the others,

$$P(\mathbf{x} = 31) = P(x_1 = 3)P(x_2 = 1)$$

and

$$P(\mathbf{x} = 32) = P(x_1 = 3)P(x_2 = 2).$$

Therefore,

$$\begin{aligned} P(\mathbf{x}=31)+P(\mathbf{x}=32) &= P(x_1=3)(P(x_2=1)+P(x_2=2)) \\ &= P(x_1=3)F_X(2). \end{aligned}$$

Noting that

$$P(x_1=3)=F_X(3)-F_X(2)$$

we can write

$$P(\mathbf{x}=31)+P(\mathbf{x}=32)=(F_X(3)-F_X(2))F_X(2)$$

and

$$F_X^{(2)}(32)=F_X(2)+(F_X(3)-F_X(2))F_X(2).$$

We can also write this as

$$u^{(2)}=l^{(1)}+(u^{(1)}-l^{(1)})F_X(2).$$

We can similarly show that

$$F_X^{(2)}(31)=F_X(2)+(F_X(3)-F_X(2))F_X(1)$$

or

$$l^{(2)}=l^{(1)}+(u^{(1)}-l^{(1)})F_X(1).$$

The third element of the observed sequence is 2, and the sequence is $\mathbf{x}=322$. The upper and lower limits of the interval containing the tag for this sequence are

$$u^{(3)}=F_X^{(3)}(322), \qquad l^{(3)}=F_X^{(3)}(321).$$

Using the same approach as above we find that

$$\begin{aligned} F_X^{(3)}(322) &= F_X^{(2)}(31)+(F_X^{(2)}(32)-F_X^{(2)}(31))F_X(2) \\ F_X^{(3)}(321) &= F_X^{(2)}(31)+(F_X^{(2)}(32)-F_X^{(2)}(31))F_X(1) \end{aligned} \tag{4.8}$$

or

$$\begin{aligned} u^{(3)} &= l^{(2)}+(u^{(2)}-l^{(2)})F_X(2) \\ l^{(3)} &= l^{(2)}+(u^{(2)}-l^{(2)})F_X(1). \end{aligned}$$

♦

In general, we can show that for any sequence $\mathbf{x}=(x_1x_2\ldots x_n)$

$$l^{(n)} = l^{(n-1)}+(u^{(n-1)}-l^{(n-1)})F_X(x_n-1) \tag{4.9}$$

$$u^{(n)} = l^{(n-1)}+(u^{(n-1)}-l^{(n-1)})F_X(x_n). \tag{4.10}$$

Notice that throughout this process we did not explicitly need to compute any joint probabilities.

If we are using the midpoint of the interval for the tag, then

$$\tilde{T}_X(\mathbf{x}) = \frac{u^{(n)} + l^{(n)}}{2}.$$

Therefore, the tag for any sequence can be computed in a sequential fashion. The only information required by the tag generation procedure is the *cdf* of the source, which can be obtained directly from the probability model.

Example 4.3.5: Generating a tag

Consider the source in Example 3.2.4. Define the random variable $X(a_i) = i$. Suppose we wish to encode the sequence **1 3 2 1**. From the probability model we know that

$$F_X(k) = 0, \ k \leq 0, \quad F_X(1) = 0.8, \quad F_X(2) = 0.82, \quad F_X(3) = 1, \quad F_X(k) = 1, \ k > 3.$$

We can use Equations (4.9) and (4.10) sequentially to determine the lower and upper limits of the interval containing the tag. Initializing $u^{(0)}$ to 1, and $l^{(0)}$ to 0, the first element of the sequence **1** results in the following update:

$$\begin{aligned} l^{(1)} &= 0 + (1-0)0 = 0 \\ u^{(1)} &= 0 + (1-0)(0.8) = 0.8. \end{aligned}$$

That is, the tag is contained in the interval $[0, 0.8)$. The second element of the sequence is **3**. Using the update equations we get

$$\begin{aligned} l^{(2)} &= 0 + (0.8-0)F_X(2) = 0.8 \times 0.82 = 0.656 \\ u^{(2)} &= 0 + (0.8-0)F_X(3) = 0.8 \times 1.0 = 0.8. \end{aligned}$$

Therefore, the interval containing the tag for the sequence **1 3** is $[0.656, 0.8)$. The third element, **2**, results in the following update equations:

$$\begin{aligned} l^{(3)} &= 0.656 + (0.8-0.656)F_X(1) = 0.656 + 0.144 \times 0.8 = 0.7712 \\ u^{(3)} &= 0.656 + (0.8-0.656)F_X(2) = 0.656 + 0.144 \times 0.82 = 0.77408 \end{aligned}$$

and the interval for the tag is $[0.7712, 0.77408)$. Continuing with the last element, the upper and lower limits of the interval containing the tag are

$$\begin{aligned} l^{(4)} &= 0.7712 + (0.77408-0.7712)F_X(0) = 0.7712 + 0.00288 \times 0.0 = 0.7712 \\ u^{(4)} &= 0.7712 + (0.77408-0.1152)F_X(1) = 0.7712 + 0.00288 \times 0.8 = 0.773504 \end{aligned}$$

and the tag for the sequence **1 3 2 1** can be generated as

$$\tilde{T}_X(1321) = \frac{0.7712 + 0.773504}{2} = 0.772352.$$

♦

Notice that each succeeding interval is contained in the preceding interval. If we examine the equations used to generate the intervals, we see that this will always be the case. This property will be used to decipher the tag. An undesirable consequence of this process is that the intervals get smaller and smaller and require higher precision as the sequence gets longer. To combat this problem, a rescaling strategy needs to be adopted. In Section 4.4.2, we will describe a simple rescaling approach that takes care of this problem.

4.3.2 Deciphering the Tag

We have spent a considerable amount of time showing how a sequence can be assigned a unique tag, given a minimal amount of information. However, the tag is useless unless we can also decipher it with minimal computational cost. Fortunately, deciphering the tag is as simple as generating it. We can see this most easily through an example.

Example 4.3.6: Deciphering a tag

Given the tag obtained in Example 4.3.5, let's try to obtain the sequence represented by the tag. We will try to mimic the encoder in order to do the decoding. The tag value is 0.772352. The interval containing this tag value is a subset of every interval obtained in the encoding process. Our decoding strategy will be to decode the elements in the sequence in such a way that the upper and lower limits $u^{(k)}$ and $l^{(k)}$ will always contain the tag value for each k. We start with $l^{(0)} = 0$ and $u^{(0)} = 1$. After decoding the first element of the sequence x_1, the upper and lower limits become

$$\begin{aligned} l^{(1)} &= 0 + (1-0)F_X(x_1 - 1) = F_X(x_1 - 1) \\ u^{(1)} &= 0 + (1-0)F_X(x_1) = F_X(x_1). \end{aligned}$$

In other words, the interval containing the tag is $[F_X(x_1 - 1), F_X(x_1))$. We need to find the value of x_1 for which 0.772352 lies in the interval $[F_X(x_1 - 1), F_X(x_1))$. If we pick $x_1 = \mathbf{1}$, the interval is $[0, 0.8)$. If we pick $x_1 = \mathbf{2}$, the interval is $[0.8, 0.82)$, and if we pick $x_1 = \mathbf{3}$, the interval is $[0.82, 1.0)$. As 0.772352 lies in the interval $[0.0, 0.8]$, we choose $x_1 = \mathbf{1}$. We now repeat this procedure for the second element x_2, using the updated values of $l^{(1)}$ and $u^{(1)}$:

$$\begin{aligned} l^{(2)} &= 0 + (0.8-0)F_X(x_2 - 1) = 0.8F_X(x_2 - 1) \\ u^{(2)} &= 0 + (0.8-0)F_X(x_2) = 0.8F_X(x_2). \end{aligned}$$

If we pick $x_2 = \mathbf{1}$, the updated interval is $[0, 0.64)$, which does not contain the tag. Therefore, x_2 cannot be **1**. If we pick $x_2 = \mathbf{2}$, the updated interval is $[0.64, 0.656)$, which also does not contain the tag. If we pick $x_2 = \mathbf{3}$, the updated interval is $[0.656, 0.8)$, which does contain the tag value of 0.772352. Therefore, the second element in the sequence is **3**. Knowing the second element of the sequence, we can update the values of $l^{(2)}$ and $u^{(2)}$ and find the element x_3, which will give us an interval containing the tag:

$$\begin{aligned} l^{(3)} &= 0.656 + (0.8-0.656)F_X(x_3 - 1) = 0.656 + 0.144 \times F_X(x_3 - 1) \\ u^{(3)} &= 0.656 + (0.8-0.656)F_X(x_3) = 0.656 + 0.144 \times F_X(x_3). \end{aligned}$$

However, the expressions for $l^{(3)}$ and $u^{(3)}$ are cumbersome in this form. To make the comparisons more easily, we could subtract the value of $l^{(2)}$ from both the limits and the tag. That is, we find the value of x_3 for which the interval $[0.144 \times F_X(x_3 - 1), 0.144 \times F_X(x_3))$ contains $0.772352 - 0.656 = 0.116352$. Or, we could make this even simpler and divide the residual tag value of 0.116352 by 0.144 to get 0.808, and find the value of x_3 for which 0.808 falls in the interval $[F_X(x_3 - 1), F_X(x_3))$. We can see that the only value of x_3 for which this is possible is **2**. Substituting **2** for x_3 in the update equations, we can update the values of $l^{(3)}$ and $u^{(3)}$. We can now find the element x_4 by computing the upper and lower limits as

$$l^{(4)} = 0.7712 + (0.77408 - 0.7712)F_X(x_4 - 1) = 0.7712 + 0.00288 \times F_X(x_4 - 1)$$
$$u^{(4)} = 0.7712 + (0.77408 - 0.1152)F_X(x_4) = 0.7712 + 0.00288 \times F_X(x_4).$$

Again we can subtract $l^{(3)}$ from the tag to get $0.772352 - 0.7712 = 0.001152$ and find the value of x_4 for which the interval $[0.00288 \times F_X(x_4 - 1), 0.00288 \times F_X(x_4))$ contains 0.001152. To make the comparisons simpler, we can divide the residual value of the tag by 0.00288 to get 0.4, and find the value of x_4 for which 0.4 is contained in $[F_X(x_4 - 1), F_X(x_4))$. We can see that the value is $x_4 = \mathbf{1}$, and we have decoded the entire sequence. Note that we knew the length of the sequence beforehand and, therefore, we knew when to stop. ♦

From the example above, we can deduce an algorithm that can decipher the tag.

1. Initialize $l^{(0)} = 0$ and $u^{(0)} = 1$.
2. For each k find $t^* = (tag - l^{(k-1)})/(u^{(k-1)} - l^{(k-1)})$.
3. Find the value of x_k for which $F_X(x_k - 1) \leq t^* < F_X(x_k)$.
4. Update $u^{(k)}$ and $l^{(k)}$.
5. Continue until the entire sequence has been decoded.

There are two ways to know when the entire sequence has been decoded. The decoder may know the length of the sequence, in which case the deciphering process is stopped when that many symbols have been obtained. The second way to know if the entire sequence has been decoded is that a particular symbol is denoted as an end-of-transmission symbol. The decoding of this symbol would bring the decoding process to a close.

4.4 Generating a Binary Code

Using the algorithm described in the previous section, we can obtain a tag for a given sequence $\mathbf{x}$. 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 $\mathbf{x}$ 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? We will examine all these questions in the next section.

Even if we show the code to be unique and efficient, the method described to this point is highly impractical. In Section 4.4.2, we will describe a more practical algorithm for generating the arithmetic code for a sequence. We will give an integer implementation of this algorithm in Section 4.4.3.

4.4.1 Uniqueness and Efficiency of the Arithmetic Code

$\tilde{T}_X(x)$ is a number in the interval $[0, 1)$. A binary code for $\tilde{T}_X(x)$ can be obtained by taking the binary representation of this number and truncating it to $l(x) = \lceil \log \frac{1}{P(x)} \rceil + 1$ bits.

Example 4.4.1:

Consider a source $\mathcal{A}$ that generates letters from an alphabet of size four,

$$\mathcal{A} = \{a_1, a_2, a_3, a_4\}$$

with probabilities

$$P(a_1) = \frac{1}{2}, \quad P(a_2) = \frac{1}{4}, \quad P(a_3) = \frac{1}{8}, \quad P(a_4) = \frac{1}{8}.$$

A binary code for this source can be generated as shown in Table 4.4. The quantity $\tilde{T}_x$ is obtained using Equation (4.3). The binary representation of $\tilde{T}_x$ is truncated to $\lceil \log \frac{1}{P(x)} \rceil + 1$ bits to obtain the binary code.

TABLE 4.4 A binary code for a four-letter alphabet.

Symbol	F_X	$\tilde{T}_X$	In Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	Code
1	.5	.25	.010	2	01
2	.75	.625	.101	3	101
3	.875	.8125	.1101	4	1101
4	1.0	.9375	.1111	4	1111

♦

We will show that a code obtained in this fashion is a uniquely decodable code. We first show that this code is unique, and then we will show that it is uniquely decodable.

Recall that while we have been using $\tilde{T}_X(x)$ as the tag for a sequence $\mathbf{x}$, any number in the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ would be a unique identifier. Therefore, to show that the code $\lfloor \tilde{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is unique, all we need to do is show that it is contained in the interval

$[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$. Because we are truncating the binary representation of $\bar{T}_X(\mathbf{x})$ to obtain $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$, $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is less than or equal to $\bar{T}_X(\mathbf{x})$. More specifically,

$$0 \le \bar{T}_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} < \frac{1}{2^{l(\mathbf{x})}}. \tag{4.11}$$

As $\bar{T}_X(\mathbf{x})$ is strictly less than $F_X(\mathbf{x}))$,

$$\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} < F_X(\mathbf{x}).$$

To show that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} \ge F_X(\mathbf{x}-1)$, note that

$$\begin{aligned}
\frac{1}{2^{l(\mathbf{x})}} &= \frac{1}{2^{\lceil \log \frac{1}{P(\mathbf{x})} \rceil + 1}} \\
&< \frac{1}{2^{\log \frac{1}{P(\mathbf{x})} + 1}} \\
&= \frac{1}{2\frac{1}{P(\mathbf{x})}} \\
&= \frac{P(\mathbf{x})}{2}.
\end{aligned}$$

From (4.3) we have

$$\frac{P(\mathbf{x})}{2} = \bar{T}_X(\mathbf{x}) - F_X(\mathbf{x}-1).$$

Therefore,

$$\bar{T}_X(\mathbf{x}) - F_X(\mathbf{x}-1) > \frac{1}{2^{l(\mathbf{x})}}. \tag{4.12}$$

Combining (4.11) and (4.12), we have

$$\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1). \tag{4.13}$$

Therefore, the code $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is a unique representation of $\bar{T}_X(\mathbf{x})$.

To show that this code is uniquely decodable, we will show that the code is a prefix code; that is, no codeword is a prefix of another codeword. Because a prefix code is always uniquely decodable, by showing that an arithmetic code is a prefix code, we automatically show that it is uniquely decodable. Given a number a in the interval $[0, 1)$ with an n-bit binary representation $[b_1 b_2 \ldots b_n]$, for any other number b to have a binary representation with $[b_1 b_2 \ldots b_n]$ as the prefix, b has to lie in the interval $[a, a + \frac{1}{2^n})$. (See Problem 1.)

If $\mathbf{x}$ and $\mathbf{y}$ are two distinct sequences, we know that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ and $\lfloor \bar{T}_X(\mathbf{y}) \rfloor_{l(\mathbf{y})}$ lie in two *disjoint* intervals, $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ and $[F_X(\mathbf{y}-1), F_X(\mathbf{y}))$. Therefore, if we can show that for any sequence $\mathbf{x}$, the interval $[\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}, \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} + \frac{1}{2^{l(\mathbf{x})}})$ lies entirely within the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$, this will mean that the code for one sequence cannot be the prefix for the code for another sequence.

We have already shown that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1)$. Therefore, all we need to do is show that

$$F_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > \frac{1}{2^{l(\mathbf{x})}}.$$

This is true because

$$\begin{aligned} F_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} &> F_X(\mathbf{x}) - \bar{T}_X(\mathbf{x}) \\ &= \frac{P(\mathbf{x})}{2} \\ &> \frac{1}{2^{l(\mathbf{x})}}. \end{aligned}$$

This code is prefix free, and by taking the binary representation of $\bar{T}_X(\mathbf{x})$ and truncating it to $l(x) = \lceil \log \frac{1}{P(x)} \rceil + 1$ bits, we obtain a uniquely decodable code.

Although the code is uniquely decodable, how efficient is it? We have shown that the number of bits $l(\mathbf{x})$ required to represent $F_X(\mathbf{x})$ with enough accuracy such that the code for different values of $\mathbf{x}$ are distinct is

$$l(\mathbf{x}) = \left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1.$$

Remember that $l(\mathbf{x})$ is the number of bits required to encode the *entire* sequence $\mathbf{x}$. So, the average length of an arithmetic code for a sequence of length m is given by

$$\begin{aligned} l_{A^m} &= \sum P(\mathbf{x}) l(\mathbf{x}) && (4.14) \\ &= \sum P(\mathbf{x}) \left[\left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1 \right] && (4.15) \\ &< \sum P(\mathbf{x}) \left[\log \frac{1}{P(\mathbf{x})} + 1 + 1 \right] && (4.16) \\ &= -\sum P(\mathbf{x}) \log P(\mathbf{x}) + 2 \sum P(\mathbf{x}) && (4.17) \\ &= H(X^m) + 2. && (4.18) \end{aligned}$$

Given that the average length is always greater than the entropy, the bounds on $l_{A^{(m)}}$ are

$$H(X^{(m)}) \leq l_{A^{(m)}} < H(X^{(m)}) + 2.$$

The length per symbol, l_A, or rate of the arithmetic code is $\frac{l_{A^{(m)}}}{m}$. Therefore, the bounds on l_A are

$$\frac{H(X^{(m)})}{m} \leq l_A < \frac{H(X^{(m)})}{m} + \frac{2}{m}. \quad (4.19)$$

We have shown in Chapter 3 that for *iid* sources

$$H(X^{(m)}) = mH(X). \quad (4.20)$$

Therefore,

$$H(X) \leq l_A < H(X) + \frac{2}{m}. \tag{4.21}$$

By increasing the length of the sequence, we can guarantee a rate as close to the entropy as we desire.

4.4.2 Algorithm Implementation

In Section 4.3.1 we developed a recursive algorithm for the boundaries of the interval containing the tag for the sequence being encoded as

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1) \tag{4.22}$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n) \tag{4.23}$$

where x_n is the value of the random variable corresponding to the nth observed symbol, $l^{(n)}$ is the lower limit of the tag interval at the nth iteration, and $u^{(n)}$ is the upper limit of the tag interval at the nth iteration.

