4

Arithmetic Coding

4.1 Overview

In the previous chapter, we saw one approach to generating variable-length codes. In this chapter, we see another, increasingly popular, method of generating variable-length codes called *arithmetic coding*. Arithmetic coding is especially useful when dealing with sources with small alphabets, such as binary sources, and alphabets with highly skewed probabilities. It is also a very useful approach when, for various reasons, the modeling and coding aspects of lossless compression are to be kept separate. In this chapter, we look at the basic ideas behind arithmetic coding, study some of the properties of arithmetic codes, and describe an implementation.

4.2 Introduction

In the last chapter, we studied the Huffman coding method, which guarantees a coding rate R within 1 bit of the entropy H. Recall that the coding rate is the average number of bits used to represent a symbol from a source, and, for a given probability model, the entropy is the lowest rate at which the source can be coded. We can tighten this bound somewhat. It has been shown [21] that the Huffman algorithm will generate a code whose rate is within $p_{\max} + 0.086$ of the entropy, where $p_{\max}$ is the probability of the most frequently occurring symbol. We noted in the last chapter that, in applications where the alphabet size is large, $p_{\max}$ is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small. However, in cases where the alphabet is small and the probability

Introduction to Data Compression. DOI: http://dx.doi.org/10.1016/B978-0-12-415796-5.00004-1

of occurrence of the different letters is skewed, the value of $p_{\max}$ can be quite large, and the Huffman code can become rather inefficient when compared to the entropy. One way to avoid this problem is to block more than one symbol together and generate an extended Huffman code. Unfortunately, this approach does not always work.

Example 4.2.1:

Consider a source that puts out independent, identically distributed (*iid*) letters from the alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.95$, $P(a_2) = 0.02$, and $P(a_3) = 0.03$. The entropy for this source is 0.335 bits/symbol. A Huffman code for this source is given in Table 4.1.

TABLE 4.1 Huffman code for three-letter alphabet.

Letter	Codeword
a_1	0
a_2	11
a_3	10

The average length for this code is 1.05 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.715 bits/symbol, which is 213% of the entropy. This means that to code this sequence, we would need more than twice the number of bits promised by the entropy.

Recall Example 3.2.4. Here also we can group the symbols in blocks of two. The extended alphabet, probability model, and code can be obtained as shown in Table 4.2. The average rate for the extended alphabet is 1.222 bits/symbol, which in terms of the original alphabet is 0.611 bits/symbol. As the entropy of the source is 0.335 bits/symbol, the additional rate over the entropy is still about 82% of the entropy! By continuing to block symbols together, we find that the redundancy drops to acceptable values when we block eight symbols together. The corresponding alphabet size for this level of blocking is 6561! A code of this size is impractical for a number of reasons. Storage of a code like this requires memory that may not be available for many applications. While it may be possible to design reasonably efficient encoders, decoding a Huffman code of this size would be a highly inefficient and time-consuming procedure. Finally, if there were some perturbation in the statistics, and some of the assumed probabilities changed slightly, this would have a major impact on the efficiency of the code. ♦

We can see that it is more efficient to generate codewords for groups or sequences of symbols rather than to generate a separate codeword for each symbol in a sequence. However, this approach becomes impractical when we try to obtain Huffman codes for long sequences of symbols. In order to find the Huffman codeword for a particular sequence of length m, we need codewords for all possible sequences of length m. This fact causes an exponential growth in the size of the codebook. We need a way of assigning codewords to *particular*

TABLE 4.2 Huffman code for extended alphabet.

Letter	Probability	Code
a_1a_1	0.9025	0
a_1a_2	0.0190	111
a_1a_3	0.0285	100
a_2a_1	0.0190	1101
a_2a_2	0.0004	110011
a_2a_3	0.0006	110001
a_3a_1	0.0285	101
a_3a_2	0.0006	110010
a_3a_3	0.0009	110000

sequences without having to generate codes for all sequences of that length. The arithmetic coding technique fulfills this requirement.

In arithmetic coding, a unique identifier or tag is generated for the sequence to be encoded. This tag corresponds to a binary fraction, which becomes the binary code for the sequence. In practice, the generation of the tag and the binary code are the same process. However, the arithmetic coding approach is easier to understand if we conceptually divide the approach into two phases. In the first phase, a unique identifier or tag is generated for a given sequence of symbols. This tag is then given a unique binary code. A unique arithmetic code can be generated for a sequence of length *m* without the need for generating codewords for all sequences of length *m*. This is unlike the situation for Huffman codes. In order to generate a Huffman code for a sequence of length *m*, where the code is not a concatenation of the codewords for the individual symbols, we need to obtain the Huffman codes for all sequences of length *m*.

4.3 Coding a Sequence

In order to distinguish a sequence of symbols from another sequence of symbols we need to tag it with a unique identifier. One possible set of tags for representing sequences of symbols are the numbers in the unit interval [0,1). Because the number of numbers in the unit interval is infinite, it should be possible to assign a unique tag to each distinct sequence of symbols. In order to do this, we need a function that will map sequences of symbols into the unit interval. A function that maps random variables, and sequences of random variables, into the unit interval is the cumulative distribution function (*cdf*) of the random variable associated with the source. This is the function we will use in developing the arithmetic code. (If you are not familiar with random variables and cumulative distribution functions or need to refresh your memory, you may wish to look at Appendix A.)

