Introduction to Arithmetic Coding - Theory and Practice

Amir Said
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2004-76
April 21, 2004*

entropy coding, compression, complexity

This introduction to arithmetic coding is divided in two parts. The first explains how and why arithmetic coding works. We start presenting it in very general terms, so that its simplicity is not lost under layers of implementation details. Next, we show some of its basic properties, which are later used in the computational techniques required for a practical implementation.

In the second part, we cover the practical implementation aspects, including arithmetic operations with low precision, the subdivision of coding and modeling, and the realization of adaptive encoders. We also analyze the arithmetic coding computational complexity, and techniques to reduce it.

We start some sections by first introducing the notation and most of the mathematical definitions. The reader should not be intimidated if at first their motivation is not clear: these are always followed by examples and explanations.

* Internal Accession Date Only
Published as a chapter in *Lossless Compression Handbook* by Khalid Sayood
Approved for External Publication

Contents

Chapter 1

Arithmetic Coding Principles

1.1 Data Compression and Arithmetic Coding

Compression applications employ a wide variety of techniques, have quite different degrees of complexity, but share some common processes. Figure 1.1 shows a diagram with typical processes used for data compression. These processes depend on the data type, and the blocks in Figure 1.1 may be in different order or combined. Numerical processing, like predictive coding and linear transforms, is normally used for waveform signals, like images and audio [20, 35, 36, 48, 55]. Logical processing consists of changing the data to a form more suited for compression, like run-lengths, zero-trees, set-partitioning information, and dictionary entries [3, 20, 38, 40, 41, 44, 47, 55]. The next stage, source modeling, is used to account for variations in the statistical properties of the data. It is responsible for gathering statistics and identifying data contexts that make the source models more accurate and reliable [14, 28, 29, 45, 46, 49, 53].

What most compression systems have in common is the fact that the final process is *entropy coding*, which is the process of representing information in the most compact form. It may be responsible for doing most of the compression work, or it may just complement what has been accomplished by previous stages.

When we consider all the different entropy-coding methods, and their possible applications in compression applications, *arithmetic coding* stands out in terms of elegance, effectiveness and versatility, since it is able to work most efficiently in the largest number of circumstances and purposes. Among its most desirable features we have the following.

- When applied to independent and identically distributed (i.i.d.) sources, the compression of each symbol is provably optimal (Section 1.5).

- It is effective in a wide range of situations and compression ratios. The same arithmetic coding implementation can effectively code all the diverse data created by the different processes of Figure 1.1, such as modeling parameters, transform coefficients, signaling, etc. (Section 1.6.1).

- It simplifies automatic modeling of complex sources, yielding near-optimal or significantly improved compression for sources that are not i.i.d (Section 1.6.3).

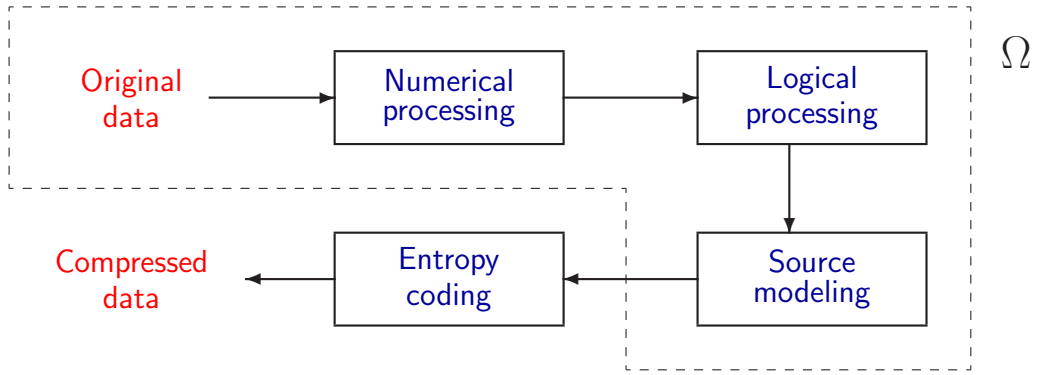

Figure 1.1: System with typical processes for data compression. Arithmetic coding is normally the final stage, and the other stages can be modeled as a single data source Ω.

- Its main process is arithmetic, which is supported with ever-increasing efficiency by all general-purpose or digital signal processors (CPUs, DSPs) (Section 2.3).
- It is suited for use as a "compression black-box" by those that are not coding experts or do not want to implement the coding algorithm themselves.