Before we can implement this algorithm, there is one major problem we have to resolve. Recall that the rationale for using numbers in the interval [0, 1) as a tag was that there are an infinite number of numbers in this interval. However, in practice the number of numbers that can be uniquely represented on a machine is limited by the maximum number of digits (or bits) we can use for representing the number. Consider the values of $l^{(n)}$ and $u^{(n)}$ in Example 4.3.5. As n gets larger, these values come closer and closer together. This means that in order to represent all the subintervals uniquely we need increasing precision as the length of the sequence increases. In a system with finite precision, the two values are bound to converge, and we will lose all information about the sequence from the point at which the two values converged. To avoid this situation, we need to rescale the interval. However, we have to do it in a way that will preserve the information that is being transmitted. We would also like to perform the encoding *incrementally*—that is, to transmit portions of the code as the sequence is being observed, rather than wait until the entire sequence has been observed before transmitting the first bit. The algorithm we describe in this section takes care of the problems of synchronized rescaling and incremental encoding.

As the interval becomes narrower, we have three possibilities:

1. The interval is entirely confined to the lower half of the unit interval [0, 0.5).
2. The interval is entirely confined to the upper half of the unit interval [0.5, 1.0).
3. The interval straddles the midpoint of the unit interval.

We will look at the third case a little later in this section. First, let us examine the first two cases. Once the interval is confined to either the upper or lower half of the unit interval, it is forever confined to that half of the unit interval. The most significant bit of the binary representation of all numbers in the interval [0, 0.5) is 0, and the most significant bit of the binary representation of all numbers in the interval [0.5, 1] is 1. Therefore, once the interval gets restricted to either the upper or lower half of the unit interval, the most significant bit of

the tag is fully determined. Therefore, without waiting to see what the rest of the sequence looks like, we can indicate to the decoder whether the tag is confined to the upper or lower half of the unit interval by sending a 1 for the upper half and a 0 for the lower half. The bit that we send is also the first bit of the tag.

Once the encoder and decoder know which half contains the tag, we can ignore the half of the unit interval not containing the tag and concentrate on the half containing the tag. As our arithmetic is of finite precision, we can do this best by mapping the half interval containing the tag to the full $[0, 1)$ interval. The mappings required are

$$E_1 : [0, 0.5) \rightarrow [0, 1); \qquad E_1(x) = 2x \tag{4.24}$$

$$E_2 : [0.5, 1) \rightarrow [0, 1); \qquad E_2(x) = 2(x - 0.5). \tag{4.25}$$

As soon as we perform either of these mappings, we lose all information about the most significant bit. However, this should not matter because we have already sent that bit to the decoder. We can now continue with this process, generating another bit of the tag every time the tag interval is restricted to either half of the unit interval. This process of generating the bits of the tag without waiting to see the entire sequence is called incremental encoding.

Example 4.4.2: Tag generation with scaling

Let's revisit Example 4.3.5. Recall that we wish to encode the sequence **1 3 2 1**. The probability model for the source is $P(a_1) = 0.8$, $P(a_2) = 0.02$, $P(a_3) = 0.18$. Initializing $u^{(0)}$ to 1, and $l^{(0)}$ to 0, the first element of the sequence, **1**, results in the following update:

$$\begin{aligned} l^{(1)} &= 0 + (1 - 0)0 = 0 \\ u^{(1)} &= 0 + (1 - 0)(0.8) = 0.8. \end{aligned}$$

The interval $[0, 0.8)$ is not confined to either the upper or lower half of the unit interval, so we proceed.

The second element of the sequence is **3**. This results in the update

$$\begin{aligned} l^{(2)} &= 0 + (0.8 - 0)F_X(2) = 0.8 \times 0.82 = 0.656 \\ u^{(2)} &= 0 + (0.8 - 0)F_X(3) = 0.8 \times 1.0 = 0.8. \end{aligned}$$

The interval $[0.656, 0.8)$ is contained entirely in the upper half of the unit interval, so we send the binary code 1 and rescale:

$$\begin{aligned} l^{(2)} &= 2 \times (0.656 - 0.5) = 0.312 \\ u^{(2)} &= 2 \times (0.8 - 0.5) = 0.6. \end{aligned}$$

The third element, **2**, results in the following update equations:

$$\begin{aligned} l^{(3)} &= 0.312 + (0.6 - 0.312)F_X(1) = 0.312 + 0.288 \times 0.8 = 0.5424 \\ u^{(3)} &= 0.312 + (0.8 - 0.312)F_X(2) = 0.312 + 0.288 \times 0.82 = 0.54816. \end{aligned}$$

The interval for the tag is [0.5424, 0.54816), which is contained entirely in the upper half of the unit interval. We transmit a 1 and go through another rescaling:

$$l^{(3)} = 2 \times (0.5424 - 0.5) = 0.0848$$
$$u^{(3)} = 2 \times (0.54816 - 0.5) = 0.09632.$$

This interval is contained entirely in the lower half of the unit interval, so we send a 0 and use the E_1 mapping to rescale:

$$l^{(3)} = 2 \times (0.0848) = 0.1696$$
$$u^{(3)} = 2 \times (0.09632) = 0.19264.$$

The interval is still contained entirely in the lower half of the unit interval, so we send another 0 and go through another rescaling:

$$l^{(3)} = 2 \times (0.1696) = 0.3392$$
$$u^{(3)} = 2 \times (0.19264) = 0.38528.$$

Because the interval containing the tag remains in the lower half of the unit interval, we send another 0 and rescale one more time:

$$l^{(3)} = 2 \times 0.3392 = 0.6784$$
$$u^{(3)} = 2 \times 0.38528 = 0.77056.$$

Now the interval containing the tag is contained entirely in the upper half of the unit interval. Therefore, we transmit a 1 and rescale using the E_2 mapping:

$$l^{(3)} = 2 \times (0.6784 - 0.5) = 0.3568$$
$$u^{(3)} = 2 \times (0.77056 - 0.5) = 0.54112.$$

At each stage we are transmitting the most significant bit that is the same in both the upper and lower limit of the tag interval. If the most significant bits in the upper and lower limit are the same, then the value of this bit will be identical to the most significant bit of the tag. Therefore, by sending the most significant bits of the upper and lower endpoint of the tag whenever they are identical, we are actually sending the binary representation of the tag. The rescaling operations can be viewed as left shifts, which make the second most significant bit the most significant bit.

Continuing with the last element, the upper and lower limits of the interval containing the tag are

$$l^{(4)} = 0.3568 + (0.54112 - 0.3568)F_X(0) = 0.3568 + 0.18422 \times 0.0 = 0.3568$$
$$u^{(4)} = 0.3568 + (0.54112 - 0.3568)F_X(1) = 0.3568 + 0.18422 \times 0.8 = 0.504256.$$

At this point, if we wished to stop encoding, all we need to do is inform the receiver of the final status of the tag value. We can do so by sending the binary representation of any value in the final tag interval. Generally, this value is taken to be $l^{(n)}$. In this particular example, it is convenient to use the value of 0.5. The binary representation of 0.5 is .10 . . . . Thus, we would transmit a 1 followed by as many 0s as required by the word length of the implementation being used. ♦

Notice that the tag interval size at this stage is approximately 64 times the size it was when we were using the unmodified algorithm. Therefore, this technique solves the finite precision problem. As we shall soon see, the bits that we have been sending with each mapping constitute the tag itself, which satisfies our desire for incremental encoding. The binary sequence generated during the encoding process in the previous example is 1100011. We could simply treat this as the binary expansion of the tag. A binary number .1100011 corresponds to the decimal number 0.7734375. Looking back to Example 4.3.5, notice that this number lies within the final tag interval. Therefore, we could use this to decode the sequence.

However, we would like to do incremental decoding as well as incremental encoding. This raises three questions:

1. How do we start decoding?
2. How do we continue decoding?
3. How do we stop decoding?

The second question is the easiest to answer. Once we have started decoding, all we have to do is mimic the encoder algorithm. That is, once we have started decoding, we know how to continue decoding. To begin the decoding process, we need to have enough information to decode the first symbol unambiguously. In order to guarantee unambiguous decoding, the number of bits received should point to an interval smaller than the smallest tag interval. Based on the smallest tag interval, we can determine how many bits we need before we start the decoding procedure. We will demonstrate this procedure in Example 4.4.4. First let's look at other aspects of decoding using the message from Example 4.4.2.

Example 4.4.3:

We will use a word length of 6 for this example. Note that because we are dealing with real numbers this word length may not be sufficient for a different sequence. As in the encoder, we start with initializing $u^{(0)}$ to 1 and $l^{(0)}$ to 0. The sequence of received bits is 110001100 . . . 0. The first 6 bits correspond to a tag value of 0.765625, which means that the first element of the sequence is **1**, resulting in the following update:

$$l^{(1)} = 0 + (1-0)0 = 0$$
$$u^{(1)} = 0 + (1-0)(0.8) = 0.8.$$

The interval [0, 0.8) is not confined to either the upper or lower half of the unit interval, so we proceed. The tag 0.765625 lies in the top 18% of the interval [0, 0.8); therefore, the second element of the sequence is **3**. Updating the tag interval we get

$$l^{(2)} = 0 + (0.8 - 0)F_X(2) = 0.8 \times 0.82 = 0.656$$
$$u^{(2)} = 0 + (0.8 - 0)F_X(3) = 0.8 \times 1.0 = 0.8.$$

The interval [0.656, 0.8) is contained entirely in the upper half of the unit interval. At the encoder, we sent the bit 1 and rescaled. At the decoder, we will shift 1 out of the receive buffer and move the next bit in to make up the 6 bits in the tag. We will also update the tag interval, resulting in

$$l^{(2)} = 2 \times (0.656 - 0.5) = 0.312$$
$$u^{(2)} = 2 \times (0.8 - 0.5) = 0.6$$

while shifting a bit to give us a tag of 0.546875. When we compare this value with the tag interval, we can see that this value lies in the 80–82% range of the tag interval, so we decode the next element of the sequence as **2**. We can then update the equations for the tag interval as

$$l^{(3)} = 0.312 + (0.6 - 0.312)F_X(1) = 0.312 + 0.288 \times 0.8 = 0.5424$$
$$u^{(3)} = 0.312 + (0.8 - 0.312)F_X(2) = 0.312 + 0.288 \times 0.82 = 0.54816.$$

As the tag interval is now contained entirely in the upper half of the unit interval, we rescale using E_2 to obtain

$$l^{(3)} = 2 \times (0.5424 - 0.5) = 0.0848$$
$$u^{(3)} = 2 \times (0.54816 - 0.5) = 0.09632.$$

We also shift out a bit from the tag and shift in the next bit. The tag is now 000110. The interval is contained entirely in the lower half of the unit interval. Therefore, we apply E_1 and shift another bit. The lower and upper limits of the tag interval become

$$l^{(3)} = 2 \times (0.0848) = 0.1696$$
$$u^{(3)} = 2 \times (0.09632) = 0.19264$$

and the tag becomes 001100. The interval is still contained entirely in the lower half of the unit interval, so we shift out another 0 to get a tag of 011000 and go through another rescaling:

$$l^{(3)} = 2 \times (0.1696) = 0.3392$$
$$u^{(3)} = 2 \times (0.19264) = 0.38528.$$

Because the interval containing the tag remains in the lower half of the unit interval, we shift out another 0 from the tag to get 110000 and rescale one more time:

$$l^{(3)} = 2 \times 0.3392 = 0.6784$$
$$u^{(3)} = 2 \times 0.38528 = 0.77056.$$

Now the interval containing the tag is contained entirely in the upper half of the unit interval. Therefore, we shift out a 1 from the tag and rescale using the E_2 mapping:

$$l^{(3)} = 2 \times (0.6784 - 0.5) = 0.3568$$
$$u^{(3)} = 2 \times (0.77056 - 0.5) = 0.54112.$$

Now we compare the tag value to the the tag interval to decode our final element. The tag is 100000, which corresponds to 0.5. This value lies in the first 80% of the interval, so we decode this element as **1**. ♦

If the tag interval is entirely contained in the upper or lower half of the unit interval, the scaling procedure described will prevent the interval from continually shrinking. Now we consider the case where the diminishing tag interval straddles the midpoint of the unit interval. As our trigger for rescaling, we check to see if the tag interval is contained in the interval $[0.25, 0.75)$. This will happen when $l^{(n)}$ is greater than 0.25 and $u^{(n)}$ is less than 0.75. When this happens, we double the tag interval using the following mapping:

$$E_3 : [0.25, 0.75) \rightarrow [0, 1); \qquad E_3(x) = 2(x - 0.25). \tag{4.26}$$

We have used a 1 to transmit information about an E_2 mapping, and a 0 to transmit information about an E_1 mapping. How do we transfer information about an E_3 mapping to the decoder? We use a somewhat different strategy in this case. At the time of the E_3 mapping, we do not send any information to the decoder; instead, we simply record the fact that we have used the E_3 mapping at the encoder. Suppose that after this, the tag interval gets confined to the upper half of the unit interval. At this point we would use an E_2 mapping and send a 1 to the receiver. Note that the tag interval at this stage is at least twice what it would have been if we had not used the E_3 mapping. Furthermore, the upper limit of the tag interval would have been less than 0.75. Therefore, if the E_3 mapping had not taken place right before the E_2 mapping, the tag interval would have been contained entirely in the lower half of the unit interval. At this point we would have used an E_1 mapping and transmitted a 0 to the receiver. In fact, the effect of the earlier E_3 mapping can be mimicked at the decoder by following the E_2 mapping with an E_1 mapping. At the encoder, right after we send a 1 to announce the E_2 mapping, we send a 0 to help the decoder track the changes in the tag interval at the decoder. If the first rescaling after the E_3 mapping happens to be an E_1 mapping, we do exactly the opposite. That is, we follow the 0 announcing an E_1 mapping with a 1 to mimic the effect of the E_3 mapping at the encoder.

What happens if we have to go through a series of E_3 mappings at the encoder? We simply keep track of the number of E_3 mappings and then send that many bits of the opposite variety after the first E_1 or E_2 mapping. If we went through three E_3 mappings at the encoder,

followed by an E_2 mapping, we would transmit a 1 followed by three 0s. On the other hand, if we went through an E_1 mapping after the E_3 mappings, we would transmit a 0 followed by three 1s. Since the decoder mimics the encoder, the E_3 mappings are also applied at the decoder when the tag interval is contained in the interval [0.25, 0.75).

4.4.3 Integer Implementation

We have described a floating-point implementation of arithmetic coding. Let us now repeat the procedure using integer arithmetic and generate the binary code in the process.

Encoder Implementation

The first thing we have to do is decide on the word length to be used. Given a word length of m, we map the important values in the [0, 1) interval to the range of 2^m binary words. The point 0 gets mapped to

$$\overbrace{00\ldots0}^{m \text{ times}},$$

1 gets mapped to

$$\overbrace{11\ldots1}^{m \text{ times}}.$$

The value of 0.5 gets mapped to

$$\overbrace{100\ldots0}^{m-1 \text{ times}}.$$

The update equations remain almost the same as Equations (4.9) and (4.10). As we are going to do integer arithmetic, we need to replace $F_X(x)$ in these equations.

Define n_j as the number of times the symbol j occurs in a sequence of length *Total Count*. Then $F_X(k)$ can be estimated by

$$F_X(k) = \frac{\sum_{i=1}^{k} n_i}{Total\ Count}. \tag{4.27}$$

If we now define

$$Cum\_Count(k) = \sum_{i=1}^{k} n_i$$

we can write Equations (4.9) and (4.10) as

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n - 1)}{Total\ Count} \right\rfloor \tag{4.28}$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n)}{Total\ Count} \right\rfloor - 1 \tag{4.29}$$

where x_n is the nth symbol to be encoded, $\lfloor x \rfloor$ is the largest integer less than or equal to x, and where the addition and subtraction of one is to handle the effects of the integer arithmetic.

Because of the way we mapped the endpoints and the halfway points of the unit interval, when both $l^{(n)}$ and $u^{(n)}$ are in either the upper half or lower half of the interval, the leading bit of $u^{(n)}$ and $l^{(n)}$ will be the same. If the leading or most significant bit (MSB) is 1, then the tag interval is contained entirely in the upper half of the [00 . . . 0, 11 . . . 1] interval. If the MSB is 0, then the tag interval is contained entirely in the lower half. Applying the E_1 and E_2 mappings is a simple matter. All we do is shift out the MSB and then shift in a 1 into the integer code for $u^{(n)}$ and a 0 into the code for $l^{(n)}$. For example, suppose m was 6, $u^{(n)}$ was 54, and $l^{(n)}$ was 33. The binary representations of $u^{(n)}$ and $l^{(n)}$ are 110110 and 100001, respectively. Notice that the MSB for both endpoints is 1. Following the procedure above, we would shift out (and transmit or store) the 1, and shift in 1 for $u^{(n)}$ and 0 for $l^{(n)}$, obtaining the new value for $u^{(n)}$ as 101101, or 45, and a new value for $l^{(n)}$ as 000010, or 2. This is equivalent to performing the E_2 mapping. We can see how the E_1 mapping would also be performed using the same operation.

To see if the E_3 mapping needs to be performed, we monitor the second most significant bit of $u^{(n)}$ and $l^{(n)}$. When the second most significant bit of $u^{(n)}$ is 0 and the second most significant bit of $l^{(n)}$ is 1, this means that the tag interval lies in the middle half of the [00 . . . 0, 11 . . . 1] interval. To implement the E_3 mapping, we complement the second most significant bit in $u^{(n)}$ and $l^{(n)}$, and shift left, shifting in a 1 in $u^{(n)}$ and a 0 in $l^{(n)}$. We also keep track of the number of E_3 mappings in Scale3.

We can summarize the encoding algorithm using the following pseudocode:

Initialize l and u.
Get symbol.

$$l \longleftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x-1)}{TotalCount} \right\rfloor$$

$$u \longleftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x)}{TotalCount} \right\rfloor - 1$$

while(MSB of u and l are both equal to b or E_3 condition holds)
if(MSB of u and l are both equal to b)

```
{
send b
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
while(Scale3 > 0)
    {
    send complement of b
    decrement Scale3
    }
}
```

if(E_3 condition holds)

{
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
complement (new) MSB of l and u
increment Scale3
}

To see how all this functions together, let's look at an example.