The use of the cumulative distribution function to generate a binary code for a sequence has a rather interesting history. Shannon, in his original 1948 paper [3], mentioned an approach using the cumulative distribution function when describing what is now known as the Shannon-Fano code. Peter Elias, another member of Fano's first information theory class at MIT (this class also included Huffman), came up with a recursive implementation for this idea. However, he never published it, and we only know about it through a mention in a 1963 book on information theory by Abramson [39]. Abramson described this coding approach in a note to a chapter. In another book on information theory by Jelinek [40] in 1968, the idea of arithmetic coding is further developed, this time in an appendix, as an example of variable-length coding. Modern arithmetic coding owes its birth to the independent discoveries in 1976 of Pasco [41] and Rissanen [42] who solved the problem of finite precision. Finally, several papers appeared that provided practical arithmetic coding algorithms, the most well known of which is the paper by Rissanen and Langdon [43].

Before we begin our development of the arithmetic code, we need to establish some notation. Recall that a random variable maps the outcomes, or sets of outcomes, of an experiment to values on the real number line. For example, in a coin-tossing experiment, the random variable could map a head to zero and a tail to one (or it could map a head to 2367.5 and a tail to -192). To use this technique, we need to map the source symbols or letters to numbers. For convenience, in the discussion in this chapter we will use the mapping

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

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

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

and the cumulative density function can be defined as

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

Notice that for each symbol a_i with a nonzero probability, we have a distinct value of $F_X(i)$. We will use this fact in what follows to develop the arithmetic code. Our development may be more detailed than what you are looking for, at least on the first reading. If so, skip or skim Sections 4.3.1–4.4.1 and go directly to Section 4.4.2.

4.3.1 Generating a Tag

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

We start out by first dividing the unit interval into subintervals of the form $[F_X(i-1), F_X(i))$, $i = 1, \ldots, m$. Because the minimum value of the *cdf* is zero and the maximum value is one, this exactly partitions the unit interval. We associate the subinterval $[F_X(i-1), F_X(i))$

with the symbol a_i. The appearance of the first symbol in the sequence restricts the interval containing the tag to one of these subintervals. Suppose the first symbol was a_k. Then the interval containing the tag value will be the subinterval $[F_X(k-1), F_X(k))$. This subinterval is now partitioned in exactly the same proportions as the original interval. That is, the jth interval corresponding to the symbol a_j is given by $[F_X(k-1) + F_X(j-1) \times (F_X(k) - F_X(k-1), F_X(k-1) + F_X(j) \times (F_X(k) - F_X(k-1))]$. So if the second symbol in the sequence is a_j, then the interval containing the tag value becomes $[F_X(k-1) + F_X(j-1) \times (F_X(k) - F_X(k-1), F_X(k-1) + F_X(j) \times (F_X(k) - F_X(k-1)))]$. Each succeeding symbol causes the tag to be restricted to a subinterval that is further partitioned in the same proportions. This process can be more clearly understood through an example.

Example 4.3.1:

Consider a three-letter alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with $P(a_1) = 0.7$, $P(a_2) = 0.1$, and $P(a_3) = 0.2$. Using the mapping of Equation (1), $F_X(1) = 0.7$, $F_X(2) = 0.8$, and $F_X(3) = 1$. This partitions the unit interval as shown in Figure 4.1.

The partition in which the tag resides depends on the first symbol of the sequence being encoded. For example, if the first symbol is a_1, the tag lies in the interval [0.0, 0.7); if the first symbol is a_2, the tag lies in the interval [0.7, 0.8); and if the first symbol is a_3, the tag lies in the interval [0.8, 1.0). Once the interval containing the tag has been determined, the rest of the unit interval is discarded, and this restricted interval is again divided in the same proportions as the original interval. Suppose the first symbol was a_1. The tag would be contained in the subinterval [0.0, 0.7). This subinterval would then be subdivided into exactly the same proportions as the original interval, yielding the subintervals [0.0, 0.49), [0.49, 0.56), and [0.56, 0.7). The first partition, as before, would correspond to the symbol a_1, the second partition would correspond to the symbol a_2, and the third partition [0.56, 0.7) would correspond to

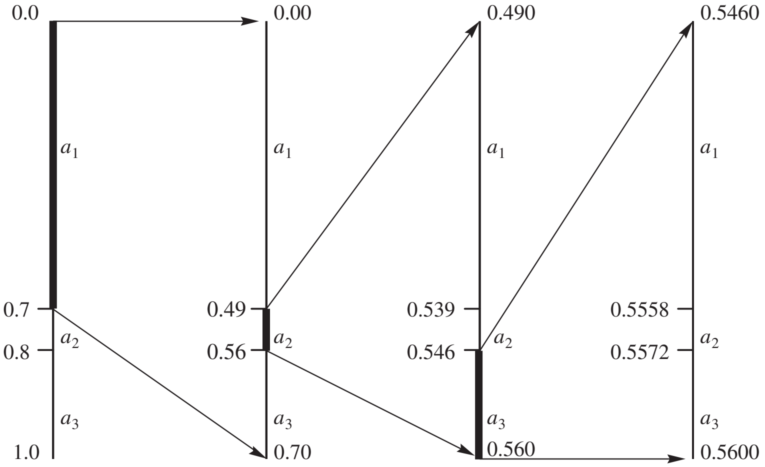

FIGURE 4.1 **Restricting the interval containing the tag for the input sequence $\{a_1, a_2, a_3, \ldots\}$.**

the symbol a_3. Suppose the second symbol in the sequence was a_2. The tag value would then be restricted to lie in the interval [0.49, 0.56). We would then partition this interval in the same proportion as the original interval to obtain the subintervals [0.49, 0.539) corresponding to the symbol a_1, [0.539, 0.546) corresponding to the symbol a_2, and [0.546, 0.56) corresponding to the symbol a_3. If the third symbol was a_3, the tag would be restricted to the interval [0.546, 0.56), which could then be subdivided further. This process is described graphically in Figure 4.1.