Even with all these advantages, arithmetic coding is not as popular and well understood as other methods. Certain practical problems held back its adoption.

- The complexity of arithmetic operations was excessive for coding applications.
- Patents covered the most efficient implementations. Royalties and the fear of patent infringement discouraged arithmetic coding in commercial products.
- Efficient implementations were difficult to understand.

However, these issues are now mostly overcome. First, the relative efficiency of computer arithmetic improved dramatically, and new techniques avoid the most expensive operations. Second, some of the patents have expired (e.g., [11, 16]), or became obsolete. Finally, we do not need to worry so much about complexity-reduction details that obscure the inherent simplicity of the method. Current computational resources allow us to implement simple, efficient, and royalty-free arithmetic coding.

1.2 Notation

Let Ω be a data source that puts out symbols s_k coded as integer numbers in the set $\{0, 1, \ldots, M-1\}$, and let $S = \{s_1, s_2, \ldots, s_N\}$ be a sequence of N random symbols put out by Ω [1, 4, 5, 21, 55, 56]. For now, we assume that the source symbols are independent and identically distributed [22], with probability

$$p(m) = \text{Prob}\{s_k = m\}, \quad m = 0, 1, 2, \ldots, M-1, \quad k = 1, 2, \ldots, N. \tag{1.1}$$

We also assume that for all symbols we have $p(m) \neq 0$, and define $c(m)$ to be the cumulative distribution,

$$c(m) = \sum_{s=0}^{m-1} p(s), \quad m = 0, 1, \ldots, M. \tag{1.2}$$

Note that $c(0) \equiv 0,\ c(M) \equiv 1$, and

$$p(m) = c(m+1) - c(m). \tag{1.3}$$

We use bold letters to represent the vectors with all $p(m)$ and $c(m)$ values, i.e.,

$$\begin{aligned} \mathbf{p} &= [\, p(0)\ p(1)\ \cdots\ p(M-1)\,], \\ \mathbf{c} &= [\, c(0)\ c(1)\ \cdots\ c(M-1)\ c(M)\,]. \end{aligned}$$

We assume that the compressed data (output of the encoder) is saved in a vector (buffer) $\mathbf{d}$. The output alphabet has D symbols, i.e., each element in $\mathbf{d}$ belongs to set $\{0, 1, \ldots, D-1\}$.

Under the assumptions above, an optimal coding method [1] codes each symbol s from Ω with an average number of bits equal to

$$\mathrm{B}(s) = -\log_2 p(s) \quad \text{bits}. \tag{1.4}$$

Example 1

◁ Data source Ω can be a file with English text: each symbol from this source is a single byte representatinh a character. This data alphabet contains $M = 256$ symbols, and symbol numbers are defined by the ASCII standard. The probabilities of the symbols can be estimated by gathering statistics using a large number of English texts. Table 1.1 shows some characters, their ASCII symbol values, and their estimated probabilities. It also shows the number of bits required to code symbol s in an optimal manner, $-\log_2 p(s)$. From these numbers we conclude that, if data symbols in English text were i.i.d., then the best possible text compression ratio would be about 2:1 (4 bits/symbol). Specialized text compression methods [8, 10, 29, 41] can yield significantly better compression ratios because they exploit the statistical dependence between letters. ▷

This first example shows that our initial assumptions about data sources are rarely found in practical cases. More commonly, we have the following issues.

1. The source symbols are not identically distributed.
2. The symbols in the data sequence are not independent (even if uncorrelated) [22].
3. We can only *estimate* the probability values, the statistical dependence between symbols, and how they change in time.

However, in the next sections we show that the generalization of arithmetic coding to time-varying sources is straightforward, and we explain how to address all these practical issues.

Character	ASCII Symbol s	Probability $p(s)$	Optimal number of bits $-\log_2 p(s)$
Space	32	0.1524	2.714
,	44	0.0136	6.205
.	46	0.0056	7.492
A	65	0.0017	9.223
B	66	0.0009	10.065
C	67	0.0013	9.548
a	97	0.0595	4.071
b	98	0.0119	6.391
c	99	0.0230	5.441
d	100	0.0338	4.887
e	101	0.1033	3.275
f	102	0.0227	5.463
t	116	0.0707	3.823
z	122	0.0005	11.069

Table 1.1: Estimated probabilities of some letters and punctuation marks in the English language. Symbols are numbered according to the ASCII standard.