Example 4.4.4:

We will encode the sequence **1 3 2 1** with parameters shown in Table 4.5. First we need to select the word length m. Note that *Cum_Count*(1) and *Cum_Count*(2) differ by only 1. Recall that the values of *Cum_Count* will get translated to the endpoints of the subintervals. We want to make sure that the value we select for the word length will allow enough range for it to be possible to represent the smallest difference between the endpoints of intervals. We always rescale whenever the interval gets small. In order to make sure that the endpoints of the intervals always remain distinct, we need to make sure that all values in the range from 0 to *Total_Count*, which is the same as *Cum_Count*(3), are uniquely represented in the smallest range an interval under consideration can be without triggering a rescaling. The interval is smallest without triggering a rescaling when $l^{(n)}$ is just below the midpoint of the interval and $u^{(n)}$ is at three-quarters of the interval, or when $u^{(n)}$ is right at the midpoint of the interval and $l^{(n)}$ is just below a quarter of the interval. That is, the smallest the interval $[l^{(n)}, u^{(n)}]$ can be is one-quarter of the total available range of 2^m values. Thus, m should be large enough to accommodate uniquely the set of values between 0 and *Total_Count*.

TABLE 4.5 Values of some of the parameters for arithmetic coding example.

Count(1) = 40	*Cum_Count*(0) = 0	Scale3 = 0
Count(2) = 1	*Cum_Count*(1) = 40	
Count(3) = 9	*Cum_Count*(2) = 41	
Total_Count = 50	*Cum_Count*(3) = 50	

For this example, this means that the total interval range has to be greater than 200. A value of $m = 8$ satisfies this requirement.

With this value of m we have

$$l^{(0)} = 0 = (00000000)_2 \tag{4.30}$$

$$u^{(0)} = 255 = (11111111)_2 \tag{4.31}$$

where $(\cdots)_2$ is the binary representation of a number.

The first element of the sequence to be encoded is **1**. Using Equations (4.28) and (4.29),

$$l^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{4.32}$$

$$u^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 203 = (11001011)_2. \tag{4.33}$$

The next element of the sequence is **3**.

$$l^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(2)}{50} \right\rfloor = 167 = (10100111)_2 \tag{4.34}$$

$$u^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(3)}{50} \right\rfloor - 1 = 203 = (11001011)_2 \tag{4.35}$$

The MSBs of $l^{(2)}$ and $u^{(2)}$ are both 1. Therefore, we shift this value out and send it to the decoder. All other bits are shifted left by 1 bit, giving

$$l^{(2)} = (01001110)_2 = 78 \tag{4.36}$$

$$u^{(2)} = (10010111)_2 = 151. \tag{4.37}$$

Notice that while the MSBs of the limits are different, the second MSB of the upper limit is 0, while the second MSB of the lower limit is 1. This is the condition for the E_3 mapping. We complement the second MSB of both limits and shift 1 bit to the left, shifting in a 0 as the LSB of $l^{(2)}$ and a 1 as the LSB of $u^{(2)}$. This gives us

$$l^{(2)} = (00011100)_2 = 28 \tag{4.38}$$

$$u^{(2)} = (10101111)_2 = 175. \tag{4.39}$$

We also increment Scale3 to a value of 1.

The next element in the sequence is **2**. Updating the limits, we have

$$l^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(1)}{50} \right\rfloor = 146 = (10010010)_2 \tag{4.40}$$

$$u^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(2)}{50} \right\rfloor - 1 = 148 = (10010100)_2. \tag{4.41}$$

The two MSBs are identical, so we shift out a 1 and shift left by 1 bit:

$$l^{(3)} = (00100100)_2 = 36 \tag{4.42}$$

$$u^{(3)} = (00101001)_2 = 41. \tag{4.43}$$

As Scale3 is 1, we transmit a 0 and decrement Scale3 to 0. The MSBs of the upper and lower limits are both 0, so we shift out and transmit 0:

$$l^{(3)} = (01001000)_2 = 72 \tag{4.44}$$

$$u^{(3)} = (01010011)_2 = 83. \tag{4.45}$$

Both MSBs are again 0, so we shift out and transmit 0:

$$l^{(3)} = (10010000)_2 = 144 \tag{4.46}$$

$$u^{(3)} = (10100111)_2 = 167. \tag{4.47}$$

Now both MSBs are 1, so we shift out and transmit a 1. The limits become

$$l^{(3)} = (00100000)_2 = 32 \tag{4.48}$$

$$u^{(3)} = (01001111)_2 = 79. \tag{4.49}$$

Once again the MSBs are the same. This time we shift out and transmit a 0.

$$l^{(3)} = (01000000)_2 = 64 \tag{4.50}$$

$$u^{(3)} = (10011111)_2 = 159. \tag{4.51}$$

Now the MSBs are different. However, the second MSB for the lower limit is 1 while the second MSB for the upper limit is 0. This is the condition for the E_3 mapping. Applying the E_3 mapping by complementing the second MSB and shifting 1 bit to the left, we get

$$l^{(3)} = (00000000)_2 = 0 \tag{4.52}$$

$$u^{(3)} = (10111111)_2 = 191. \tag{4.53}$$

We also increment Scale3 to 1.

The next element in the sequence to be encoded is **1**. Therefore,

$$l^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{4.54}$$

$$u^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 152 = (10011000)_2. \tag{4.55}$$

The encoding continues in this fashion. To this point we have generated the binary sequence 1100010. If we wished to terminate the encoding at this point, we have to send the current status of the tag. This can be done by sending the value of the lower limit $l^{(4)}$. As $l^{(4)}$ is 0, we will end up sending eight 0s. However, Scale3 at this point is 1. Therefore, after we send the first 0 from the value of $l^{(4)}$, we need to send a 1 before sending the remaining seven 0s. The final transmitted sequence is 1100010010000000. ♦

Decoder Implementation

Once we have the encoder implementation, the decoder implementation is easy to describe. As mentioned earlier, once we have started decoding all we have to do is mimic the encoder algorithm. Let us first describe the decoder algorithm using pseudocode and then study its implementation using Example 4.4.5.

Decoder Algorithm

Initialize l and u.
Read the first m bits of the received bitstream into tag t.
$k = 0$
while$\left(\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor \geq Cum_Count(k)\right)$
$k \longleftarrow k+1$
decode symbol x.
$l \longleftarrow l+\left\lfloor \frac{(u-l+1)\times Cum\_Count(x-1)}{Total\ Count} \right\rfloor$
$u \longleftarrow l+\left\lfloor \frac{(u-l+1)\times Cum\_Count(x)}{Total\ Count} \right\rfloor -1$
while(MSB of u and l are both equal to b or E_3 condition holds)
if(MSB of u and l are both equal to b)

{
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
shift t to the left by 1 bit and read next bit from received bitstream into LSB
}

if(E_3 condition holds)

{
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
shift t to the left by 1 bit and read next bit from received bitstream into LSB
complement (new) MSB of l, u, and t
}

Example 4.4.5:

After encoding the sequence in Example 4.4.4, we ended up with the following binary sequence: 1100010010000000. Treating this as the received sequence and using the parameters from Table 4.5, let us decode this sequence. Using the same word length, eight, we read in the first 8 bits of the received sequence to form the tag t:

$$t = (11000100)_2 = 196.$$

We initialize the lower and upper limits as

$$l = (00000000)_2 = 0$$
$$u = (11111111)_2 = 255.$$

To begin decoding, we compute

$$\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor = \left\lfloor \frac{197\times 50-1}{255-0+1} \right\rfloor = 38$$

and compare this value to

$$Cum\_Count = \begin{bmatrix} 0 \\ 40 \\ 41 \\ 50 \end{bmatrix}$$

Since

$$0 \le 38 < 40,$$

we decode the first symbol as **1**. Once we have decoded a symbol, we update the lower and upper limits:

$$l = 0+\left\lfloor \frac{256\times Cum\_Count[0]}{Total\ Count} \right\rfloor = 0+\left\lfloor 256\times \frac{0}{50} \right\rfloor = 0$$

$$u = 0+\left\lfloor \frac{256\times Cum\_Count[1]}{Total\ Count} \right\rfloor - 1 = 0+\left\lfloor 256\times \frac{40}{50} \right\rfloor - 1 = 203$$

or

$$l = (00000000)_2$$

$$u = (11001011)_2.$$

The MSB of the limits are different and the E_3 condition does not hold. Therefore, we continue decoding without modifying the tag value. To obtain the next symbol, we compare

$$\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor$$

which is 48, against the *Cum_Count* array:

$$Cum\_Count[2] \le 48 < Cum\_Count[3].$$

Therefore, we decode **3** and update the limits:

$$l = 0+\left\lfloor \frac{204\times Cum\_Count[2]}{Total\ Count} \right\rfloor = 0+\left\lfloor 204\times \frac{41}{50} \right\rfloor = 167 = (1010011)_2$$

$$u = 0+\left\lfloor \frac{204\times Cum\_Count[3]}{Total\ Count} \right\rfloor - 1 = 0+\left\lfloor 204\times \frac{50}{50} \right\rfloor - 1 = 203 = (11001011)_2.$$

As the MSB of u and l are the same, we shift the MSB out and read in a 0 for the LSB of l and a 1 for the LSB of u. We mimic this action for the tag as well, shifting the MSB out and reading in the next bit from the received bitstream as the LSB:

$$l = (01001110)_2$$
$$u = (10010111)_2$$
$$t = (10001001)_2.$$

Examining l and u we can see we have an E_3 condition. Therefore, for l, u, and t, we shift the MSB out, complement the new MSB, and read in a 0 as the LSB of l, a 1 as the LSB of u, and the next bit in the received bitstream as the LSB of t. We now have

$$l = (00011100)_2 = 28$$
$$u = (10101111)_2 = 175$$
$$t = (10010010)_2 = 146.$$

To decode the next symbol, we compute

$$\left\lfloor \frac{(t-l+1) \times Total\ Count - 1}{u-l+1} \right\rfloor = 40.$$

Since $40 \leq 40 < 41$, we decode **2**.

Updating the limits using this decoded symbol, we get

$$l = 28 + \left\lfloor \frac{(175-28+1) \times 40}{50} \right\rfloor = 146 = (10010010)_2$$
$$u = 28 + \left\lfloor \frac{(175-28+1) \times 41}{50} \right\rfloor - 1 = 148 = (10010100)_2.$$

We can see that we have quite a few bits to shift out. However, notice that the lower limit l has the same value as the tag t. Furthermore, the remaining received sequence consists entirely of 0s. Therefore, we will be performing identical operations on numbers that are the same, resulting in identical numbers. This will result in the final decoded symbol being **1**. We knew this was the final symbol to be decoded because only four symbols had been encoded. In practice this information has to be conveyed to the decoder. ♦

4.5 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

$$l = 2 \times 0.5 + 3 \times 0.25 + 4 \times 0.125 + 4 \times 0.125 \tag{4.56}$$

$$= 2.75 \text{ bits/symbol.} \tag{4.57}$$

Recall from Section 2.4 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. (Note that we are only doing this to demonstrate a point. In practice, we would not code sequences this short using an arithmetic code.)

Example 4.5.1:

If we encode two symbols at a time, the resulting code is shown in Table 4.6.

TABLE 4.6 Arithmetic code for two-symbol sequences.

Message	$P(x)$	$\tilde{T}_X(x)$	$\tilde{T}_X(x)$ in Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	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

The average length per message is 4.5 bits. Therefore, using two symbols at a time we get a rate of 2.25 bits/symbol (certainly better than 2.75 bits/symbol, but still not as good as the best rate of 1.75 bits/symbol). However, we see that as we increase the number of symbols per message, our results get better and better. ♦

How many samples do we have to group together to make the arithmetic coding scheme perform better than the Huffman coding scheme? We can get some idea by looking at the bounds on the coding rate.

Recall that the bounds on the average length l_A of the arithmetic code are

$$H(X) \leq l_A \leq H(X) + \frac{2}{m}.$$

It does not take many symbols in a sequence before the coding rate for the arithmetic code becomes quite close to the entropy. However, recall that for Huffman codes, if we block m symbols together, the coding rate is

$$H(X) \leq l_H \leq H(X) + \frac{1}{m}.$$

The advantage seems to lie with the Huffman code, although the advantage decreases with increasing m. However, remember that to generate a codeword for a sequence of length m, using the Huffman procedure requires building the entire code for all possible sequences of length m. If the original alphabet size was k, then the size of the codebook would be k^m. Taking relatively reasonable values of $k = 16$ and $m = 20$ gives a codebook size of 16^{20}! This is obviously not a viable option. For the arithmetic coding procedure, we do not need to build the entire codebook. Instead, we simply obtain the code for the tag corresponding to a given sequence. Therefore, it is entirely feasible to code sequences of length 20 or much more. In practice, we can make m large for the arithmetic coder and not for the Huffman coder. This means that for most sources we can get rates closer to the entropy using arithmetic coding than by using Huffman coding. The exceptions are sources whose probabilities are powers of two. In these cases, the single-letter Huffman code achieves the entropy, and we cannot do any better with arithmetic coding, no matter how long a sequence we pick.

The amount of gain also depends on the source. Recall that for Huffman codes we are guaranteed to obtain rates within $0.086 + p_{\max}$ of the entropy, where $p_{\max}$ is the probability of the most probable letter in the alphabet. If the alphabet size is relatively large and the probabilities are not too skewed, the maximum probability $p_{\max}$ is generally small. In these cases, the advantage of arithmetic coding over Huffman coding is small, and it might not be worth the extra complexity to use arithmetic coding rather than Huffman coding. However, there are many sources, such as facsimile, in which the alphabet size is small, and the probabilities are highly unbalanced. In these cases, the use of arithmetic coding is generally worth the added complexity.

Another major advantage of arithmetic coding is that it is easy to implement a system with multiple arithmetic codes. This may seem contradictory, as we have claimed that arithmetic coding is more complex than Huffman coding. However, it is the computational machinery that causes the increase in complexity. Once we have the computational machinery to implement one arithmetic code, all we need to implement more than a single arithmetic code is the availability of more probability tables. If the alphabet size of the source is small, as in the case of a binary source, there is very little added complexity indeed. In fact, as we shall see in the next section, it is possible to develop multiplication-free arithmetic coders that are quite simple to implement (nonbinary multiplication-free arithmetic coders are described in [44]).

Finally, it is much easier to adapt arithmetic codes to changing input statistics. All we need to do is estimate the probabilities of the input alphabet. This can be done by keeping a count of the letters as they are coded. There is no need to preserve a tree, as with adaptive Huffman codes. Furthermore, there is no need to generate a code a priori, as in the case of

Huffman coding. This property allows us to separate the modeling and coding procedures in a manner that is not very feasible with Huffman coding. This separation permits greater flexibility in the design of compression systems, which can be used to great advantage.

4.6 Adaptive Arithmetic Coding

We have seen how to construct arithmetic coders when the distribution of the source, in the form of cumulative counts, is available. In many applications such counts are not available a priori. It is a relatively simple task to modify the algorithms discussed so that the coder learns the distribution as the coding progresses. A straightforward implementation is to start out with a count of 1 for each letter in the alphabet. We need a count of at least 1 for each symbol, because if we do not we will have no way of encoding the symbol when it is first encountered. This assumes that we know nothing about the distribution of the source. If we do know something about the distribution of the source, we can let the initial counts reflect our knowledge.

After coding is initiated, the count for each letter encountered is incremented *after* that letter has been encoded. The cumulative count table is updated accordingly. It is very important that the updating take place after the encoding; otherwise the decoder will not be using the same cumulative count table as the encoder to perform the decoding. At the decoder, the count and cumulative count tables are updated after each letter is decoded.

In the case of the static arithmetic code, we picked the size of the word based on Total Count, the total number of symbols to be encoded. In the adaptive case, we may not know ahead of time what the total number of symbols is going to be. In this case we have to pick the word length independent of the total count. However, given a word length m we know that we can only accomodate a total count of 2^{m-2} or less. Therefore, during the encoding and decoding processes when the total count approaches 2^{m-2}, we have to go through a rescaling, or renormalization, operation. A simple rescaling operation is to divide all counts by 2 and rounding up the result so that no count gets rescaled to zero. This periodic rescaling can have an added benefit in that the count table better reflects the local statisites of the source.

4.7 Applications

Arithmetic coding is used in a variety of lossless and lossy compression applications. It is a part of many international standards. In the area of multimedia there are a few principal organizations that develop standards. The International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) are industry groups that work on multimedia standards, while the International Telecommunications Union (ITU), which is part of the United Nations, works on multimedia standards on behalf of the member states of the United Nations. Quite often these institutions work together to create international standards. In later chapters we will be looking at a number of these standards, and we will see how arithmetic coding is used in image compression, audio compression, and video compression standards.

For now let us look at the lossless compression example from the previous chapter.

TABLE 4.7 Compression using adaptive arithmetic coding of pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	6.52	53,431	1.23	1.16
Sensin	7.12	58,306	1.12	1.27
Earth	4.67	38,248	1.71	1.67
Omaha	6.84	56,061	1.17	1.14

TABLE 4.8 Compression using adaptive arithmetic coding of pixel differences.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	3.89	31,847	2.06	2.08
Sensin	4.56	37,387	1.75	1.73
Earth	3.92	32,137	2.04	2.04
Omaha	6.27	51,393	1.28	1.26

In Tables 4.7 and 4.8, we show the results of using adaptive arithmetic coding to encode the same test images that were previously encoded using Huffman coding. We have included the compression ratios obtained using Huffman code from the previous chapter for comparison. Comparing these values to those obtained in the previous chapter, we can see very little change. The reason is that beacuse the alphabet size for the images is quite large, the value of p_{max} is quite small, and in the Huffman coder performs very close to the entropy.

As we mentioned before, a major advantage of arithmetic coding over Huffman coding is the ability to separate the modeling and coding aspects of the compression approach. In terms of image coding, this allows us to use a number of different models that take advantage of local properties. For example, we could use different decorrelation strategies in regions of the image that are quasi-constant and will, therefore, have differences that are small, and in regions where there is a lot of activity, causing the presence of larger difference values.

4.8 Summary

In this chapter we introduced the basic ideas behind arithmetic coding. We have shown that the arithmetic code is a uniquely decodable code that provides a rate close to the entropy for long stationary sequences. This ability to encode sequences directly instead of as a concatenation of the codes for the elements of the sequence makes this approach more efficient than Huffman coding for alphabets with highly skewed probabilities. We have looked in some detail at the implementation of the arithmetic coding approach.

The arithmetic coding results in this chapter were obtained by using the program provided by Witten, Neal, and Cleary [45]. This code can be used (with some modifications) for exploring different aspects of arithmetic coding (see problems).