Notice that the appearance of each new symbol restricts the tag to a subinterval that is disjoint from any other subinterval that may have been generated using this process. For the sequence beginning with $a_1, a_2, a_3, \ldots$, by the time the third symbol a_3 is received, the tag has been restricted to the subinterval [0.546, 0.56). If the third symbol had been a_1 instead of a_3, the tag would have resided in the subinterval [0.49, 0.539), which is disjoint from the subinterval [0.546, 0.56). Even if the two sequences are identical from this point on (one starting with a_1, a_2, a_3 and the other beginning with a_1, a_2, a_1), the tag interval for the two sequences will always be disjoint. ♦

As we can see, the interval in which the tag for a particular sequence resides is disjoint from all intervals in which the tag for any other sequence may reside. As such, any member of this interval can be used as a tag. One popular choice is the lower limit of the interval; another possibility is the midpoint of the interval. For the moment, let's use the midpoint of the interval as the tag.

In order to see how the tag generation procedure works mathematically, we start with sequences of length one. Suppose we have a source that puts out symbols from some alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$. We can map the symbols $\{a_i\}$ to real numbers $\{i\}$. Define $\overline{T}_X(a_i)$ as

$$\overline{T}_X(a_i) = \sum_{k=1}^{i-1} P(X=k) + \frac{1}{2}P(X=i) \tag{2}$$

$$= F_X(i-1) + \frac{1}{2}P(X=i) \tag{3}$$

For each a_i, $\overline{T}_X(a_i)$ will have a unique value. This value can be used as a unique tag for a_i.

Example 4.3.2:

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

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

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

$$\overline{T}_X(2) = P(X=1) + \frac{1}{2}P(X=2) = \frac{1}{6} + \frac{1}{12} = 0.25$$

and the tag for $X = 5$ as

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

The tags for all other outcomes are shown in Table4.3 where the overbar denotes repeating decimal values.

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

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

♦

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

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

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 4.3.2 so that the sequence consists of two rolls of a die. Using the ordering scheme described above, the outcomes (in order) would be 11, 12, 13, ..., 66. The tags can then be generated using Equation (4). For example, the tag for the sequence 1 3 would be

$$\overline{T}_X(13) = P(\mathbf{x} = 11) + P(\mathbf{x} = 12) + 1/2P(\mathbf{x} = 13) \tag{5}$$

$$= 1/36 + 1/36 + 1/2(1/36) \tag{6}$$

$$= 5/72 \tag{7}$$

♦

Notice that to generate the tag for the sequence 1 3 we did not have to generate a tag for every other possible message. However, based on Equation (4) and Example 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 3 2 2. 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:

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

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 $\mathbf{x} = x_1x_2$. 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

$$\begin{aligned} F_X^{(3)}(322) &= F_X^{(2)}(31) + (F_X^{(2)}(32) - F_X^{(2)}(31))F_X(2) \\ F_X^{(3)}(321) &= F_X^{(2)}(31) + (F_X^{(2)}(32) - F_X^{(2)}(31))F_X(1) \end{aligned} \tag{8}$$

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 $\mathbf{x} = (x_1 x_2 \ldots x_n)$

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_n - 1) \tag{9}$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_n) \tag{10}$$

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

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

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

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

We can use Equations (9) and (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:

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

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

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

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

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

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

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

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

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

♦

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

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

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 $F_X(x_k - 1) \leqslant t^* < F_X(x_k)$.
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 **x**. However, the *binary code* for the sequence is what we really want to know. We want to find a binary code that will represent the sequence **x** in a unique and efficient manner.

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

$\overline{T}_X(x)$ is a number in the interval [0, 1). A binary code for $\overline{T}_X(x)$ 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. Recall that the binary representations of decimal numbers in the interval [0, 1) are obtained as the negative powers of two. The decimal equivalent of the binary number $.b_1 b_2 b_3 \ldots b_k$ is $b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \cdots + b_k 2^{-k}$. Thus $.101 = 2^{-1} + 2^{-3}$.

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 Table4.4. The quantity $\overline{T}_x$ is obtained using Equation (3). The binary representation of $\overline{T}_x$ 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	$\overline{T}_X$	In Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	Code
1	.500	.2500	.0100	2	01
2	.750	.6250	.1010	3	101
3	.875	.8125	.1101	4	1101
4	1.000	.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. We will use $\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ to denote the truncation of $\overline{T}_X(x)$ to l($\mathbf{x}$) bits.

Recall that while we have been using $\overline{T}_X(x)$ 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 $\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ 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 $\overline{T}_X(\mathbf{x})$ to obtain $\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$, $\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is less than or equal to $\overline{T}_X(\mathbf{x})$. More specifically,

$$0 \leqslant \overline{T}_X(\mathbf{x}) - \lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} < \frac{1}{2^{l(\mathbf{x})}} \tag{11}$$

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

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

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

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

From (3) we have

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

Therefore,

$$\overline{T}_X(\mathbf{x}) - F_X(\mathbf{x}-1) > \frac{1}{2^{l(\mathbf{x})}} \tag{12}$$

Combining (11) and (12), we have

$$\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1) \tag{13}$$

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

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, this is enough to show that an arithmetic code 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 at the end of this chapter.)

If $\mathbf{x}$ and $\mathbf{y}$ are two distinct sequences, we know that $\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ and $\lfloor \overline{T}_X(\mathbf{y}) \rfloor_{l(\mathbf{y})}$ 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 $[\lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}, \lfloor \overline{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} + \frac{1}{2^{l(\mathbf{x})}})$ lies entirely within the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$, the code for one sequence cannot be the prefix for the code for another sequence.