1.3 Code Values

Arithmetic coding is different from other coding methods for which we know the exact relationship between the coded symbols and the actual bits that are written to a file. It codes one data symbol at a time, and assigns to each symbol a real-valued number of bits (see examples in the last column of Table 1.1). To figure out how this is possible, we have to understand the *code value* representation: coded messages mapped to real numbers in the interval [0, 1).

The code value v of a compressed data sequence is the real number with fractional digits equal to the sequence's symbols. We can convert sequences to code values by simply adding "0." to the beginning of a coded sequence, and then interpreting the result as a number in base-D notation, where D is the number of symbols in the coded sequence alphabet. For example, if a coding method generates the sequence of bits 0011000101100, then we have

$$\begin{aligned} &\text{Code sequence} \quad \mathbf{d} = \quad [\,\underbrace{0011000101100}\,] \\ &\text{Code value} \qquad\quad v = \quad 0.\overbrace{0011000101100}_{2} \quad = 0.19287109375 \end{aligned} \tag{1.5}$$

where the "2" subscript denotes base-2 notation. As usual, we omit the subscript for decimal notation.

This construction creates a convenient mapping between infinite sequences of symbols from a D-symbol alphabet and real numbers in the interval [0, 1), where any data sequence can be represented by a real number, and vice-versa. The code value representation can be used for any coding system and it provides a universal way to represent large amounts of

information independently of the set of symbols used for coding (binary, ternary, decimal, etc.). For instance, in (1.5) we see the same code with base-2 and base-10 representations.

We can evaluate the efficacy of any compression method by analyzing the distribution of the code values it produces. From Shannon's information theory [1] we know that, if a coding method is optimal, then the cumulative distribution [22] of its code values has to be a straight line from point (0, 0) to point (1, 1).

Example 2

◁ Let us assume that the i.i.d. source Ω has four symbols, and the probabilities of the data symbols are $\mathbf{p} = [\,0.65 \;\; 0.2 \;\; 0.1 \;\; 0.05\,]$. If we code random data sequences from this source with two bits per symbols, the resulting code values produce a cumulative distribution as shown in Figure 1.2, under the label "uncompressed." Note how the distribution is skewed, indicating the possibility for significant compression.

The same sequences can be coded with the Huffman code for Ω [2, 4, 21, 55, 56], with one bit used for symbol "0", two bits for symbol "1", and three bits for symbols "2" and "3". The corresponding code value cumulative distribution in Figure 1.2 shows that there is substantial improvement over the uncompressed case, but this coding method is still clearly not optimal. The third line in Figure 1.2 shows that the sequences compressed with arithmetic coding simulation produce a code value distribution that is practically identical to the optimal. ▷

The straight-line distribution means that if a coding method is optimal then there is no statistical dependence or redundancy left in the compressed sequences, and consequently its code values are uniformly distributed on the interval [0, 1). This fact is essential for understanding of how arithmetic coding works. Moreover, code values are an integral part of the arithmetic encoding/decoding procedures, with arithmetic operations applied to real numbers that are directly related to code values.

One final comment about code values: two infinitely long different sequences can correspond to the same code value. This follows from the fact that for any $D > 1$ we have

$$\sum_{n=k}^{\infty} (D-1)D^{-n} = D^{1-k}. \tag{1.6}$$

For example, if $D = 10$ and $k = 2$, then (1.6) is the equality $0.09999999\ldots = 0.1$. This fact has no important practical significance for coding purposes, but we need to take it into account when studying some theoretical properties of arithmetic coding.

1.4 Arithmetic Coding

1.4.1 Encoding Process

In this section we first introduce the notation and equations that describe arithmetic encoding, followed by a detailed example. Fundamentally, the arithmetic encoding process consists of creating a sequence of nested intervals in the form $\Phi_k(S) = [\,\alpha_k,\, \beta_k\,)\,,\; k = 0, 1, \ldots, N,$