Further Reading

1. The book *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1], contains a very readable section on arithmetic coding, complete with pseudocode and C code.
2. A thorough treatment of various aspects of arithmetic coding can be found in the excellent chapter *Arithmetic Coding*, by Amir Said [46] in the *Lossless Compression Handbook*.
3. There is an excellent tutorial article by G.G. Langdon, Jr. [47] in the March 1984 issue of the *IBM Journal of Research and Development*.
4. The separate model and code paradigm is explored in a precise manner in the context of arithmetic coding in a paper by J.J. Rissanen and G.G. Langdon [48].
5. The separation of modeling and coding is exploited in a very nice manner in an early paper by G.G. Langdon and J.J. Rissanen [49].
6. Various models for text compression that can be used effectively with arithmetic coding are described by T.G. Bell, I.H. Witten, and J.G. Cleary [50] in an article in the *ACM Computing Surveys*.
7. The coder used in the JBIG algorithm is a descendant of the Q coder, described in some detail in several papers [51, 52, 53] in the November 1988 issue of the *IBM Journal of Research and Development*.

4.9 Projects and Problems

1. Given a number a in the interval $[0, 1)$ with an n-bit binary representation $[b_1 b_2 \ldots b_n]$, show that for any other number b to have a binary representation with $[b_1 b_2 \ldots b_n]$ as the prefix, b has to lie in the interval $[a, a + \frac{1}{2^n})$.
2. The binary arithmetic coding approach specified in the JBIG standard can be used for coding gray-scale images via *bit plane encoding*. In bit plane encoding, we combine the most significant bits for each pixel into one bit plane, the next most significant bits into another bit plane, and so on. Use the function `extrctbp` to obtain eight bit planes for the `sena.img` and `omaha.img` test images, and encode them using arithmetic coding. Use the low-resolution contexts shown in Figure 7.11.
3. Bit plane encoding is more effective when the pixels are encoded using a *Gray code*. The Gray code assigns numerically adjacent values binary codes that differ by only 1 bit. To convert from the standard binary code $b_0 b_1 b_2 \ldots b_7$ to the Gray code $g_0 g_1 g_2 \ldots g_7$, we can use the equations

$$g_0 = b_0$$
$$g_k = b_k \oplus b_{k-1}.$$

Convert the test images `sena.img` and `omaha.img` to a Gray code representation, and bit plane encode. Compare with the results for the non-Gray-coded representation.

TABLE 4.9 Probability model for Problems 5 and 6.

Letter	Probability
a_1	.2
a_2	.3
a_3	.5

TABLE 4.10 Frequency counts for Problem 7.

Letter	Count
a	37
b	38
c	25

4. In Example 4.4.4, repeat the encoding using $m = 6$. Comment on your results.

5. Given the probability model in Table 4.9, find the real valued tag for the sequence $a_1\ a_1\ a_3\ a_2\ a_3\ a_1$.

6. For the probability model in Table 4.9, decode a sequence of length 10 with the tag 0.63215699.

7. Given the frequency counts shown in Table 4.10:

(a) What is the word length required for unambiguous encoding?

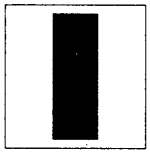
(b) Find the binary code for the sequence *abacabb*.

(c) Decode the code you obtained to verify that your encoding was correct.

8. Generate a binary sequence of length L with $P(0) = 0.8$, and use the arithmetic coding algorithm to encode it. Plot the difference of the rate in bits/symbol and the entropy as a function of L. Comment on the effect of L on the rate.

Dictionary Techniques

5.1 Overview



n the previous two chapters we looked at coding techniques that assume a source that generates a sequence of independent symbols. As most sources are correlated to start with, the coding step is generally preceded by a decorrelation step. In this chapter we will look at techniques that incorporate the structure in the data in order to increase the amount of compression. These techniques—both static and adaptive (or dynamic)—build a list of commonly occurring patterns and encode these patterns by transmitting their index in the list. They are most useful with sources that generate a relatively small number of patterns quite frequently, such as text sources and computer commands. We discuss applications to text compression, modem communications, and image compression.

5.2 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 with great rarity. For example, we can be reasonably sure that the word *Limpopo*[1] 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

[1] "How the Elephant Got Its Trunk" in *Just So Stories* by Rudyard Kipling.

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. Suppose 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

$$R = 9p + 21(1-p) = 21 - 12p. \tag{5.1}$$

For our scheme to be useful, R should have a value less than 20. This happens when $p \geq 0.084$. 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. We will look at both these approaches in this chapter.

5.3 Static Dictionary

Choosing a static dictionary technique is most appropriate when considerable prior knowledge 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 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 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*. We describe this in the next section.

5.3.1 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 5.3.1:

Suppose we have a source with a five-letter alphabet $\mathcal{A} = \{a, b, c, d, r\}$. Based on knowledge about the source, we build the dictionary shown in Table 5.1.

TABLE 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*

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. ♦

TABLE 5.2 Thirty most frequently occurring pairs of characters in a 41,364-character-long LaTeX document.

Pair	Count	Pair	Count
e␣	1128	*ar*	314
␣t	838	*at*	313
␣␣	823	*␣w*	309
th	817	*te*	296
he	712	*␣s*	295
in	512	*d␣*	272
s␣	494	*␣o*	266
er	433	*io*	257
␣a	425	*co*	256
t␣	401	*re*	247
en	392	*␣$*	246
on	385	*r␣*	239
n␣	353	*di*	230
ti	322	*ic*	229
␣i	317	*ct*	226

TABLE 5.3 Thirty most frequently occurring pairs of characters in a collection of C programs containing 64,983 characters.

Pair	Count	Pair	Count
␣␣	5728	*st*	442
nl␣	1471	*le*	440
;nl	1133	*ut*	440
in	985	*f(*	416
nt	739	*ar*	381
=␣	687	*or*	374
␣i	662	*r␣*	373
t␣	615	*en*	371
␣=	612	*er*	358
);	558	*ri*	357
,␣	554	*at*	352
nlnl	506	*pr*	351
␣f	505	*te*	349
e␣	500	*an*	348
*␣**	444	*lo*	347

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, generally we want techniques that will be able to compress a variety of source outputs. If we wanted 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.

5.4 Adaptive Dictionary

Most adaptive-dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 [54] and 1978 [55]. 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 approach followed by some of the more well-known variations.

5.4.1 The 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 *look-ahead buffer* that contains the next portion of the sequence to be encoded. In Figure 5.1, the search buffer contains eight symbols, while the look-ahead buffer contains seven symbols. In practice, the sizes of the buffers are significantly larger; however, for the purpose of explanation, we will keep the buffer sizes small.

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

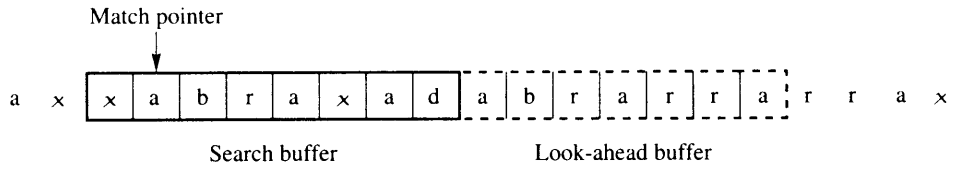


FIGURE 5.1 Encoding using the LZ77 approach.

buffer. The distance of the pointer from the look-ahead 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 look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead 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 $\langle o, l, c \rangle$, where o is the offset, l is the length of the match, and c is the codeword corresponding to the symbol in the look-ahead 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 look-ahead 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 look-ahead 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.

If the size of the search buffer is S, the size of the window (search and look-ahead buffers) is W, and the size of the source alphabet is A, then the number of bits needed to code the triple using fixed-length codes is $\lceil \log_2 S \rceil + \lceil \log_2 W \rceil + \lceil \log_2 A \rceil$. Notice that the second term is $\lceil \log_2 W \rceil$, not $\lceil \log_2 S \rceil$. The reason for this is that the length of the match can actually exceed the length of the search buffer. We will see how this happens in Example 5.4.1.

In the following example, we will look at three different possibilities that may be encountered during the coding process:

1. There is no match for the next character to be encoded in the window.
2. There is a match.
3. The matched string extends inside the look-ahead buffer.

Example 5.4.1: The LZ77 approach

Suppose the sequence to be encoded is

$$\ldots cabracadabrarrarrad\ldots$$

Suppose the length of the window is 13, the size of the look-ahead buffer is six, and the current condition is as follows:

cabraca	*dabrar*

with *dabrar* in the look-ahead 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 $\langle 0, 0, C(d) \rangle$. 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 look-ahead 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 $\langle 7, 4, C(r) \rangle$, and move the window forward by five characters. The window now contains the following characters:

adabrar	*rarrad*

Now the look-ahead 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.

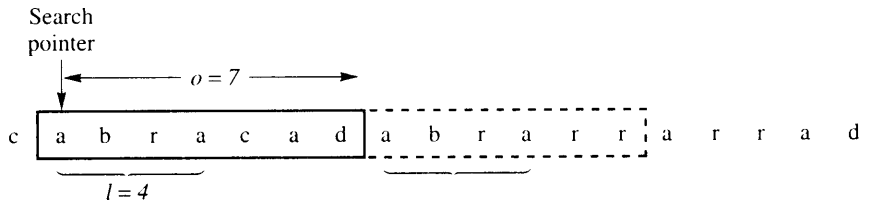


FIGURE 5. 2 The encoding process.

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 $\langle 0, 0, C(d) \rangle$, $\langle 7, 4, C(r) \rangle$, and $\langle 3, 5, C(d) \rangle$. 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 $\langle 3, 5, C(d) \rangle$ 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 look-ahead buffer. In fact, if the last character in the look-ahead buffer

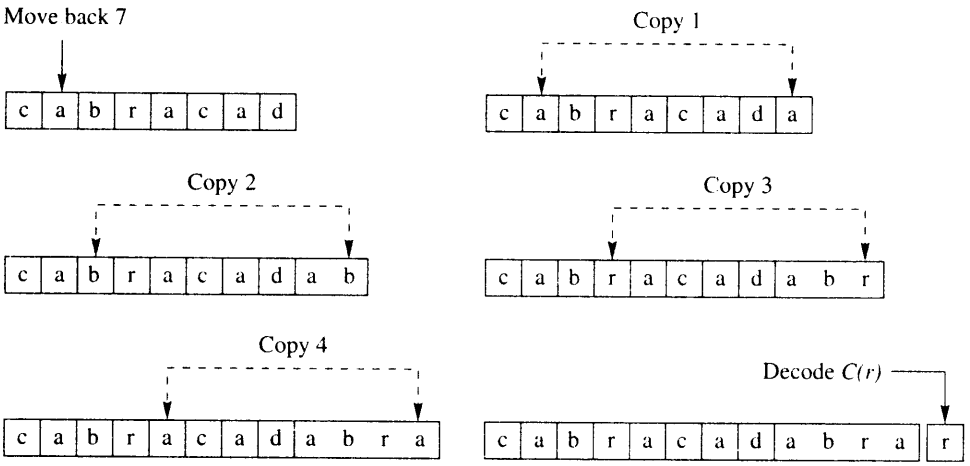


FIGURE 5. 3 **Decoding of the triple ⟨7, 4, C(r)⟩.**

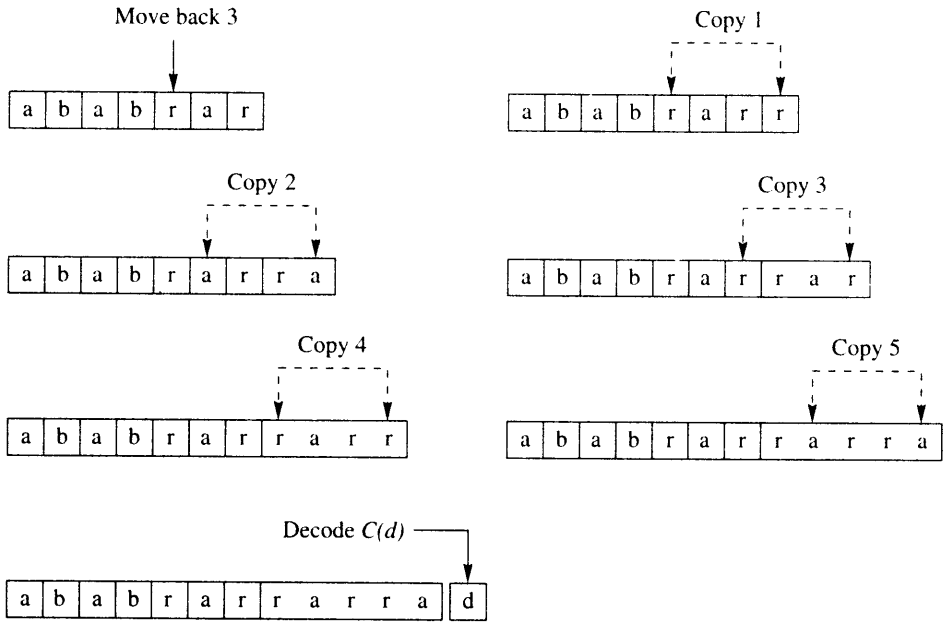


FIGURE 5. 4 **Decoding the triple ⟨3, 5, C(d)⟩.**

had been r instead of d, followed by several more repetitions of rar, the entire sequence of repeated rars 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 asymptotically the performance of this algorithm 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, in LZ78 the authors removed this "assumption" and came up with an entirely different adaptive-dictionary-based scheme. Before we get to that, let us look at the different variations of the LZ77 algorithm.

Variations on the LZ77 Theme

There are a number of ways that the LZ77 scheme can be made more efficient, and most of these have appeared in the literature. Many of the improvements deal with the efficient encoding of the triples. In the description of the LZ77 algorithm, we assumed that the triples were encoded using a fixed-length code. However, if we were willing to accept more complexity, we could encode the triples using variable-length codes. As we saw in earlier chapters, these codes can be adaptive or, if we were willing to use a two-pass algorithm, they can be semiadaptive. Popular compression packages, such as PKZip, Zip, LHarc, PNG, gzip, and ARJ, all use an LZ77-based algorithm followed by a variable-length coder.

Other variations on the LZ77 algorithm include varying the size of the search and look-ahead buffers. To make the search buffer large requires the development of more effective search strategies. Such strategies can be implemented more effectively if the contents of the search buffer are stored in a manner conducive to fast searches.

The simplest modification to the LZ77 algorithm, and one that is used by most variations of the LZ77 algorithm, is to eliminate the situation where we use a triple to encode a single character. Use of a triple is highly inefficient, especially if a large number of characters occur infrequently. The modification to get rid of this inefficiency is simply the addition of a flag bit, to indicate whether what follows is the codeword for a single symbol. By using this flag bit we also get rid of the necessity for the third element of the triple. Now all we need to do is to send a pair of values corresponding to the offset and length of match. This modification to the LZ77 algorithm is referred to as LZSS [56, 57].

5.4.2 The 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.

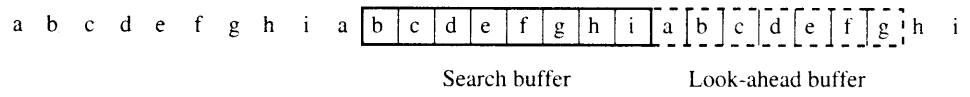


FIGURE 5.5 The Achilles' heel of LZ77.

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.

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 $\langle i, c \rangle$, 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 5.4.2: The LZ78 approach

Let us encode the following sequence using the LZ78 approach:

$$wabba\textvisiblespace wabba\textvisiblespace wabba\textvisiblespace wabba\textvisiblespace woo\textvisiblespace woo\textvisiblespace woo$$[2]

where $\textvisiblespace$ 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 $\langle 0, C(w) \rangle$, $\langle 0, C(a) \rangle$, $\langle 0, C(b) \rangle$, 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 $\langle 3, C(a) \rangle$, 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

[2] "The Monster Song" from *Sesame Street*.

TABLE 5.4 The initial dictionary.

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

TABLE 5.5 Development of dictionary.

	Dictionary	
Encoder Output	Index	Entry
$\langle 0, C(w)\rangle$	1	*w*
$\langle 0, C(a)\rangle$	2	*a*
$\langle 0, C(b)\rangle$	3	*b*
$\langle 3, C(a)\rangle$	4	*ba*
$\langle 0, C(\textvisiblespace)\rangle$	5	␣
$\langle 1, C(a)\rangle$	6	*wa*
$\langle 3, C(b)\rangle$	7	*bb*
$\langle 2, C(\textvisiblespace)\rangle$	8	*a*␣
$\langle 6, C(b)\rangle$	9	*wab*
$\langle 4, C(\textvisiblespace)\rangle$	10	*ba*␣
$\langle 9, C(b)\rangle$	11	*wabb*
$\langle 8, C(w)\rangle$	12	*a*␣*w*
$\langle 0, C(o)\rangle$	13	*o*
$\langle 13, C(\textvisiblespace)\rangle$	14	*o*␣
$\langle 1, C(o)\rangle$	15	*wo*
$\langle 14, C(w)\rangle$	16	*o*␣*w*
$\langle 13, C(o)\rangle$	17	*oo*

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.

Variations on the LZ78 Theme—The LZW Algorithm

There are a number of ways the LZ78 algorithm can be modified, and as is the case with the LZ77 algorithm, anything that can be modified probably has been. The most well-known modification, one that initially sparked much of the interest in the LZ algorithms, is a modification by Terry Welch known as LZW [58]. Welch proposed a technique for removing

the necessity of encoding the second element of the pair $\langle i, c \rangle$. That is, the encoder would only send the index to the dictionary. In order to do this, the dictionary has to be primed with all the letters of the source alphabet. The input to the encoder is accumulated in a pattern p as long as p is contained in the dictionary. If the addition of another letter a results in a pattern $p * a$ ($*$ denotes concatenation) that is not in the dictionary, then the index of p is transmitted to the receiver, the pattern $p * a$ is added to the dictionary, and we start another pattern with the letter a. The LZW algorithm is best understood with an example. In the following two examples, we will look at the encoder and decoder operations for the same sequence used to explain the LZ78 algorithm.

Example 5.4.3: The LZW algorithm—encoding

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

$$wabbab\not{b}wabbab\not{b}wabbab\not{b}wabbab\not{b}woo\not{b}woo\not{b}woo$$

Assuming that the alphabet for the source is $\{\not{b}, a, b, o, w\}$, the LZW dictionary initially looks like Table 5.6.

TABLE 5.6 Initial LZW dictionary.

Index	Entry
1	$\not{b}$
2	a
3	b
4	o
5	w

The encoder first encounters the letter w. This "pattern" is in the dictionary so we concatenate 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 second $wabba$. At this point the output of the encoder consists entirely of indices from the initial dictionary: 5 2 3 3 2 1. The dictionary at this point 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 of the 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.

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

Index	Entry
1	*b*
2	*a*
3	*b*
4	*o*
5	*w*
6	*wa*
7	*ab*
8	*bb*
9	*ba*
10	*ab*
11	*bw*
12	*w...*

TABLE 5.8 The LZW dictionary for encoding wabbabwabbabwabbabwoobwoobwoo.