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

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

This is true because

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

This code is prefix free; and by taking the binary representation of $\overline{T}_X(\mathbf{x})$ 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 $\mathbf{x}$ is 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

$$\begin{aligned} l_{A^{(m)}} &= \sum P(\mathbf{x}) l(\mathbf{x}) && (14) \\ &= \sum P(\mathbf{x}) \left[\left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1 \right] && (15) \\ &< \sum P(\mathbf{x}) \left[\log \frac{1}{P(\mathbf{x})} + 1 + 1 \right] && (16) \\ &= -\sum P(\mathbf{x}) \log P(\mathbf{x}) + 2 \sum P(\mathbf{x}) && (17) \\ &= H(X^m) + 2 && (18) \end{aligned}$$

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

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

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

$$\frac{H(X^{(m)})}{m} \leqslant l_A < \frac{H(X^{(m)})}{m} + \frac{2}{m} \qquad (19)$$

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

$$H(X^{(m)}) = mH(X) \qquad (20)$$

Therefore,

$$H(X) \leqslant l_A < H(X) + \frac{2}{m} \qquad (21)$$

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

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1) \tag{22}$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n) \tag{23}$$

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 of 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. Thus, 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

$$E_1 : [0, 0.5) \rightarrow [0, 1); \qquad E_1(x) = 2x \tag{24}$$

$$E_2 : [0.5, 1) \rightarrow [0, 1); \qquad E_2(x) = 2(x - 0.5) \tag{25}$$

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 **1321**. 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:

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

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

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

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

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

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

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

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

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:

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

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

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

The interval [0.656, 0.8) is contained entirely in the upper half of the unit interval. 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

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

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 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:

$$E_3 : [0.25, 0.75) \rightarrow [0, 1); \qquad E_3(x) = 2(x - 0.25) \tag{26}$$

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

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

and 1 gets mapped to

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

The value of 0.5 gets mapped to

$$1\overbrace{00\ldots0}^{m-1\text{times}}$$

The update equations remain almost the same as Equations (9) and (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

$$F_X(k) = \frac{\sum_{i=1}^{k} n_i}{Total\_Count}. \tag{27}$$

If we now define

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

we can write Equations (9) and (10) as

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n - 1)}{Total\_Count} \right\rfloor \tag{28}$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n)}{Total\_Count} \right\rfloor - 1 \tag{29}$$

where x_n is the nth symbol to be encoded, $\lfloor x \rfloor$ is the largest integer less than or equal to x, and the addition and subtraction of one is to handle the effects of the integer arithmetic. Note that in order to reduce effects of finite precision arithmetic it is a good idea to perform the multiplication in the numerator prior to the division.

The word length m has to be large enough to accommodate all the upper and lower limits of all the subintervals, which means that there should be enough values to unambiguously represent each entry of the *Cum_Count* array. As the maximum number of distinct values is *Total_Count* this means that the number of values that need to be represented is *Total_Count* and, therefore, we need to pick m such that

$$2^m > Total\_Count$$

or

$$m > \lceil \log_2(Total\_Count) \rceil$$

However, this may not be sufficient as often the active interval, that is the interval $[l^{(n)}, u^{(n)}]$ is only a portion of the total range available. As all the subintervals need to be contained within the active interval at any given time, what we need to do is determine the smallest size the active interval can be and then make sure that m is large enough to contain *Total_Count* different values in this limited range. So what is the smallest the active interval can be? At first sight it might seem that the smallest the active interval can be is about half the maximum range because as soon as the upper limit slips below the halfway mark or the lower limit slips above the halfway mark we double the interval. However, on closer examination, we can see that this is not the case. When the upper limit is barely above the halfway mark, the lower limit is not required to be at 0. In fact, the lower limit can be just below the quarter range without any rescaling being triggered. However, the moment the lower limit goes above the quarter mark an E_3 rescaling is triggered and the interval is redoubled one or more times. Thus the smallest the active range can be is a quarter of the total range. This means that we need to accommodate *Total_Count* values in a quarter of the total range available, that is

$$\frac{1}{4}2^m > Total\_Count$$

or

$$\begin{aligned} m &> \lceil \log_2(4 \times Total\_Count) \rceil \\ &= 2 + \lceil \log_2(Total\_Count) \rceil \end{aligned}$$

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 $[00\ \ldots 0, 11\ \ldots 1]$ 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 a 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 $[00\ \ldots 0, 11\ \ldots 1]$ 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.
$l \leftarrow l + \left\lfloor \frac{(u-l+1) \times Cum\_Count(x-1)}{Total\_Count} \right\rfloor$
$u \leftarrow l + \left\lfloor \frac{(u-l+1) \times Cum\_Count(x)}{Total\_Count} \right\rfloor - 1$
```

```
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 $b$)
  {
    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 the parameters shown in Table4.5. 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 uniquely accommodate the set of values between 0 and *Total_Count*.

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

$$l^{(0)} = 0 = (00000000)_2 \tag{30}$$

$$u^{(0)} = 255 = (11111111)_2 \tag{31}$$

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

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

$Count(1) = 40$	$Cum\_Count(0) = 0$	Scale3 = 0
$Count(2) = 1$	$Cum\_Count(1) = 40$	
$Count(3) = 9$	$Cum\_Count(2) = 41$	
$Total\_Count = 50$	$Cum\_Count(3) = 50$	

The first element of the sequence to be encoded is **1**. Using Equations (9) and (10)

$$l^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{32}$$

$$u^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 203 = (11001011)_2 \tag{33}$$

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

$$l^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(2)}{50} \right\rfloor = 167 = (10100111)_2 \tag{34}$$

$$u^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(3)}{50} \right\rfloor - 1 = 203 = (11001011)_2 \tag{35}$$

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

$$l^{(2)} = (01001110)_2 = 78 \tag{36}$$

$$u^{(2)} = (10010111)_2 = 151 \tag{37}$$

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 least significant bit (LSB) of $l^{(2)}$ and a 1 as the LSB of $u^{(2)}$. This gives us

$$l^{(2)} = (00011100)_2 = 28 \tag{38}$$

$$u^{(2)} = (10101111)_2 = 175 \tag{39}$$

We also increment Scale3 to a value of 1.