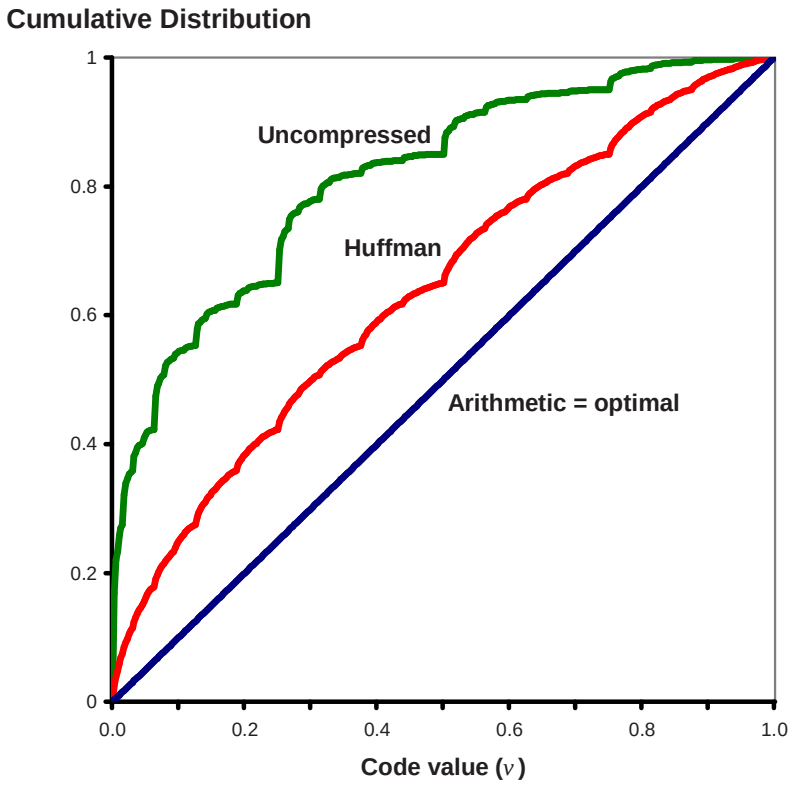

Figure 1.2: Cumulative distribution of code values generated by different coding methods when applied to the source of Example 2.

where S is the source data sequence, α_k and β_k are real numbers such that $0 \leq \alpha_k \leq \alpha_{k+1}$, and $\beta_{k+1} \leq \beta_k \leq 1$. For a simpler way to describe arithmetic coding we represent intervals in the form $|\, b,\ l \,\rangle$, where b is called the *base* or *starting point* of the interval, and l the *length* of the interval. The relationship between the traditional and the new interval notation is

$$|\, b,\ l \,\rangle = [\, \alpha,\ \beta \,) \quad \text{if} \quad b = \alpha \quad \text{and} \quad l = \beta - \alpha. \tag{1.7}$$

The intervals used during the arithmetic coding process are, in this new notation, defined by the set of recursive equations [5, 13]

$$\Phi_0(S) = |\, b_0,\ l_0 \,\rangle = |\, 0,\ 1 \,\rangle\,, \tag{1.8}$$

$$\Phi_k(S) = |\, b_k,\ l_k \,\rangle = |\, b_{k-1} + c(s_k)\, l_{k-1},\ p(s_k)\, l_{k-1} \,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.9}$$

The properties of the intervals guarantee that $0 \leq b_k \leq b_{k+1} < 1$, and $0 < l_{k+1} < l_k \leq 1$. Figure 1.3 shows a dynamic system corresponding to the set of recursive equations (1.9). We later explain how to choose, at the end of the coding process, a code value in the final interval, i.e., $\hat{v}(S) \in \Phi_N(S)$.

The coding process defined by (1.8) and (1.9), also called *Elias coding,* was first described in [5]. Our convention of representing an interval using its base and length has been used

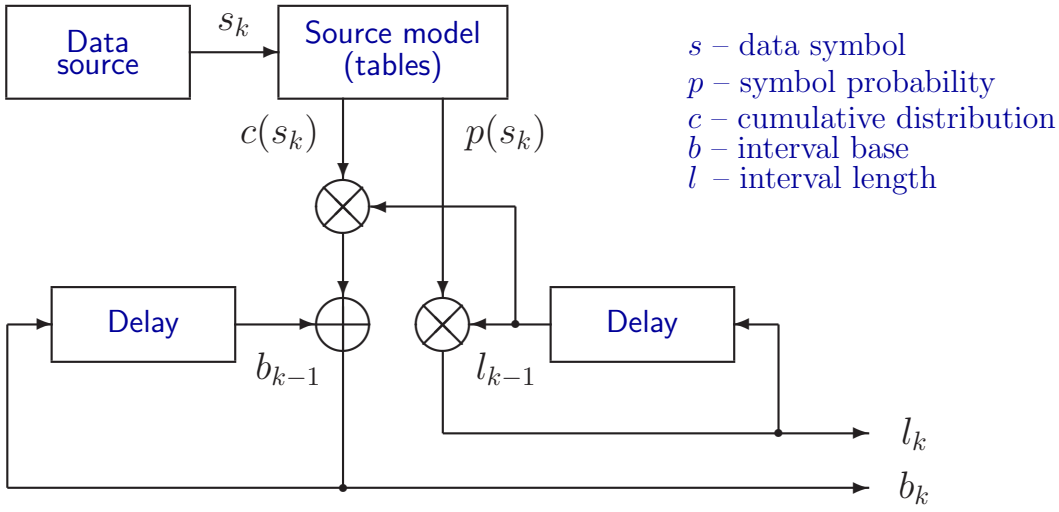

Figure 1.3: Dynamic system for updating arithmetic coding intervals.