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

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. ♦

Example 5.4.4: 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, in order to mimic the dictionary construction procedure of the encoder, we begin construction of the next element of the dictionary. 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 $bba\flat$ and generate the dictionary entries ab, bb, ba, and $a\flat$. The dictionary now looks like Table 5.9, where the 11th entry is under construction.

TABLE 5.9 Constructing the 11th entry of the LZW dictionary while decoding.

Index	Entry
1	$\flat$
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb
9	ba
10	$a\flat$
11	$\flat$...

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 $\flat$, and form the pattern $\flat w$. As $\flat w$ 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 $\mathcal{A} = \{a, b\}$, 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 1 2 3 5.... 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.

We begin with the same initial dictionary as the encoder (Table 5.10). The first two elements in the received sequence 1 2 3 5... 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.

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*. . .

TABLE 5.14 Constructing the fifth entry (stage two).

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

The next input to the decoder is 3. This corresponds to the dictionary entry *ab*. Decoding each in turn, we first conçatenate *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.

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. (Of course, if it were, we would not now be studying LZW.) 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.

TABLE 5.15 Completion of the fifth entry.

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

5.5 Applications

Since the publication of Terry Welch's article [58], 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 of the best-known applications of LZW: GIF, and V.42 bis. While the LZW algorithm was initially the algorithm of choice patent concerns has lead 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.

5.5.1 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 $2^{b_{\max}}$ 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.

5.5.2 Image Compression—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

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

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.

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 straight forward 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 PNG standard described next takes advantage of the fact that in natural images the pixel-to-pixel variation is generally small to develop an appropriate preprocessor. We will also revisit this subject in Chapter 7.

5.5.3 Image Compression—Portable Network Graphics (PNG)

The PNG standard is one of the first standards to be collaboratively developed over the Internet. The impetus for it was an announcement in December 1994 by Unisys (which had acquired the patent for LZW from Sperry) and CompuServe that they would start charging

royalties to authors of software that included support for GIF. The announcement resulted in an uproar in the segment of the compression community that formed the core of the Usenet group comp.compression. The community decided that a patent-free replacement for GIF should be developed, and within three months PNG was born. (For a more detailed history of PNG as well as software and much more, go to the PNG website maintained by Greg Roelof, http://www.libpng.org/pub/png/.)

Unlike GIF, the compression algorithm used in PNG is based on LZ77. In particular, it is based on the *deflate* [59] implementation of LZ77. This implementation allows for match lengths of between 3 and 258. At each step the encoder examines three bytes. If it cannot find a match of at least three bytes it puts out the first byte and examines the next three bytes. So, at each step it either puts out the value of a single byte, or literal, or the pair $< match\ length,\ offset >$. The alphabets of the *literal* and *match length* are combined to form an alphabet of size 286 (indexed by $0--285$). The indices $0--255$ represent literal bytes and the index 256 is an end-of-block symbol. The remaining 29 indices represent codes for ranges of lengths between 3 and 258, as shown in Table 5.17. The table shows the index, the number of selector bits to follow the index, and the lengths represented by the index and selector bits. For example, the index 277 represents the range of lengths from 67 to 82. To specify which of the sixteen values has actually occurred, the code is followed by four selector bits.

The index values are represented using a Huffman code. The Huffman code is specified in Table 5.18.

The *offset* can take on values between 1 and 32,768. These values are divided into 30 ranges. The thirty range values are encoded using a Huffman code (different from the Huffman code for the *literal* and *length* values) and the code is followed by a number of selector bits to specify the particular distance within the range.

We have mentioned earlier that in natural images there is not great deal of repetition of sequences of pixel values. However, pixel values that are spatially close also tend to have values that are similar. The PNG standard makes use of this structure by estimating the value of a pixel based on its causal neighbors and subtracting this estimate from the pixel. The difference modulo 256 is then encoded in place of the original pixel. There are four different ways of getting the estimate (five if you include no estimation), and PNG allows

TABLE 5.17 Codes for representations of *match length* [59].

Index	# of selector bits	Length	Index	# of selector bits	Length	Index	# of selector bits	Length
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11, 12	275	3	51–58	285	0	258
266	1	13, 14	276	3	59–66			

TABLE 5.18 Huffman codes for the *match length* alphabet [59].

Index Ranges	# of bits	Binary Codes
0–143	8	00110000 through 10111111
144–255	9	110010000 through 111111111
256–279	7	0000000 through 0010111
280–287	8	11000000 through 11000111

TABLE 5.19 Comparison of PNG with GIF and arithmetic coding.

Image	PNG	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	31,577	51,085	53,431	31,847
Sensin	34,488	60,649	58,306	37,126
Earth	26,995	34,276	38,248	32,137
Omaha	50,185	61,580	56,061	51,393

the use of a different method of estimation for each row. The first way is to use the pixel from the row above as the estimate. The second method is to use the pixel to the left as the estimate. The third method uses the average of the pixel above and the pixel to the left. The final method is a bit more complex. An initial estimate of the pixel is first made by adding the pixel to the left and the pixel above and subtracting the pixel to the upper left. Then the pixel that is closest to the initial esitmate (upper, left, or upper left) is taken as the estimate. A comparison of the performance of PNG and GIF on our standard image set is shown in Table 5.19. The PNG method clearly outperforms GIF.

5.5.4 Compression over Modems—V.42 bis

The ITU-T Recommendation V.42 bis is a compression standard devised for use over a telephone 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.

The V.42 bis recommendation suggests periodic testing of the output of the compression algorithm to see if data expansion is taking place. The exact nature of the test is not specified in the recommendation.

In the compressed mode, the system uses LZW compression with a variable-size dictionary. The initial dictionary size is negotiated at the time a link is established between the transmitter and receiver. The V.42 bis recommendation suggests a value of 2048 for the dictionary size. It specifies that the minimum size of the dictionary is to be 512. Suppose the initial negotiations result in a dictionary size of 512. This means that our codewords that are indices into the dictionary will be 9 bits long. Actually, the entire 512 indices do not correspond to input strings; three entries in the dictionary are reserved for control codewords. These codewords in the compressed mode are shown in Table 5.20.

When the numbers of entries in the dictionary exceed a prearranged threshold C_3, the encoder sends the STEPUP control code, and the codeword size is incremented by 1 bit. At the same time, the threshold C_3 is also doubled. When all available dictionary entries are filled, the algorithm initiates a reuse procedure. The location of the first string entry in the dictionary is maintained in a variable N_5. Starting from N_5, a counter C_1 is incremented until it finds a dictionary entry that is not a prefix to any other dictionary entry. The fact that this entry is not a prefix to another dictionary entry means that this pattern has not been encountered since it was created. Furthermore, because of the way it was located, among patterns of this kind this pattern has been around the longest. This reuse procedure enables the algorithm to prune the dictionary of strings that may have been encountered in the past but have not been encountered recently, on a continual basis. In this way the dictionary is always matched to the current source statistics.

To reduce the effect of errors, the CCITT recommends setting a maximum string length. This maximum length is negotiated at link setup. The CCITT recommends a range of 6–250, with a default value of 6.

The V.42 bis recommendation avoids the need for an exception handler for the case where the decoder receives a codeword corresponding to an incomplete entry by forbidding the use of the last entry in the dictionary. Instead of transmitting the codeword corresponding to the last entry, the recommendation requires the sending of the codewords corresponding

TABLE 5.20 Control codewords in compressed mode.

Codeword	Name	Description
0	ETM	Enter transparent mode
1	FLUSH	Flush data
2	STEPUP	Increment codeword size

to the constituents of the last entry. In the example used to demonstrate this quirk of the LZW algorithm, instead of transmitting the codeword 5, the V.42 bis recommendation would have forced us to send the codewords 3 and 1.

5.6 Summary

In this chapter we have introduced techniques that keep a dictionary of recurring patterns and transmit the index of those patterns instead of the patterns themselves in order to achieve compression. There are a number of ways the dictionary can be constructed.

- In applications where certain patterns consistently recur, we can build application-specific static dictionaries. Care should be taken not to use these dictionaries outside their area of intended application. Otherwise, we may end up with data expansion instead of data compression.
- The dictionary can be the source output itself. This is the approach used by the LZ77 algorithm. When using this algorithm, there is an implicit assumption that recurrence of a pattern is a local phenomenon.
- This assumption is removed in the LZ78 approach, which dynamically constructs a dictionary from patterns observed in the source output.

Dictionary-based algorithms are being used to compress all kinds of data; however, care should be taken with their use. This approach is most useful when structural constraints restrict the frequently occurring patterns to a small subset of all possible patterns. This is the case with text, as well as computer-to-computer communication.

Further Reading

1. *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1], provides an excellent exposition of dictionary-based coding techniques.
2. *The Data Compression Book*, by M. Nelson and J.-L. Gailley [60], also does a good job of describing the Ziv-Lempel algorithms. There is also a very nice description of some of the software implementation aspects.
3. *Data Compression*, by G. Held and T.R. Marshall [61], contains a description of digram coding under the name "diatomic coding." The book also includes BASIC programs that help in the design of dictionaries.
4. The PNG algorithm is described in a very accessible manner in "PNG Lossless Compression," by G. Roelofs [62] in the *Lossless Compression Handbook*.
5. A more in-depth look at dictionary compression is provided in "Dictionary-Based Data Compression: An Algorithmic Perspective," by S.C. Şahinalp and N.M. Rajpoot [63] in the *Lossless Compression Handbook*.

5.7 Projects and Problems

1. To study the effect of dictionary size on the efficiency of a static dictionary technique, we can modify Equation (5.1) so that it gives the rate as a function of both p and the dictionary size M. Plot the rate as a function of p for different values of M, and discuss the trade-offs involved in selecting larger or smaller values of M.

2. Design and implement a digram coder for text files of interest to you.

   (a) Study the effect of the dictionary size, and the size of the text file being encoded on the amount of compression.

   (b) Use the digram coder on files that are not similar to the ones you used to design the digram coder. How much does this affect your compression?

3. Given an initial dictionary consisting of the letters *a b r y* ␣, encode the following message using the LZW algorithm: *a␣bar␣barray␣by␣barrayar␣bay*.

4. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.21.

TABLE 5.21 Initial dictionary for Problem 4.

Index	Entry
1	*a*
2	␣
3	*h*
4	*i*
5	*s*
6	*t*

   (a) The output of the LZW encoder is the following sequence:

6	3	4	5	2	3	1	6	2	9	11	16	12	14	4	20	10	8	23	13

   Decode this sequence.

   (b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

5. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.22.

   (a) The output of the LZW encoder is the following sequence:

3	1	4	6	8	4	2	1	2	5	10	6	11	13	6

   Decode this sequence.

TABLE 5.22 Initial dictionary for Problem 5.

Index	Entry
1	*a*
2	*b*
3	*r*
4	*t*

(b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

6. Encode the following sequence using the LZ77 algorithm:

barrayarbbarbbybbarrayarbbay

Assume you have a window size of 30 with a look-ahead buffer of size 15. Furthermore, assume that $C(a) = 1$, $C(b) = 2$, $C(b) = 3$, $C(r) = 4$, and $C(y) = 5$.

7. A sequence is encoded using the LZ77 algorithm. Given that $C(a) = 1$, $C(b) = 2$, $C(r) = 3$, and $C(t) = 4$, decode the following sequence of triples:

$$\langle 0, 0, 3\rangle\ \langle 0, 0, 1\rangle\ \langle 0, 0, 4\rangle\ \langle 2, 8, 2\rangle\ \langle 3, 1, 2\rangle\ \langle 0, 0, 3\rangle\ \langle 6, 4, 4\rangle\ \langle 9, 5, 4\rangle$$

Assume that the size of the window is 20 and the size of the look-ahead buffer is 10. Encode the decoded sequence and make sure you get the same sequence of triples.

8. Given the following primed dictionary and the received sequence below, build an LZW dictionary *and* decode the transmitted sequence.

Received Sequence: 4, 5, 3, 1, 2, 8, 2, 7, 9, 7, 4

Decoded Sequence:_____

Initial dictionary:

(a) S

(b) *b*

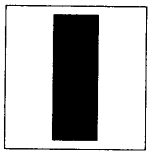
(c) I

(d) T

(e) H

Context-Based Compression

6.1 Overview



n this chapter we present a number of techniques that use minimal prior assumptions about the statistics of the data. Instead they use the context of the data being encoded and the past history of the data to provide more efficient compression. We will look at a number of schemes that are principally used for the compression of text. These schemes use the context in which the data occurs in different ways.

6.2 Introduction

In Chapters 3 and 4 we learned that we get more compression when the message that is being coded has a more skewed set of probabilities. By "skewed" we mean that certain symbols occur with much higher probability than others in the sequence to be encoded. So it makes sense to look for ways to represent the message that would result in greater skew. One very effective way to do so is to look at the probability of occurrence of a letter in the context in which it occurs. That is, we do not look at each symbol in a sequence as if it had just happened out of the blue. Instead, we examine the history of the sequence before determining the likely probabilities of different values that the symbol can take.

In the case of English text, Shannon [8] showed the role of context in two very interesting experiments. In the first, a portion of text was selected and a subject (possibly his wife, Mary Shannon) was asked to guess each letter. If she guessed correctly, she was told that she was correct and moved on to the next letter. If she guessed incorrectly, she was told the correct answer and again moved on to the next letter. Here is a result from one of these experiments. Here the dashes represent the letters that were correctly guessed.

Actual Text	THE ROOM WAS NOT VERY LIGHT A SMALL OBLONG
Subject Performance	____ROO_____NOT_V_____I_____SM____OBL___

Notice that there is a good chance that the subject will guess the letter, especially if the letter is at the end of a word or if the word is clear from the context. If we now represent the original sequence by the subject performance, we would get a very different set of probabilities for the values that each element of the sequence takes on. The probabilities are definitely much more skewed in the second row: the "letter" _ occurs with high probability. If a mathematical twin of the subject were available at the other end, we could send the "reduced" sentence in the second row and have the twin go through the same guessing process to come up with the original sequence.

In the second experiment, the subject was allowed to continue guessing until she had guessed the correct letter and the number of guesses required to correctly predict the letter was noted. Again, most of the time the subject guessed correctly, resulting in 1 being the most probable number. The existence of a mathematical twin at the receiving end would allow this skewed sequence to represent the original sequence to the receiver. Shannon used his experiments to come up with upper and lower bounds for the English alphabet (1.3 bits per letter and 0.6 bits per letter, respectively).

The difficulty with using these experiments is that the human subject was much better at predicting the next letter in a sequence than any mathematical predictor we can develop. Grammar is hypothesized to be innate to humans [64], in which case development of a predictor as efficient as a human for language is not possible in the near future. However, the experiments do provide an approach to compression that is useful for compression of all types of sequences, not simply language representations.

If a sequence of symbols being encoded does not consist of independent occurrences of the symbols, then the knowledge of which symbols have occurred in the neighborhood of the symbol being encoded will give us a much better idea of the value of the symbol being encoded. If we know the context in which a symbol occurs we can guess with a much greater likelihood of success what the value of the symbol is. This is just another way of saying that, given the context, some symbols will occur with much higher probability than others. That is, the probability distribution given the context is more skewed. If the context is known to both encoder and decoder, we can use this skewed distribution to perform the encoding, thus increasing the level of compression. The decoder can use its knowledge of the context to determine the distribution to be used for decoding. If we can somehow group like contexts together, it is quite likely that the symbols following these contexts will be the same, allowing for the use of some very simple and efficient compression strategies. We can see that the context can play an important role in enhancing compression, and in this chapter we will look at several different ways of using the context.

Consider the encoding of the word *probability*. Suppose we have already encoded the first four letters, and we want to code the fifth letter, a. If we ignore the first four letters, the probability of the letter a is about 0.06. If we use the information that the previous letter is b, this reduces the probability of several letters such as q and z occurring and boosts the probability of an a occurring. In this example, b would be the first-order context for a, ob would be the second-order context for a, and so on. Using more letters to define the context in which a occurs, or higher-order contexts, will generally increase the probability

of the occurrence of a in this example, and hence reduce the number of bits required to encode its occurrence. Therefore, what we would like to do is to encode each letter using the probability of its occurrence with respect to a context of high order.

If we want to have probabilities with respect to all possible high-order contexts, this might be an overwhelming amount of information. Consider an alphabet of size M. The number of first-order contexts is M, the number of second-order contexts is M^2, and so on. Therefore, if we wanted to encode a sequence from an alphabet of size 256 using contexts of order 5, we would need 256^5, or about 1.09951×10^{12} probability distributions! This is not a practical alternative. A set of algorithms that resolve this problem in a very simple and elegant way is based on the *prediction with partial match (ppm)* approach. We will describe this in the next section.

6.3 Prediction with Partial Match (*ppm*)

The best-known context-based algorithm is the *ppm* algorithm, first proposed by Cleary and Witten [65] 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 development of more efficient variants, *ppm*-based algorithms are becoming increasingly more 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.

6.3.1 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 a 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*, 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, each time a symbol is encountered, the count corresponding to that symbol is updated. 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 6.3.1:

Let's encode the sequence

this␣is␣the␣tithe

Assuming we have already encoded the initial seven characters *this␣is*, 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 *this␣is* has already been encoded, the next letter to be encoded is *␣*. The second-order context for this letter is *is*. Looking at Table 6.4, we can see that the letter *␣*

TABLE 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
␣	1	6
Total Count		6

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
␣	1	7
⟨*Esc*⟩	1	8
Total Count		8

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
␣	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
s	*␣*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

is the first letter in this context with a *Cum_Count* value of 1. As the *Total_Count* in this case is 2, the update equations for the lower and upper limits are

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

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor - 1 = 31 = 011111.$$

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	*␣*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
s␣	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
␣i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

As the MSBs of both l and u are the same, we shift that bit out, 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

$$\begin{aligned} \text{Transmitted sequence}: &\ 0 \\ l: &\ 000000 \\ u: &\ 111111 \end{aligned}$$

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 *s␣*. 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:

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

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 $\flat$. 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

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

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:

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

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{9} \right\rfloor - 1 = 6 = 000110.$$

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
$\flat$	2	8
⟨*Esc*⟩	1	9
Total Count		9

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 $\flat t$ 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

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
$\flat$	2	10
$\langle Esc \rangle$	1	11
Total Count		11

TABLE 6.7 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	h	2	2
	$\langle Esc \rangle$	1	3
Total Count			3
h	i	1	1
	$\langle Esc \rangle$	1	2
Total Count			2
i	s	2	2
	$\langle Esc \rangle$	1	3
Total Count			3
$\flat$	i	1	1
	t	1	2
	$\langle Esc \rangle$	1	3
Total Count			3
s	$\flat$	2	2
	$\langle Esc \rangle$	1	3
Total Count			3

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

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.