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

$$l^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(1)}{50} \right\rfloor = 146 = (10010010)_2 \tag{40}$$

$$u^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(2)}{50} \right\rfloor - 1 = 148 = (10010100)_2 \tag{41}$$

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

$$l^{(3)} = (00100100)_2 = 36 \tag{42}$$

$$u^{(3)} = (00101001)_2 = 41 \tag{43}$$

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:

$$l^{(3)} = (01001000)_2 = 72 \tag{44}$$
$$u^{(3)} = (01010011)_2 = 83 \tag{45}$$

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

$$l^{(3)} = (10010000)_2 = 144 \tag{46}$$
$$u^{(3)} = (10100111)_2 = 167 \tag{47}$$

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

$$l^{(3)} = (00100000)_2 = 32 \tag{48}$$
$$u^{(3)} = (01001111)_2 = 79 \tag{49}$$

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

$$l^{(3)} = (01000000)_2 = 64 \tag{50}$$
$$u^{(3)} = (10011111)_2 = 159 \tag{51}$$

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

$$l^{(3)} = (00000000)_2 = 0 \tag{52}$$
$$u^{(3)} = (10111111)_2 = 191 \tag{53}$$

We also increment Scale3 to 1.

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

$$l^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{54}$$
$$u^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 152 = (10011000)_2 \tag{55}$$

The encoding continues in this fashion. To this point we have generated the binary sequence 1100010. If we wish 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 $\left( \left\lfloor \frac{(t - l + 1) \times Total\_Count - 1}{u - l + 1} \right\rfloor \geqslant Cum\_Count(k) \right)$
$k \leftarrow k + 1$
Decode symbol $x$.
$l \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x - 1)}{Total\_Count} \right\rfloor$
$u \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x)}{Total\_Count} \right\rfloor - 1$
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, 8, 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

$$0 \leqslant 38 < 40$$

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 MSBs 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:

$$Cum\_Count[2] \leqslant 48 < Cum\_Count[3]$$

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

$$\begin{aligned} l &= (00011100)_2 = 28 \\ u &= (10101111)_2 = 175 \\ t &= (10010010)_2 = 146 \end{aligned}$$

To decode the next symbol, we compute

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

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

Updating the limits using this decoded symbol, we get

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

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 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 to have 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 accommodate 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 round 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 statistics of the source.

4.6 Binary Arithmetic Coding

In many applications the alphabet itself is binary. While at first sight this may seem odd (Why would one need an encoder to encode a binary sequence?), a little bit of thought makes the reasoning apparent. Consider a binary source that puts out one symbol with probability 0.125 and the other with probability 0.875. If we directly encode one of the letters as a 0 and the other as a 1, we would get a rate of 1 bit/symbol. However, the entropy of the source is 0.543; so by directly encoding the output of the source as a 0 or a 1, we are encoding at almost twice the rate we need to. In order to achieve the entropy, we need to encode the more probable symbol at a fractional bit rate, which is exactly what arithmetic coding allows us to do; hence the popularity of the arithmetic code for encoding binary sources with highly skewed probabilities. The source can be binary by its nature, such as bilevel documents; or the binary input may be the binary representation of nonbinary data, as is the case for the Context Adaptive Binary Arithmetic Coder (CABAC) in the H.264 video coding standard.

As for arithmetic coding in general, the binary arithmetic coder also requires a probability model. However, because there are only two letters in the alphabet, the probability model consists of a single number, namely the probability of one of the symbols; the probability of the other symbol is simply one minus the specified probability. Because we only need a single number to represent the probability model, it is easy to use multiple arithmetic codes to encode a source where different models represent different contexts. This results in a much more accurate modeling of the source than would have been possible with a single model, which in turn leads to better compression.

Example 4.6.1:

Consider a binary sequence generated by scanning a document where the white pixels are represented by a 0 and the black pixels by a 1. Clearly the probability of a pixel being white or black depends heavily on whether the neighboring pixel is white or black. For this example we will use a first order context, so we have two sets of *Count* and *Cum_Count* tables:

$Count^0(0) = 8$	$Cum\_Count^0(-1) = 0$	$Count^1(0) = 2$	$Cum\_Count^1(-1) = 0$
$Count^0(1) = 2$	$Cum\_Count^0(0) = 8$	$Count^1(1) = 8$	$Cum\_Count^1(0) = 2$
$Total\_Count^0 = 10$	$Cum\_Count^0(1) = 10$	$Total\_Count^1(0) = 10$	$Cum\_Count^1(1) = 10$

where the superscript indicates whether the previous pixel was a 0 or a 1. We will always assume that for each row there is an imaginary white pixel to the left of the leftmost pixel. Therefore, we will always begin our encoding using the Cum_Count^0 array. After encoding the first pixel all other pixels will be encoded using the *Cum_Count* array corresponding to the previous pixel. Assume we wish to encode the sequence