6.3.2 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

TABLE 6.9 Counts using Method A.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	⟨*Esc*⟩	1
Total Count		23

TABLE 6.10 Counts using Method B.

Context	Symbol	Count
prob	*a*	9
	l	8
	o	2
	⟨*Esc*⟩	3
Total Count		22

sequence *a* occurs 10 times in the context of *prob*, *l* occurs 9 times, and *o* occurs 3 times in the same 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. The 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 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 [66]. 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 the performance depending on the characteristics of the data being encoded, of the three methods for assigning counts to the escape symbol, on the average, Method C seems to provide the best performance.

6.3.3 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

TABLE 6.11 Counts using Method C.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	⟨*Esc*⟩	3
Total Count		25

in a higher probability if the symbol to be encoded has a nonzero count with respect to that context. However, 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.

An alternative to the policy of a fixed maximum length is used in the algorithm *ppm** [67]. This algorithm uses the fact that long contexts that give only a single prediction are seldom followed by a new symbol. If *mike* has always been followed by *y* in the past, it will probably not be followed by *b* the next time it is encountered. Contexts that are always followed by the same symbol are called *deterministic* contexts. The *ppm** algorithm first looks for the longest deterministic context. If the symbol to be encoded does not occur in that context, an escape symbol is encoded and the algorithm defaults to the maximum context length. This approach seems to provide a small but significant amount of improvement over the basic algorithm. Currently, the best variant of the *ppm** algorithm is the *ppmz* algorithm by Charles Bloom. Details of the *ppmz* algorithm as well as implementations of the algorithm can be found at *http://www.cbloom.com/src/ppmz.html*.

6.3.4 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,

TABLE 6.12 Counts for exclusion example.

Context	Symbol	Count
ob	*l*	10
	o	3
	⟨*Esc*⟩	2
Total Count		15
b	*l*	5
	o	3
	a	4
	r	2
	e	2
	⟨*Esc*⟩	5
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

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.

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 often together they provide the margin of compression that makes a particular scheme competitive.

6.4 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 [68]. 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 6.4.1:

Let's encode the sequence

this␣is␣the

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.

TABLE 6.14 Permutations of *this␣is␣the*.

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

TABLE 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

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

$$L : sshtthbiibe$$

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

$$F : bbehhiisstt$$

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.

Example 6.4.2:

In our example

$$F = \begin{bmatrix} \not{b} \\ \not{b} \\ e \\ h \\ h \\ i \\ i \\ s \\ s \\ t \\ t \end{bmatrix} \quad L = \begin{bmatrix} s \\ s \\ h \\ t \\ t \\ h \\ \not{b} \\ i \\ i \\ \not{b} \\ e \end{bmatrix}$$

the original sequence is sequence number 10, so the first letter in of 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.

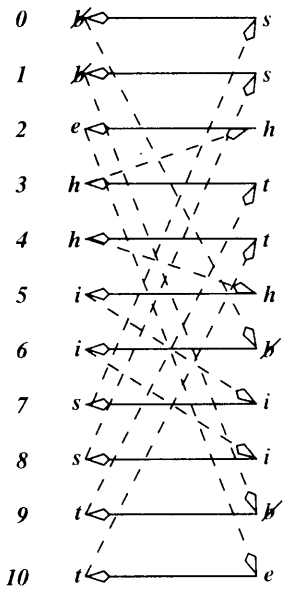


FIGURE 6. 1 Decoding process. ♦

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. Consider all the rows of A beginning with *heb*. With high probability *heb* would be preceded by t. Therefore, in L we would get a long run of ts.

6.4.1 Move-to-Front Coding

A coding scheme that takes advantage of long runs of identical symbols is the 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 6.4.3:

Let's encode $L = sshtthbiibe$. Let's assume that the source alphabet is given by

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

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. ♦

6.5 Associative Coder of Buyanovsky (ACB)

A different approach to using contexts for compression is employed by the eponymous compression utility developed by George Buyanovsky. The details of this very efficient coder are not well known; however, the way the context is used is interesting and we will briefly describe this aspect of ACB. More detailed descriptions are available in [69] and [70]. The ACB coder develops a sorted dictionary of all encountered contexts. In this it is similar to other context based encoders. However, it also keeps track of the *contents* of these contexts. The content of a context is what appears after the context. In a traditional left-to-right reading of text, the contexts are unbounded to the left and the contents to the right (to the limits of text that has already been encoded). When encoding the coder searches for the longest match to the current context reading right to left. This again is not an unusual thing to do. What is interesting is what the coder does after the best match is found. Instead of simply examining the *content* corresponding to the best matched context, the coder also examines the *contents* of the coders in the neighborhood of the best matched contexts. Fenwick [69] describes this process as first finding an anchor point then searching the *contents* of the neighboring contexts for the best match. The location of the anchor point is known to both the encoder and the decoder. The location of the best *content* match is signalled to the decoder by encoding the offset δ of the context of this *content* from the anchor point. We have not specified what we mean by "best" match. The coder takes the utilitarian approach that the best match is the one that ends up providing the most compression. Thus, a longer match farther away from the anchor may not be as advantageous as a shorter match closer to the anchor because of the number of bits required to encode δ. The length of the match λ is also sent to the decoder.

The interesting aspect of this scheme is that it moves away from the idea of exactly matching the past. It provides a much richer environment and flexibility to enhance the compression and will, hopefully, provide a fruitful avenue for further research.

6.6 Dynamic Markov Compression

Quite often the probabilities of the value that the next symbol in a sequence takes on depend not only on the current value but on the past values as well. The *ppm* scheme relies on this longer-range correlation. The *ppm* scheme, in some sense, reflects the application, that is, text compression, for which it is most used. Dynamic Markov compression (DMC), introduced by Cormack and Horspool [71], uses a more general framework to take advantage of relationships and correlations, or contexts, that extend beyond a single symbol.

Consider the sequence of pixels in a scanned document. The sequence consists of runs of black and white pixels. If we represent black by 0 and white by 1, we have runs of 0s and 1s. If the current value is 0, the probability that the next value is 0 is higher than if the current value was 1. The fact that we have two different sets of probabilities is reflected in the two-state model shown in Figure 6.2. Consider state A. The probability of the next value being 1 changes depending on whether we reached state A from state B or from state A itself. We can have the model reflect this by *cloning* state A, as shown in Figure 6.3, to create state A'. Now if we see a white pixel after a run of black pixels, we go to state A'. The probability that the next value will be 1 is very high in this state. This way, when we estimate probabilities for the next pixel value, we take into account not only the value of the current pixel but also the value of the previous pixel.

This process can be continued as long as we wish to take into account longer and longer histories. "As long as we wish" is a rather vague statement when it comes to implementing the algorithm. In fact, we have been rather vague about a number of implementation issues. We will attempt to rectify the situation.

There are a number of issues that need to be addressed in order to implement this algorithm:

1. What is the initial number of states?
2. How do we estimate probabilities?

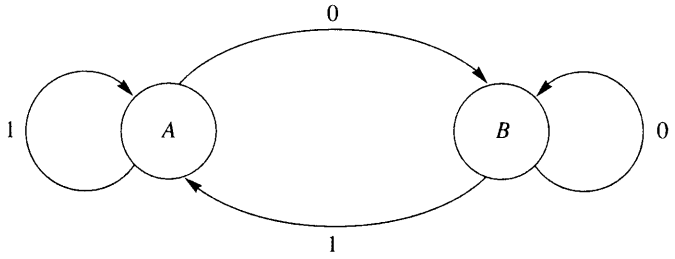


FIGURE 6.2 **A two-state model for binary sequences.**

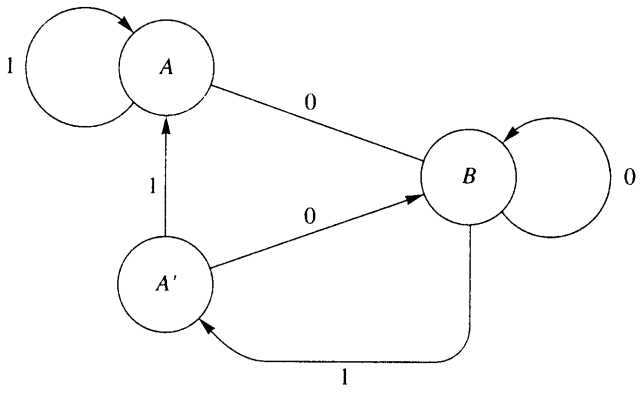


FIGURE 6.3 A three-state model obtained by cloning.

3. How do we decide when a state needs to be cloned?
4. What do we do when the number of states becomes too large?

Let's answer each question in turn.

We can start the encoding process with a single state with two self-loops for 0 and 1. This state can be cloned to two and then a higher number of states. In practice it has been found that, depending on the particular application, it is more efficient to start with a larger number of states than one.

The probabilities from a given state can be estimated by simply counting the number of times a 0 or a 1 occurs in that state divided by the number of times the particular state is occupied. For example, if in state V the number of times a 0 occurs is denoted by n_0^V and the number of times a 1 occurs is denoted by n_1^V, then

$$P(0|V) = \frac{n_0^V}{n_0^V + n_1^V}$$

$$P(1|V) = \frac{n_1^V}{n_0^V + n_1^V}.$$

What if a 1 has never previously occurred in this state? This approach would assign a probability of zero to the occurrence of a 1. This means that there will be no subinterval assigned to the possibility of a 1 occurring, and when it does occur, we will not be able to represent it. In order to avoid this, instead of counting from zero, we start the count of 1s and 0s with a small number c and estimate the probabilities as

$$P(0|V) = \frac{n_0^V + c}{n_0^V + n_1^V + 2c}$$

$$P(1|V) = \frac{n_1^V + c}{n_0^V + n_1^V + 2c}.$$

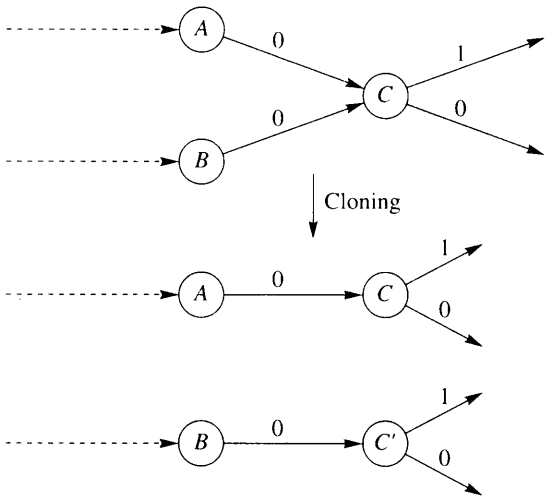


FIGURE 6.4 The cloning process.

Whenever we have two branches leading to a state, it can be cloned. And, theoretically, cloning is never harmful. By cloning we are providing additional information to the encoder. This might not reduce the rate, but it should never result in an increase in the rate. However, cloning does increase the complexity of the coding, and hence the decoding, process. In order to control the increase in the number of states, we should only perform cloning when there is a reasonable expectation of reduction in rate. We can do this by making sure that both paths leading to the state being considered for cloning are used often enough. Consider the situation shown in Figure 6.4. Suppose the current state is A and the next state is C. As there are two paths entering C, C is a candidate for cloning. Cormack and Horspool suggest that C be cloned if $n_0^A > T_1$ and $n_0^B > T_2$, where T_1 and T_2 are threshold values set by the user. If there are more than three paths leading to a candidate for cloning, then we check that both the number of transitions from the current state is greater than T_1 and the number of transitions from all other states to the candidate state is greater than T_2.

Finally, what do we do when, for practical reasons, we cannot accommodate any more states? A simple solution is to restart the algorithm. In order to make sure that we do not start from ground zero every time, we can train the initial state configuration using a certain number of past inputs.

6.7 Summary

The context in which a symbol occurs can be very informative about the value that the symbol takes on. If this context is known to the decoder then this information need not be encoded: it can be inferred by the decoder. In this chapter we have looked at several creative ways in which the knowledge of the context can be used to provide compression.

Further Reading

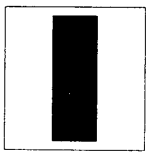
1. The basic *ppm* algorithm is described in detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].
2. For an excellent description of Burrows-Wheeler Coding, including methods of implementation and improvements to the basic algorithm, see "Burrows-Wheeler Compression," by P. Fenwick [72] in *Lossless Compression Handbook*.
3. The ACB algorithm is described in "Symbol Ranking and ACB Compression," by P. Fenwick [69] in the *Lossless Compression Handbook*, and in *Data Compression: The Complete Reference* by D. Salomon [70]. The chapter by Fenwick also explores compression schemes based on Shannon's experiments.

6.8 Projects and Problems

1. Decode the bitstream generated in Example 6.3.1. Assume you have already decoded *this␣is* and Tables 6.1–6.4 are available to you.
2. Given the sequence *the␣beta␣cat␣ate␣the␣ceta␣hat*:
   (a) Encode the sequence using the *ppma* algorithm and an adaptive arithmetic coder. Assume a six-letter alphabet $\{h, e, t, a, c, ␣\}$.
   (b) Decode the encoded sequence.
3. Given the sequence *eta␣ceta␣and␣beta␣ceta*:
   (a) Encode using the Burrows-Wheeler transform and move-to-front coding.
   (b) Decode the encoded sequence.
4. A sequence is encoded using the Burrows-Wheeler transform. Given $L = elbkkee$, and index = 5 (we start counting from 1, not 0), find the original sequence.

Lossless Image Compression

7.1 Overview



n this chapter we examine a number of schemes used for lossless compression of images. We will look at schemes for compression of grayscale and color images as well as schemes for compression of binary images. Among these schemes are several that are a part of international standards.

7.2 Introduction

In the previous chapters we have focused on compression techniques. Although some of them may apply to some preferred applications, the focus has been on the technique rather than on the application. However, there are certain techniques for which it is impossible to separate the technique from the application. This is because the techniques rely upon the properties or characteristics of the application. Therefore, we have several chapters in this book that focus on particular applications. In this chapter we will examine techniques specifically geared toward lossless image compression. Later chapters will examine speech, audio, and video compression.

In the previous chapters we have seen that a more skewed set of probabilities for the message being encoded results in better compression. In Chapter 6 we saw how the use of context to obtain a skewed set of probabilities can be especially effective when encoding text. We can also transform the sequence (in an invertible fashion) into another sequence that has the desired property in other ways. For example, consider the following sequence:

1	2	5	7	2	−2	0	−5	−3	−1	1	−2	−7	−4	−2	1	3	4

If we consider this sample to be fairly typical of the sequence, we can see that the probability of any given number being in the range from -7 to 7 is about the same. If we were to encode this sequence using a Huffman or arithmetic code, we would use almost 4 bits per symbol.

Instead of encoding this sequence directly, we could do the following: add two to the previous number in the sequence and send the difference between the current element in the sequence and this *predicted* value. The transmitted sequence would be

1	−1	1	0	−7	−4	0	−7	0	0	0	−5	−7	1	0	1	0	−1

This method uses a rule (add two) and the history (value of the previous symbol) to generate the new sequence. If the rule by which this *residual sequence* was generated is known to the decoder, it can recover the original sequence from the residual sequence. The length of the residual sequence is the same as the original sequence. However, notice that the residual sequence is much more likely to contain 0s, 1s, and -1s than other values. That is, the probability of 0, 1, and -1 will be significantly higher than the probabilities of other numbers. This, in turn, means that the entropy of the residual sequence will be low and, therefore, provide more compression.

We used a particular method of prediction in this example (add two to the previous element of the sequence) that was specific to this sequence. In order to get the best possible performance, we need to find the prediction approach that is best suited to the particular data we are dealing with. We will look at several prediction schemes used for lossless image compression in the following sections.

7.2.1 The Old JPEG Standard

The Joint Photographic Experts Group (JPEG) is a joint ISO/ITU committee responsible for developing standards for continuous-tone still-picture coding. The more famous standard produced by this group is the lossy image compression standard. However, at the time of the creation of the famous JPEG standard, the committee also created a lossless standard [73]. At this time the standard is more or less obsolete, having been overtaken by the much more efficient JPEG-LS standard described later in this chapter. However, the old JPEG standard is still useful as a first step into examining predictive coding in images.

The old JPEG lossless still compression standard [73] provides eight different predictive schemes from which the user can select. The first scheme makes no prediction. The next seven are listed below. Three of the seven are one-dimensional predictors, and four are two-dimensional prediction schemes. Here, $I(i, j)$ is the (i, j)th pixel of the original image, and $\hat{I}(i, j)$ is the predicted value for the (i, j)th pixel.

$$\mathbf{1} \quad \hat{I}(i, j) = I(i-1, j) \tag{7.1}$$

$$\mathbf{2} \quad \hat{I}(i, j) = I(i, j-1) \tag{7.2}$$

$$\mathbf{3} \quad \hat{I}(i, j) = I(i-1, j-1) \tag{7.3}$$

$$\mathbf{4} \quad \hat{I}(i, j) = I(i, j-1) + I(i-1, j) - I(i-1, j-1) \tag{7.4}$$

$$5 \quad \hat{I}(i,j) = I(i,j-1) + (I(i-1,j) - I(i-1,j-1))/2 \tag{7.5}$$

$$6 \quad \hat{I}(i,j) = I(i-1,j) + (I(i,j-1) - I(i-1,j-1))/2 \tag{7.6}$$

$$7 \quad \hat{I}(i,j) = (I(i,j-1) + I(i-1,j))/2 \tag{7.7}$$

Different images can have different structures that can be best exploited by one of these eight modes of prediction. If compression is performed in a nonreal-time environment—for example, for the purposes of archiving—all eight modes of prediction can be tried and the one that gives the most compression is used. The mode used to perform the prediction can be stored in a 3-bit header along with the compressed file. We encoded our four test images using the various JPEG modes. The residual images were encoded using adaptive arithmetic coding. The results are shown in Table 7.1.

The best results—that is, the smallest compressed file sizes—are indicated in bold in the table. From these results we can see that a different JPEG predictor is the best for the different images. In Table 7.2, we compare the best JPEG results with the file sizes obtained using GIF and PNG. Note that PNG also uses predictive coding with four possible predictors, where each row of the image can be encoded using a different predictor. The PNG approach is described in Chapter 5.

Even if we take into account the overhead associated with GIF, from this comparison we can see that the predictive approaches are generally better suited to lossless image compression than the dictionary-based approach when the images are "natural" gray-scale images. The situation is different when the images are graphic images or pseudocolor images. A possible exception could be the Earth image. The best compressed file size using the second JPEG mode and adaptive arithmetic coding is 32,137 bytes, compared to 34,276 bytes using GIF. The difference between the file sizes is not significant. We can see the reason by looking at the Earth image. Note that a significant portion of the image is the

TABLE 7.1 Compressed file size in bytes of the residual images obtained using the various JPEG prediction modes.