000000111111

Notice that the marginal probabilities of 1 and 0 are 1/2 so if we used an arithmetic coder that did not take the context into account we would end up with a 1 bit/pixel encoding. Using the conditional probability tables we hope to do better. We will use a word length *m* of six:

$$l^{(0)} = 0 = (000000)_2 \tag{56}$$

$$u^{(0)} = 63 = (111111)_2 \tag{57}$$

The first bit to be encoded is 0:

$$\begin{aligned} l^{(1)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^0(-1)}{10} \right\rfloor = \left\lfloor \frac{64 \times 0}{10} \right\rfloor = 0 \\ &= (000000)_2 \\ u^{(1)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^0(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{64 \times 8}{10} \right\rfloor - 1 = 50 \\ &= (110010)_2 \end{aligned}$$

The second bit to be encoded is also 0:

$$\begin{aligned} l^{(2)} &= 0 + \left\lfloor \frac{51 \times Cum\_Count^0(-1)}{10} \right\rfloor = \left\lfloor \frac{51 \times 0}{10} \right\rfloor = 0 \\ &= (000000)_2 \\ u^{(2)} &= 0 + \left\lfloor \frac{51 \times Cum\_Count^0(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{51 \times 8}{10} \right\rfloor - 1 = 39 \\ &= (100111)_2 \end{aligned}$$

The third bit to be encoded is also 0:

$$\begin{aligned} l^{(3)} &= 0 + \left\lfloor \frac{40 \times Cum\_Count^0(-1)}{10} \right\rfloor = \left\lfloor \frac{40 \times 0}{10} \right\rfloor = 0 \\ &= (\mathbf{0}00000)_2 \\ u^{(3)} &= 0 + \left\lfloor \frac{40 \times Cum\_Count^0(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{40 \times 8}{10} \right\rfloor - 1 = 31 \\ &= (\mathbf{0}11111)_2 \end{aligned}$$

The MSBs of $l^{(3)}$ and $u^{(3)}$ are both 0. Therefore, we shift this value out and send it to the decoder. All other bits are shifted to the left and we get

$$\begin{aligned} l^{(3)} &= (000000)_2 = 0 \\ u^{(3)} &= (111111)_2 = 63 \end{aligned}$$

Encoding another 0, we obtain

$$\begin{aligned}
l^{(4)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(-1)}{10} \right\rfloor = \left\lfloor \frac{64 \times 0}{10} \right\rfloor = 0 \\
&= (000000)_2 \\
u^{(4)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{64 \times 8}{10} \right\rfloor - 1 = 50 \\
&= (110010)_2
\end{aligned}$$

Continuing with the next two 0s we get

$$\begin{aligned}
l^{(5)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(-1)}{10} \right\rfloor = \left\lfloor \frac{51 \times 0}{10} \right\rfloor = 0 \\
&= (000000)_2 \\
u^{(5)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{51 \times 8}{10} \right\rfloor - 1 = 39 \\
&= (100111)_2 \\
l^{(6)} &= 0 + \left\lfloor \frac{40 \times Cum\_Count^{0}(-1)}{10} \right\rfloor = \left\lfloor \frac{40 \times 0}{10} \right\rfloor = 0 \\
&= (\mathbf{0}00000)_2 \\
u^{(6)} &= 0 + \left\lfloor \frac{40 \times Cum\_Count^{0}(0)}{10} \right\rfloor - 1 = \left\lfloor \frac{40 \times 8}{10} \right\rfloor - 1 = 31 \\
&= (\mathbf{0}11111)_2
\end{aligned}$$

The MSBs of $l^{(6)}$ and $u^{(6)}$ are both 0, therefore, we shift out a 0 from both $l^{(6)}$ and $u^{(6)}$:

$$\begin{aligned}
l^{(6)} &= (000000)_2 = 0 \\
u^{(6)} &= (111111)_2 = 63
\end{aligned}$$

The next bit to be encoded is a 1. However the bit prior to it was a 0, therefore we use the Cum_Count^{0} tables:

$$\begin{aligned}
l^{(7)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(0)}{10} \right\rfloor = \left\lfloor \frac{64 \times 8}{10} \right\rfloor = 51 \\
&= (110011)_2 \\
u^{(7)} &= 0 + \left\lfloor \frac{64 \times Cum\_Count^{0}(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{64 \times 10}{10} \right\rfloor - 1 = 63 \\
&= (111111)_2
\end{aligned}$$

The first two MSBs of the upper and lower limit are the same, so we shift out two bits, 11, from both the upper and the lower limit:

$$\begin{aligned}
l^{(7)} &= (001100)_2 = 12 \\
u^{(7)} &= (111111)_2 = 63
\end{aligned}$$

The next bit to be encoded is also a 1. However, as the bit prior to that was a 1 we use the Cum_Count^1 tables:

$$\begin{aligned} l^{(8)} &= 12 + \left\lfloor \frac{52 \times Cum\_Count^1(0)}{10} \right\rfloor = \left\lfloor \frac{52 \times 2}{10} \right\rfloor = 22 \\ &= (010110)_2 \\ u^{(8)} &= 12 + \left\lfloor \frac{52 \times Cum\_Count^1(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{52 \times 10}{10} \right\rfloor - 1 = 63 \\ &= (111111)_2 \end{aligned}$$