Image	JPEG 0	JPEG 1	JPEG 2	JPEG 3	JPEG 4	JPEG 5	JPEG 6	JPEG 7
Sena	53,431	37,220	31,559	38,261	31,055	**29,742**	33,063	32,179
Sensin	58,306	41,298	37,126	43,445	**32,429**	33,463	35,965	36,428
Earth	38,248	32,295	**32,137**	34,089	33,570	33,057	33,072	32,672
Omaha	56,061	**48,818**	51,283	53,909	53,771	53,520	52,542	52,189

TABLE 7.2 Comparison of the file sizes obtained using JPEG lossless compression, GIF, and PNG.

Image	Best JPEG	GIF	PNG
Sena	31,055	51,085	31,577
Sensin	32,429	60,649	34,488
Earth	32,137	34,276	26,995
Omaha	48,818	61,341	50,185

background, which is of a constant value. In dictionary coding, this would result in some very long entries that would provide significant compression. We can see that if the ratio of background to foreground were just a little different in this image, the dictionary method in GIF might have outperformed the JPEG approach. The PNG approach which allows the use of a different predictor (or no predictor) on each row, prior to dictionary coding significantly outperforms both GIF and JPEG on this image.

7.3 CALIC

The Context Adaptive Lossless Image Compression (CALIC) scheme, which came into being in response to a call for proposal for a new lossless image compression scheme in 1994 [74, 75], uses both context and prediction of the pixel values. The CALIC scheme actually functions in two modes, one for gray-scale images and another for bi-level images. In this section, we will concentrate on the compression of gray-scale 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 these factors, the algorithm has to make a determination of the environment of the pixel to be encoded. The only information that can be used to make this determination has to be available to both 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 encoder and decoder.

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

FIGURE 7.1 **Labeling the neighbors of pixel 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
  X̂ ← N
else if d_v - d_h > 80
  X̂ ← W
else
{
      X̂ ← (N + W)/2 + (NE - NW)/4
      if d_h - d_v > 32
        X̂ ← (X̂ + N)/2
      else if d_v - d_h > 32
        X̂ ← (X̂ + W)/2
      else if d_h - d_v > 8
        X̂ ← (3X̂ + N)/4
      else if d_v - d_h > 8
        X̂ ← (3X̂ + 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 provides 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 $\hat{X}$. 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

$$\delta = d_h + d_v + 2|N - \hat{N}| \tag{7.8}$$

where $\hat{N}$ is the predicted value of N. This range of values of δ is divided into four intervals, each being 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 Equation (7.8). 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:

$$0 \le \delta < q_1 \Rightarrow \text{Context 1}$$
$$q_1 \le \delta < q_2 \Rightarrow \text{Context 2}$$
$$q_2 \le \delta < q_3 \Rightarrow \text{Context 3}$$
$$q_3 \le \delta < q_4 \Rightarrow \text{Context 4}$$
$$q_4 \le \delta < q_5 \Rightarrow \text{Context 5}$$
$$q_5 \le \delta < q_6 \Rightarrow \text{Context 6}$$
$$q_6 \le \delta < q_7 \Rightarrow \text{Context 7}$$
$$q_7 \le \delta < q_8 \Rightarrow \text{Context 8}$$

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 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 $\hat{X}$, the difference $X - \hat{X}$ will take on values in the range $-\hat{X}$ to $M - 1 - \hat{X}$. We can use this fact to map the difference values into the range $[0, M-1]$, using the following mapping:

$$\begin{aligned}
0 &\rightarrow 0\\
1 &\rightarrow 1\\
-1 &\rightarrow 2\\
2 &\rightarrow 3\\
&\vdots \quad \vdots\\
-\hat{X} &\rightarrow 2\hat{X}\\
\hat{X}+1 &\rightarrow 2\hat{X}+1\\
\hat{X}+2 &\rightarrow 2\hat{X}+2\\
&\vdots \quad \vdots\\
M-1-\hat{X} &\rightarrow M-1
\end{aligned}$$

where we have assumed that $\hat{X} \leq (M-1)/2$.

Another approach used by CALIC to reduce the size of its alphabet is to use a modification of a technique called *recursive indexing* [76]. 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, would decode it at its face value, and when it saw 7, would keep accumulating the values until a value between 0 and 6 was received. This method of representation followed by entropy coding has been shown to be optimal for sequences that follow a geometric distribution [77].

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 initial prediction $\hat{X}$.
2. Compute prediction context.
3. Refine prediction by removing the estimate of the bias in that context.
4. Update 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. We study such a scheme in the next section.

7.4 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 proposed a much simpler predictive coder, under the name LOCO-I (for low complexity), that still performed close to CALIC [78].

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 NW ≥ max(W, N)
X̂ = max(W, N)
else
{
    if NW ≤ min(W, N)
    X̂ = min(W, N)
    else
    X̂ = W + N − NW
}
```

This prediction approach is a variation of Median Adaptive Prediction [79], 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:

$$\begin{aligned} D_1 &= NE - N \\ D_2 &= N - NW \\ D_3 &= NW - W. \end{aligned}$$

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

$$\begin{aligned} D_i \le -T_3 &\Rightarrow Q_i = -4 \\ -T_3 < D_i \le -T_2 &\Rightarrow Q_i = -3 \\ -T_2 < D_i \le -T_1 &\Rightarrow Q_i = -2 \\ -T_1 < D_i \le 0 &\Rightarrow Q_i = -1 \\ D_i = 0 &\Rightarrow Q_i = 0 \\ 0 < D_i \le T_1 &\Rightarrow Q_i = 1 \\ T_1 < D_i \le T_2 &\Rightarrow Q_i = 2 \\ T_2 < D_i \le T_3 &\Rightarrow Q_i = 3 \\ T_3 < D_i &\Rightarrow Q_i = 4 \end{aligned} \tag{7.9}$$

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 $\mathbf{Q}$ whose first nonzero element is negative by $-\mathbf{Q}$. Whenever

TABLE 7.3 Comparison of the file sizes obtained using new and old JPEG lossless compression standard 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

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 $\mathbf{Q}$ 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 multiplied 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:

$$r_n < -\frac{M}{2} \Rightarrow r_n \leftarrow r_n + M$$

$$r_n > \frac{M}{2} \Rightarrow r_n \leftarrow r_n - M$$

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 may not be very impressive, we should keep in mind that for the old JPEG we are picking the best result out of eight. 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.

7.5 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.

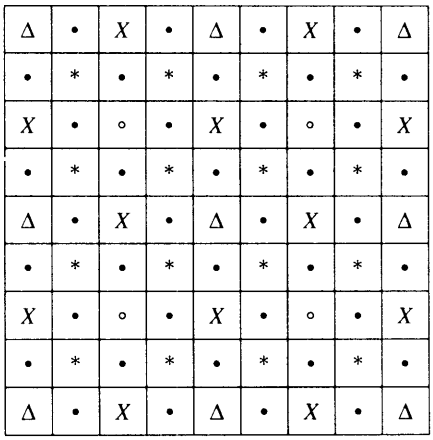


FIGURE 7.2 The HINT scheme for hierarchical prediction.

Multiresolution models generate representations of an image with varying spatial resolution. This usually results in a pyramidlike 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 INTerpolation) [80]. The specific steps involved in HINT are as follows. First, residuals corresponding to the pixels labeled Δ in Figure 7.2 are obtained using linear prediction and transmitted. Then, the intermediate pixels (○) are estimated by linear interpolation, and the error in estimation is then transmitted. Then, the pixels X are estimated from Δ 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. We describe this application in the next section.

7.5.1 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 CAT scans, magnetic resonance images, 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, see [81].)

Suppose a user wants to browse through a number of images in a remote database. The user is connected to the database via a 56 kbits per second (kbps) modem. Suppose the

images are of size 1024×1024, and on the average users have to look through 30 images before finding the image they are looking for. If these images were monochrome with 8 bits per pixel, this process would take close to an hour and 15 minutes, which is not very practical. Even if we compressed these images before transmission, lossless compression on average gives us about a two-to-one 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 still is 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.*

Example 7.5.1:

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 1024×1024 image at 8 bits per pixel over a 56 kbps line takes about two and a half minutes. Using a block size of 8×8, and using the top-left pixel in each block as the representative value, means we approximate the 1024×1024 image with a 128×128 subsampled image. Using 8 bits per pixel and a 56 kbps line, the time required to transmit this approximation to the image takes less than two and a half seconds. 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 a minute and a half instead of the hour and a half 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 about seven seconds, so even if the user had to request a finer approximation every third image, this would only increase the total search time by a little more than a minute. 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-left 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.

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

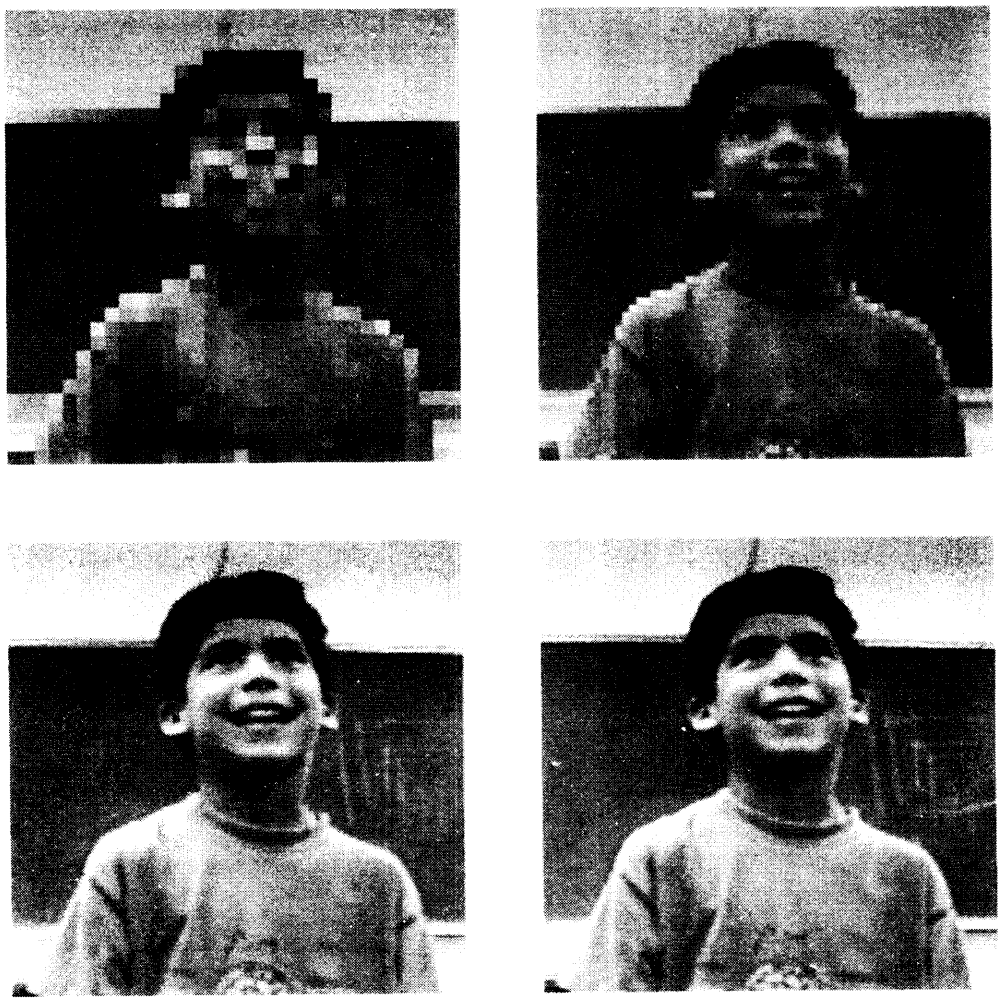


FIGURE 7. 3 **Sena image coded using different block sizes for progressive transmission. Top row: block size 8 x 8 and block size 4 x 4. Bottom row: block size 2 x 2 and original image.**

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.

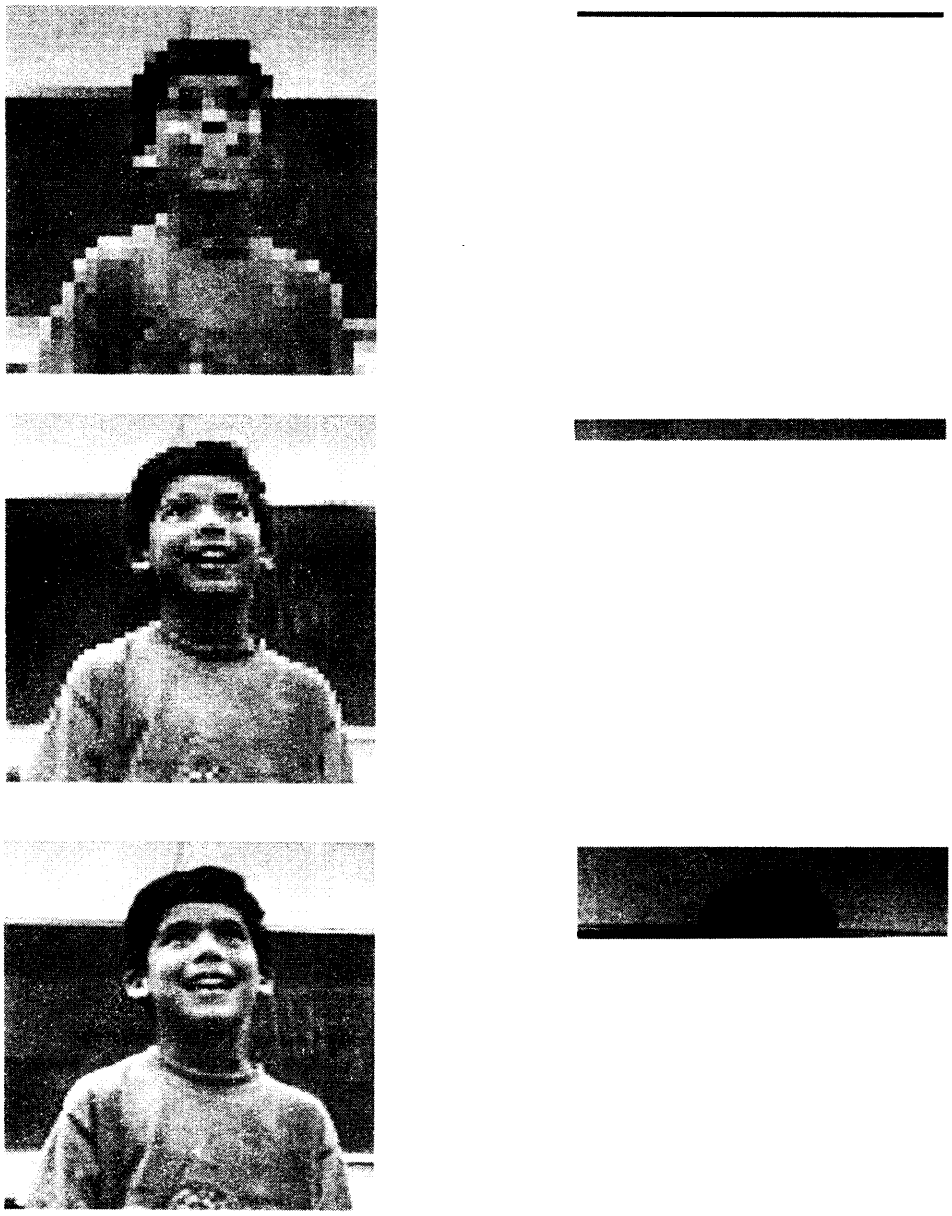


FIGURE 7. 4 **Comparison between the received image using progressive transmission and using the standard raster scan order.**

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 schemes for their initial approximations. Most of these schemes have a form similar to the one described in Example 7.5.1, 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 7.5.1 is that if the pixel values vary a lot within a block, the "representative" value may not be very representative. 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 [82, 83]. 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 7.5.1, 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

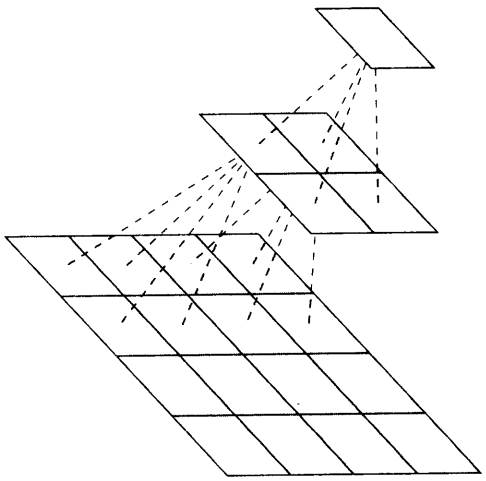


FIGURE 7.5 The pyramid structure for progressive transmission.

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.

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, see [84].)

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 can 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 [85].

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 [86]. 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.

7.6 Facsimile Encoding

One of the earliest applications of lossless compression in the modern era has been the compression 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 CCITT (now ITU-T) has 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 six 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 three 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 the ITU-T recommendations T.4, T.6, T.82 (JBIG) T.88 (JBIG2), and T.42 (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.

7.6.1 Run-Length Coding

The model that gives rise to run-length coding is the Capon model [87], a two-state Markov model with states S_w and S_b (S_w corresponds to the case where the pixel that has just been encoded is a white pixel, and S_b corresponds to the case where the pixel that has just been encoded is a black pixel). The transition probabilities $P(w|b)$ and $P(b|w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$, completely specify this model. For facsimile images, $P(w|w)$ and $P(w|b)$ are generally significantly higher than $P(b|w)$ and $P(b|b)$. The Markov model is represented by the state diagram shown in Figure 7.6.

The entropy of a finite state process with states S_i is given by Equation (2.16). Recall that in Example 2.3.1, the entropy using a probability model and the *iid* assumption was significantly more than the entropy using the Markov model.

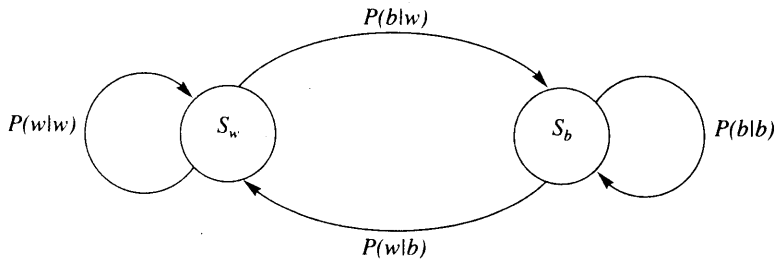


FIGURE 7.6 The Capon model for binary images.

Let us try to interpret what the model says about the structure of the data. The highly skewed nature of the probabilities $P(b|w)$ and $P(w|w)$, and to a lesser extent $P(w|b)$ and $P(b|b)$, says that once a pixel takes on a particular color (black or white), it is highly likely that the following pixels will also be of the same color. So, rather than code the color of each pixel separately, we can simply code the length of the runs of each color. For example, if we had 190 white pixels followed by 30 black pixels, followed by another 210 white pixels, instead of coding the 430 pixels individually, we would code the sequence 190, 30, 210, along with an indication of the color of the first string of pixels. Coding the lengths of runs instead of coding individual values is called run-length coding.

7.6.2 CCITT Group 3 and 4—Recommendations T.4 and T.6

The recommendations for Group 3 facsimile include two coding schemes. One is a one-dimensional scheme in which the coding on each line is performed independently of any other line. The other is two-dimensional; the coding of one line is performed using the line-to-line correlations.

The one-dimensional coding scheme is a run-length coding scheme in which each line is represented as a series of alternating white runs and black runs. The first run is always a white run. If the first pixel is a black pixel, then we assume that we have a white run of length zero.

Runs of different lengths occur with different probabilities; therefore, they are coded using a variable-length code. The approach taken in the CCITT standards T.4 and T.6 is to use a Huffman code to encode the run lengths. However, the number of possible lengths of runs is extremely large, and it is simply not feasible to build a codebook that large. Therefore, instead of generating a Huffman code for each run length r_l, the run length is expressed in the form

$$r_l = 64 \times m + t \qquad \text{for } t = 0, 1, \ldots, 63, \text{ and } m = 1, 2, \ldots, 27. \tag{7.10}$$

When we have to represent a run length r_l, instead of finding a code for r_l, we use the corresponding codes for m and t. The codes for t are called the *terminating codes*, and the codes for m are called the *make-up codes*. If $r_l < 63$, we only need to use a terminating code. Otherwise, both a make-up code and a terminating code are used. For the range of m and t given here, we can represent lengths of 1728, which is the number of pixels per line in an A4-size document. However, if the document is wider, the recommendations provide for those with an optional set of 13 codes. Except for the optional codes, there are separate codes for black and white run lengths. This coding scheme is generally referred to as a *modified Huffman (MH)* scheme.

In the two-dimensional scheme, instead of reporting the run lengths, which in terms of our Markov model is the length of time we remain in one state, we report the transition times when we move from one state to another state. Look at Figure 7.7. We can encode this in two ways. We can say that the first row consists of a sequence of runs 0, 2, 3, 3, 8, and the second row consists of runs of length 0, 1, 8, 3, 4 (notice the first runs of length zero). Or, we can encode the location of the pixel values that occur at a transition from white to

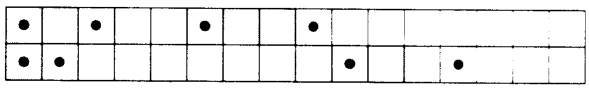


FIGURE 7.7 Two rows of an image. The transition pixels are marked with a dot.

black or black to white. The first pixel is an imaginary white pixel assumed to be to the left of the first actual pixel. Therefore, if we were to code transition locations, we would encode the first row as 1, 3, 6, 9 and the second row as 1, 2, 10, 13.

Generally, rows of a facsimile image are heavily correlated. Therefore, it would be easier to code the transition points with reference to the previous line than to code each one in terms of its absolute location, or even its distance from the previous transition point. This is the basic idea behind the recommended two-dimensional coding scheme. This scheme is a modification of a two-dimensional coding scheme called the *Relative Element Address Designate* (READ) code [88, 89] and is often referred to as *Modified READ* (MR). The READ code was the Japanese proposal to the CCITT for the Group 3 standard.

To understand the two-dimensional coding scheme, we need some definitions.

a_0: This is the last pixel whose value is known to both encoder and decoder. At the beginning of encoding each line, a_0 refers to an imaginary white pixel to the left of the first actual pixel. While it is often a transition pixel, it does not have to be.

a_1: This is the first transition pixel to the right of a_0. By definition its color should be the opposite of a_0. The location of this pixel is known only to the encoder.

a_2: This is the second transition pixel to the right of a_0. Its color should be the opposite of a_1, which means it has the same color as a_0. The location of this pixel is also known only to the encoder.

b_1: This is the first transition pixel on the line above the line currently being encoded to the right of a_0 whose color is the opposite of a_0. As the line above is known to both encoder and decoder, as is the value of a_0, the location of b_1 is also known to both encoder and decoder.

b_2: This is the first transition pixel to the right of b_1 in the line above the line currently being encoded.

For the pixels in Figure 7.7, if the second row is the one being currently encoded, and if we have encoded the pixels up to the second pixel, the assignment of the different pixels is shown in Figure 7.8. The pixel assignments for a slightly different arrangement of black and white pixels are shown in Figure 7.9.

If b_1 and b_2 lie between a_0 and a_1, we call the coding mode used the *pass mode*. The transmitter informs the receiver about the situation by sending the code 0001. Upon receipt of this code, the receiver knows that from the location of a_0 to the pixel right below b_2, all pixels are of the same color. If this had not been true, we would have encountered a transition pixel. As the first transition pixel to the right of a_0 is a_1, and as b_2 occurs before a_1, no transitions have occurred and all pixels from a_0 to right below b_2 are the same color. At this time, the last pixel known to both the transmitter and receiver is the pixel below b_2.

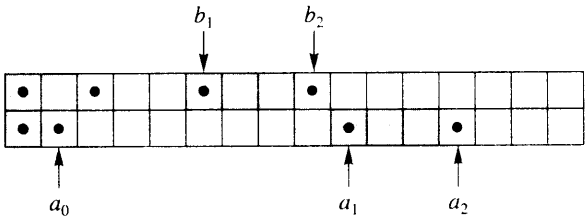


FIGURE 7.8 **Two rows of an image. The transition pixels are marked with a dot.**

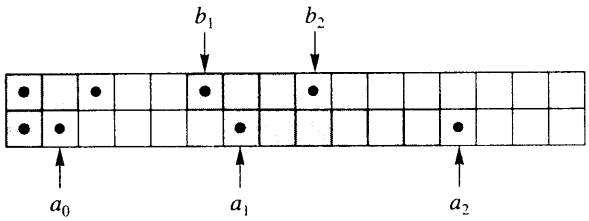


FIGURE 7.9 **Two rows of an image. The transition pixels are marked with a dot.**

Therefore, this now becomes the new a_0, and we find the new positions of b_1 and b_2 by examining the row above the one being encoded and continue with the encoding process.

If a_1 is detected before b_2 by the encoder, we do one of two things. If the distance between a_1 and b_1 (the number of pixels from a_1 to right under b_1) is less than or equal to three, then we send the location of a_1 with respect to b_1, move a_0 to a_1, and continue with the coding process. This coding mode is called the *vertical mode*. If the distance between a_1 and b_1 is large, we essentially revert to the one-dimensional technique and send the distances between a_0 and a_1, and a_1 and a_2, using the modified Huffman code. Let us look at exactly how this is accomplished.

In the vertical mode, if the distance between a_1 and b_1 is zero (that is, a_1 is exactly under b_1), we send the code 1. If the a_1 is to the right of b_1 by one pixel (as in Figure 7.9), we send the code 011. If a_1 is to the right of b_1 by two or three pixels, we send the codes 000011 or 0000011, respectively. If a_1 is to the left of b_1 by one, two, or three pixels, we send the codes 010, 000010, or 0000010, respectively.

In the horizontal mode, we first send the code 001 to inform the receiver about the mode, and then send the modified Huffman codewords corresponding to the run length from a_0 to a_1, and a_1 to a_2.

As the encoding of a line in the two-dimensional algorithm is based on the previous line, an error in one line could conceivably propagate to all other lines in the transmission. To prevent this from happening, the T.4 recommendations contain the requirement that after each line is coded with the one-dimensional algorithm, at most $K - 1$ lines will be coded using the two-dimensional algorithm. For standard vertical resolution, $K = 2$, and for high resolution, $K = 4$.

The Group 4 encoding algorithm, as standardized in CCITT recommendation T.6, is identical to the two-dimensional encoding algorithm in recommendation T.4. The main difference between T.6 and T.4 from the compression point of view is that T.6 does not have a one-dimensional coding algorithm, which means that the restriction described in the previous paragraph is also not present. This slight modification of the modified READ algorithm has earned the name *modified modified READ* (MMR)!

7.6.3 JBIG

Many bi-level images have a lot of local structure. Consider a digitized page of text. In large portions of the image we will encounter white pixels with a probability approaching 1. In other parts of the image there will be a high probability of encountering a black pixel. We can make a reasonable guess of the situation for a particular pixel by looking at values of the pixels in the neighborhood of the pixel being encoded. For example, if the pixels in the neighborhood of the pixel being encoded are mostly white, then there is a high probability that the pixel to be encoded is also white. On the other hand, if most of the pixels in the neighborhood are black, there is a high probability that the pixel being encoded is also black. Each case gives us a skewed probability—a situation ideally suited for arithmetic coding. If we treat each case separately, using a different arithmetic coder for each of the two situations, we should be able to obtain improvement over the case where we use the same arithmetic coder for all pixels. Consider the following example.

Suppose the probability of encountering a black pixel is 0.2 and the probability of encountering a white pixel is 0.8. The entropy for this source is given by

$$H = -0.2\log_2 0.2 - 0.8\log_2 0.8 = 0.722. \tag{7.11}$$

If we use a single arithmetic coder to encode this source, we will get an average bit rate close to 0.722 bits per pixel. Now suppose, based on the neighborhood of the pixels, that we can divide the pixels into two sets, one comprising 80% of the pixels and the other 20%. In the first set, the probability of encountering a white pixel is 0.95, and in the second set the probability of encountering a black pixel is 0.7. The entropy of these sets is 0.286 and 0.881, respectively. If we used two different arithmetic coders for the two sets with frequency tables matched to the probabilities, we would get rates close to 0.286 bits per pixel about 80% of the time and close to 0.881 bits per pixel about 20% of the time. The average rate would be about 0.405 bits per pixel, which is almost half the rate required if we used a single arithmetic coder. If we use only those pixels in the neighborhood that had already been transmitted to the receiver to make our decision about which arithmetic coder to use, the decoder can keep track of which encoder was used to encode a particular pixel.

As we have mentioned before, the arithmetic coding approach is particularly amenable to the use of multiple coders. All coders use the same computational machinery, with each coder using a different set of probabilities. The JBIG algorithm makes full use of this feature of arithmetic coding. Instead of checking to see if most of the pixels in the neighborhood are white or black, the JBIG encoder uses the pattern of pixels in the neighborhood, or *context*, to decide which set of probabilities to use in encoding a particular pixel. If the neighborhood consists of 10 pixels, with each pixel capable of taking on two different values, the number of

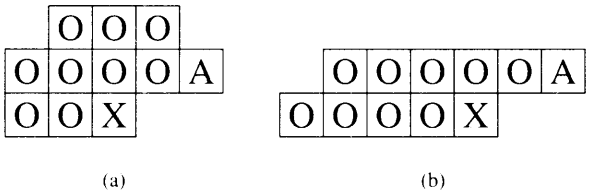


FIGURE 7. 10 (a) Three-line and (b) two-line neighborhoods.

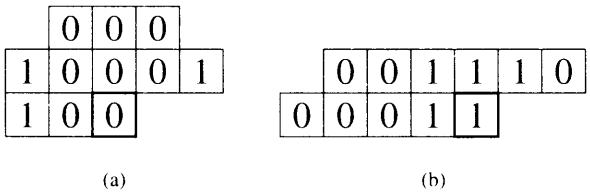


FIGURE 7. 11 (a) Three-line and (b) two-line contexts.

possible patterns is 1024. The JBIG coder uses 1024 to 4096 coders, depending on whether a low- or high-resolution layer is being encoded.

For the low-resolution layer, the JBIG encoder uses one of the two different neighborhoods shown in Figure 7.10. The pixel to be coded is marked **X**, while the pixels to be used for templates are marked **O** or **A**. The **A** and **O** pixels are previously encoded pixels and are available to both encoder and decoder. The **A** pixel can be thought of as a floating member of the neighborhood. Its placement is dependent on the input being encoded. Suppose the image has vertical lines 30 pixels apart. The **A** pixel would be placed 30 pixels to the left of the pixel being encoded. The **A** pixel can be moved around to capture any structure that might exist in the image. This is especially useful in halftone images in which the **A** pixels are used to capture the periodic structure. The location and movement of the **A** pixel are transmitted to the decoder as side information.

In Figure 7.11, the symbols in the neighborhoods have been replaced by 0s and 1s. We take 0 to correspond to white pixels, while 1 corresponds to black pixels. The pixel to be encoded is enclosed by the heavy box. The pattern of 0s and 1s is interpreted as a binary number, which is used as an index to the set of probabilities. The context in the case of the three-line neighborhood (reading left to right, top to bottom) is 0001000110, which corresponds to an index of 70. For the two-line neighborhood, the context is 0011100001, or 225. Since there are 10 bits in these templates, we will have 1024 different arithmetic coders.

In the JBIG standard, the 1024 arithmetic coders are a variation of the arithmetic coder known as the QM coder. The QM coder is a modification of an adaptive binary arithmetic coder called the Q coder [51, 52, 53], which in turn is an extension of another binary adaptive arithmetic coder called the skew coder [90].

In our description of arithmetic coding, we updated the tag interval by updating the endpoints of the interval, $u^{(n)}$ and $l^{(n)}$. We could just as well have kept track of one endpoint

and the size of the interval. This is the approach adopted in the QM coder, which tracks the lower end of the tag interval $l^{(n)}$ and the size of the interval $A^{(n)}$, where

$$A^{(n)} = u^{(n)} - l^{(n)}. \tag{7.12}$$

The tag for a sequence is the binary representation of $l^{(n)}$.

We can obtain the update equation for $A^{(n)}$ by subtracting Equation (4.9) from Equation (4.10) and making this substitution

$$A^{(n)} = A^{(n-1)}(F_X(x_n) - F_X(x_n - 1)) \tag{7.13}$$

$$= A^{(n-1)}P(x_n). \tag{7.14}$$

Substituting $A^{(n)}$ for $u^{(n)} - l^{(n)}$ in Equation (4.9), we get the update equation for $l^{(n)}$:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}F_X(x_n - 1). \tag{7.15}$$

Instead of dealing directly with the 0s and 1s put out by the source, the QM coder maps them into a More Probable Symbol (MPS) and Less Probable Symbol (LPS). If 0 represents black pixels and 1 represents white pixels, then in a mostly black image 0 will be the MPS, whereas in an image with mostly white regions 1 will be the MPS. Denoting the probability of occurrence of the LPS for the context C by q_c and mapping the MPS to the lower subinterval, the occurrence of an MPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} \tag{7.16}$$

$$A^{(n)} = A^{(n-1)}(1 - q_c) \tag{7.17}$$

while the occurrence of an LPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c) \tag{7.18}$$

$$A^{(n)} = A^{(n-1)}q_c. \tag{7.19}$$

Until this point, the QM coder looks very much like the arithmetic coder described earlier in this chapter. To make the implementation simpler, the JBIG committee recommended several deviations from the standard arithmetic coding algorithm. The update equations involve multiplications, which are expensive in both hardware and software. In the QM coder, the multiplications are avoided by assuming that $A^{(n)}$ has a value close to 1, and multiplication with $A^{(n)}$ can be approximated by multiplication with 1. Therefore, the update equations become

For MPS:

$$l^{(n)} = l^{(n-1)} \tag{7.20}$$

$$A^{(n)} = 1 - q_c \tag{7.21}$$

For LPS:

$$l^{(n)} = l^{(n-1)} + (1 - q_c) \tag{7.22}$$

$$A^{(n)} = q_c \tag{7.23}$$

In order not to violate the assumption on $A^{(n)}$ whenever the value of $A^{(n)}$ drops below 0.75, the QM coder goes through a series of rescalings until the value of $A^{(n)}$ is greater than or equal to 0.75. The rescalings take the form of repeated doubling, which corresponds to a left shift in the binary representation of $A^{(n)}$. To keep all parameters in sync, the same scaling is also applied to $l^{(n)}$. The bits shifted out of the buffer containing the value of $l^{(n)}$ make up the encoder output. Looking at the update equations for the QM coder, we can see that a rescaling will occur every time an LPS occurs. Occurrence of an MPS may or may not result in a rescale, depending on the value of $A^{(n)}$.

The probability q_c of the LPS for context C is updated each time a rescaling takes place and the context C is active. An ordered list of values for q_c is listed in a table. Every time a rescaling occurs, the value of q_c is changed to the next lower or next higher value in the table, depending on whether the rescaling was caused by the occurrence of an LPS or an MPS.

In a nonstationary situation, the symbol assigned to LPS may actually occurs more often than the symbol assigned to MPS. This condition is detected when $q_c > (A^{(n)} - q_c)$. In this situation, the assignments are reversed; the symbol assigned the LPS label is assigned the MPS label and vice versa. The test is conducted every time a rescaling takes place.

The decoder for the QM coder operates in much the same way as the decoder described in this chapter, mimicking the encoder operation.

Progressive Transmission

In some applications we may not always need to view an image at full resolution. For example, if we are looking at the layout of a page, we may not need to know what each word or letter on the page is. The JBIG standard allows for the generation of progressively lower-resolution images. If the user is interested in some gross patterns in the image (for example, if they were interested in seeing if there were any figures on a particular page) they could request a lower-resolution image, which could be transmitted using fewer bits. Once the lower-resolution image was available, the user could decide whether a higher-resolution image was necessary. The JBIG specification recommends generating one lower-resolution pixel for each 2×2 block in the higher-resolution image. The number of lower-resolution images (called layers) is not specified by JBIG.

A straightforward method for generating lower-resolution images is to replace every 2×2 block of pixels with the average value of the four pixels, thus reducing the resolution by two in both the horizontal and vertical directions. This approach works well as long as three of the four pixels are either black or white. However, when we have two pixels of each kind, we run into trouble; consistently replacing the four pixels with either a white or black pixel causes a severe loss of detail, and randomly replacing with a black or white pixel introduces a considerable amount of noise into the image [81].

Instead of simply taking the average of every 2×2 block, the JBIG specification provides a table-based method for resolution reduction. The table is indexed by the neighboring pixels shown in Figure 7.12, in which the circles represent the lower-resolution layer pixels and the squares represent the higher-resolution layer pixels.

Each pixel contributes a bit to the index. The table is formed by computing the expression

$$4e + 2(b + d + f + h) + (a + c + g + i) - 3(B + C) - A.$$