Encoding the next two 1s we obtain

$$\begin{aligned} l^{(9)} &= 22 + \left\lfloor \frac{42 \times Cum\_Count^1(0)}{10} \right\rfloor = \left\lfloor \frac{42 \times 2}{10} \right\rfloor = 30 \\ &= (011110)_2 \\ u^{(9)} &= 22 + \left\lfloor \frac{42 \times Cum\_Count^1(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{42 \times 10}{10} \right\rfloor - 1 = 63 \\ &= (111111)_2 \\ l^{(10)} &= 30 + \left\lfloor \frac{34 \times Cum\_Count^1(0)}{10} \right\rfloor = \left\lfloor \frac{34 \times 2}{10} \right\rfloor = 36 \\ &= (100100)_2 \\ u^{(10)} &= 30 + \left\lfloor \frac{34 \times Cum\_Count^1(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{34 \times 10}{10} \right\rfloor - 1 = 63 \\ &= (111111)_2 \end{aligned}$$

The MSBs of the upper and lower limits are both equal to 1 so we shift out 1:

$$\begin{aligned} l^{(10)} &= (001000)_2 = 8 \\ u^{(10)} &= (111111)_2 = 63 \end{aligned}$$

Encoding the next two 1s we get

$$\begin{aligned} l^{(11)} &= 8 + \left\lfloor \frac{56 \times Cum\_Count^1(0)}{10} \right\rfloor = \left\lfloor \frac{56 \times 2}{10} \right\rfloor = 19 \\ &= (001011)_2 \\ u^{(11)} &= 8 + \left\lfloor \frac{56 \times Cum\_Count^1(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{56 \times 10}{10} \right\rfloor - 1 = 63 \\ &= (111111)_2 \\ l^{(12)} &= 19 + \left\lfloor \frac{45 \times Cum\_Count^1(0)}{10} \right\rfloor = \left\lfloor \frac{45 \times 2}{10} \right\rfloor = 28 \\ &= (011100)_2 \\ u^{(11)} &= 19 + \left\lfloor \frac{45 \times Cum\_Count^1(1)}{10} \right\rfloor - 1 = \left\lfloor \frac{45 \times 10}{10} \right\rfloor - 1 = 63 \\ &= (111111)_2 \end{aligned}$$

We have encoded twelve bits, and the sequence of bits generated by the encoder until this point is 00111. In other words we have a coding rate of 5/12 bits/pixel, which is less than half the coding rate we would have achieved had we used a single *Cum_Count* table. At this point if we wish to terminate the encoding we would have to incur the overhead of transmitting the bits in the lower limit. The six bits of the lower limit would be a significant overhead for this toy sequence. However, in practice, when the input sequence is much longer than twelve, the six-bit overhead would be negligible. We leave the decoding of this sequence as an exercise for the reader. ♦

Furthermore, the simple nature of the coder allows for approximations that result in simple and fast implementations. We will look at three applications of the binary coder including the QM coder used in the JBIG standard for encoding bilevel images and the M (or modulo) coder, which is a part of the coder CABAC used in the H.264 video coding standard.

Before we describe the particular implementations, let us take a general view of binary arithmetic coding. 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 many of the binary coders, which track the lower end of the tag interval $l^{(n)}$ and the size of the interval $A^{(n)}$, where

$$A^{(n)} = u^{(n)} - l^{(n)} \tag{58}$$

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

We can obtain the update equation for $A^{(n)}$ by subtracting Equation (9) from Equation (10) and substituting $A^{(n)}$ for $u^{(n)} - l^{(n)}$.

$$A^{(n)} = A^{(n-1)}(F_X(x_n) - F_X(x_n - 1)) \tag{59}$$

$$= A^{(n-1)}P(x_n) \tag{60}$$

Substituting $A^{(n)}$ for $u^{(n)} - l^{(n)}$ in Equation (9), we get the update equation for $l^{(n)}$:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}F_X(x_n - 1) \tag{61}$$

Instead of dealing directly with the 0s and 1s put out by the source, many of the binary coders map 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 update equations

$$l^{(n)} = l^{(n-1)} \tag{62}$$

$$A^{(n)} = A^{(n-1)}(1 - q_c) \tag{63}$$

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

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c) \tag{64}$$

$$A^{(n)} = A^{(n-1)}q_c \tag{65}$$

4.6.1 The QM Coder

Until this point, the binary coder looks very much like the arithmetic coder described earlier in this chapter. To make the implementation simpler and computationally more efficient, the Joint Bi-level Image Experts Group (JBIG) recommended several deviations from the standard arithmetic coding algorithm for the version of the arithmetic coder used in the JBIG algorithm for compression of bi-level images. 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

$$\text{For MPS}:$$

$$l^{(n)} = l^{(n-1)} \tag{66}$$

$$A^{(n)} = 1 - q_c \tag{67}$$

$$\text{For LPS}:$$

$$l^{(n)} = l^{(n-1)} + (1 - q_c) \tag{68}$$

$$A^{(n)} = q_c \tag{69}$$

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 occur 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.

4.6.2 The MQ Coder

The MQ coder is a variant of the QM coder. Unlike the QM coder the MQ coder assigns the lower interval to the LPS and the upper interval to the MPS. The update equations in this case without any approximations would become

$$\begin{aligned} &\text{For MPS :} \\ l^{(n)} &= l^{(n-1)} + A^{(n-1)}q_c \qquad (70) \\ A^{(n)} &= A^{(n-1)}(1 - q_c) \qquad (71) \\ &\text{For LPS :} \\ l^{(n)} &= l^{(n-1)} \qquad (72) \\ A^{(n)} &= A^{(n-1)}q_c \qquad (73) \end{aligned}$$

However, as in the case of the QM coder, we wish to avoid multiplication; therefore, with the same assumption that $A^{(n)}$ has a value close to one we modify the update equations to

$$\begin{aligned} &\text{For MPS :} \\ l^{(n)} &= l^{(n-1)} + q_c \qquad (74) \\ A^{(n)} &= A^{(n-1)} - q_c \qquad (75) \\ &\text{For LPS :} \\ l^{(n)} &= l^{(n-1)} \qquad (76) \\ A^{(n)} &= q_c \qquad (77) \end{aligned}$$

The adaptation in the MQ coder is modeled by a state machine. In practice, the A and l registers are assigned 16 bits of precision. When the value of A falls below 0x8000, it is left shifted until the value reaches or exceeds 0x8000. The same operation is performed on the register where l is stored, and the bits shifted out of the l register become the output codewords of the arithmetic coder.

4.6.3 The M Coder

The M coder is another variant of the QM coder in which the multiply operation is replaced with a table lookup. To better understand the approximations used by the M coder, let us rewrite the update equations without approximation. The occurrence of an MPS symbol results in the update equations

$$l^{(n)} = l^{(n-1)} \qquad (78)$$

$$A^{(n)} = A^{(n-1)} - A^{(n-1)}q_c \qquad (79)$$

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

$$l^{(n)} = l^{(n-1)} + A^{(n-1)} - A^{(n-1)}q_c \qquad (80)$$

$$A^{(n)} = A^{(n-1)}q_c \qquad (81)$$

Notice that the only multiplications are between the estimate of the range $A^{(n-1)}$ and the probability of the LPS q_c. The M coder gets rid of the costly multiplication operation by allowing the range and the probability to take on only a specified number of values and then replacing the multiplication by a table lookup indexed by the quantized values of the range and the LPS probability [44]. Given a minimum value A_{min} and a maximum value A_{max} for the range A, the range is restricted to four values:

$$A = A_{min} + (k+1)\Delta A, \; k = 0, 1, 2, 3$$

where

$$\Delta A = \frac{A_{max} - A_{min}}{4}$$

The LPS probability q_c can take on one of 64 possible values q_m where

$$q_m = \alpha q_{m-1}, \; m = 1, 2, \cdots, 63 \tag{82}$$

where

$$\alpha = \left(\frac{0.01875}{0.5}\right)^{\frac{1}{63}} \text{ and } q_0 = 0.5$$

In the update equation, instead of multiplying the range and the lower limit, the range is mapped to the closest of the four quantized ranges, the corresponding index k along with the index m of the probability of the LPS are used as pointers into a lookup table, and the product is read from the table.

In order to make the coder adaptive, all we need to do is update the value of the LPS probability as we see more data. If we see more occurrences of MPS, we should decrease the LPS probability. If we encounter more occurrences of LPS, we should increase the LPS probability. The M coder needs to do this while keeping the LPS probability restricted to the 64 allowed probabilities generated by Equation (82). This can be done by simply incrementing or decrementing the index m of q_m. When an MPS is encountered, we increment the index and when an LPS is encountered, we decrement the index. The index is incremented by one each time an MPS is encountered until it reaches the maximum allowed value where it remains. When an LPS is encountered, the index is decremented by a variable amount until the index reaches 0. At this point the LPS probability is one half and further occurrence of LPS indicates that this symbol is no longer the least probable and therefore, the value of MPS is changed. In other words, the MPS and LPS symbols can be swapped. The M coder forms the core of the CABAC coder, which is part of the H.264 standard for video compression.

4.7 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

$$l = 2 \times 0.5 + 3 \times 0.25 + 4 \times 0.125 + 4 \times 0.125 \tag{83}$$

$$= 2.75 \text{ bits/symbol} \tag{84}$$

TABLE 4.6 Arithmetic code for two-symbol sequences.

Message	$P(x)$	$\overline{T}_X(x)$	$\overline{T}_X(x)$ 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

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 4.7.1:

If we encode two symbols at a time, the resulting code is shown in Table4.6.

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

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

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

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

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 by 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 [45]).

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.

4.8 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.

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 because the alphabet size for the images is quite large, the value of $p_{\max}$ is quite small, and 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.

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

4.9 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 [46]. This code can be used (with some modifications) for exploring different aspects of arithmetic coding (see the following 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 [47] in the *Lossless Compression Handbook*.
3. There is an excellent tutorial article by G.G. Langdon, Jr. [48] 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 [49].
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 [50].
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 [51] 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 in the November 1988 issue of the *IBM Journal of Research and Development*.

4.10 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 grayscale 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 7.13.

TABLE 4.9 Probability model for Problems 5 and 6.

Letter	Probability
a_1	0.2
a_2	0.3
a_3	0.5

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 $b_0b_1b_2\ldots b_7$ to the Gray code $g_0g_1g_2\ldots g_7$, 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.

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

5. Given the probability model in Table4.9, find the real valued tag for the sequence $a_1a_1a_3a_2a_3a_1$.

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

7. Consider the frequency counts shown in Table4.10:

(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.

9. Decode the bitstream generated in Example 4.6.1. Do not forget to append the bits in the lower limit to terminate the bitstream.

10. Generate a random binary sequence of length 100,000 with $P(0) = 0.75$.

TABLE 4.10 Frequency counts for Problem 7.

Letter	Count
a	37
b	38
c	25

(a) Encode this sequence using a binary arithmetic coder as described by the update Equations (60) and (61). Note the size of the compressed sequence.

(b) Implement the QM coder described in Section 4.6.1 and encode the binary sequence using this coder. Compare the length of the compressed sequence. Comment on any differences.

11. Generate a binary sequence of length 100,000 using the model $P[0|0] = P[1|1] = 0.9$.

(a) Encode this sequence using a binary arithmetic coder as described by the update Equations (60) and (61). Note the size of the compressed sequence.

(b) Encode the sequence using a context-based arithmetic coder where the context is determined by the previous two bits. Compare the difference in the size of the compressed sequences.