since the first arithmetic coding papers [12, 13]. Other authors have intervals represented by their extreme points, like [base, base+length), but there is no mathematical difference between the two notations.

Example 3

◁ Let us assume that source Ω has four symbols ($M = 4$), the probabilities and distribution of the symbols are $\mathbf{p} = [\,0.2 \;\; 0.5 \;\; 0.2 \;\; 0.1\,]$ and $\mathbf{c} = [\,0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1\,]$, and the sequence of ($N = 6$) symbols to be encoded is $S = \{2, 1, 0, 0, 1, 3\}$.

Figure 1.4 shows graphically how the encoding process corresponds to the selection of intervals in the line of real numbers. We start at the top of the figure, with the interval [0, 1), which is divided into four subintervals, each with length equal to the probability of the data symbols. Specifically, interval [0, 0.2) corresponds to $s_1 = 0$, interval [0.2, 0.7) corresponds to $s_1 = 1$, interval [0.7, 0.9) corresponds to $s_1 = 2$, and finally interval [0.9, 1) corresponds to $s_1 = 3$. The next set of allowed nested subintervals also have length proportional to the probability of the symbols, but their lengths are also proportional to the length of the interval they belong to. Furthermore, they represent more than one symbol value. For example, interval [0, 0.04) corresponds to $s_1 = 0,\ s_2 = 0$, interval [0.04, 0.14) corresponds to $s_1 = 0,\ s_2 = 1$, and so on.

The interval lengths are reduced by factors equal to symbol probabilities in order to obtain code values that are uniformly distributed in the interval [0, 1) (a necessary condition for optimality, as explained in Section 1.3). For example, if 20% of the sequences start with symbol "0", then 20% of the code values must be in the interval assigned to those sequences, which can only be achieved if we assign to the first symbol "0" an interval with length equal to its probability, 0.2. The same reasoning applies to the assignment of the subinterval lengths: every occurrence of symbol "0" must result in a reduction of the interval length to 20% its current length. This way, after encoding

Iteration k	**Input Symbol** s_k	**Interval base** b_k	**Interval length** l_k	**Decoder updated value** $\tilde{v}_k = \frac{\hat{v}-b_{k-1}}{l_{k-1}}$	**Output symbol** $\hat{s}_k$
0	—	0	1	—	—
1	2	0.7	0.2	0.74267578125	2
2	1	0.74	0.1	0.21337890625	1
3	0	0.74	0.02	0.0267578125	0
4	0	0.74	0.004	0.1337890625	0
5	1	0.7408	0.002	0.6689453125	1
6	3	0.7426	0.0002	0.937890625	3
7	—	—	—	0.37890625	1
8	—	—	—	0.3578125	1

Table 1.2: Arithmetic encoding and decoding results for Examples 3 and 4. The last two rows show what happens when decoding continues past the last symbol.

several symbols the distribution of code values should be a very good approximation of a uniform distribution.

Equations (1.8) and (1.9) provide the formulas for the sequential computation of the intervals. Applying them to our example we obtain:

$$\begin{aligned}
\Phi_0(S) &= |\,0,\,1\,\rangle = [\,0,\,1\,)\,,\\
\Phi_1(S) &= |\,b_0 + c(2)l_0,\; p(2)l_0\,\rangle = |\,0 + 0.7 \times 1,\; 0.2 \times 1\,\rangle = [\,0.7,\,0.9\,)\,,\\
\Phi_2(S) &= |\,b_1 + c(1)l_1,\; p(1)l_1\,\rangle = |\,0.7 + 0.2 \times 0.2,\; 0.5 \times 0.2\,\rangle = [\,0.74,\,0.84\,)\,,\\
&\vdots\\
\Phi_6(S) &= |\,b_5 + c(3)l_5,\; p(3)l_5\,\rangle = |\,0.7426,\; 0.0002\,\rangle = [\,0.7426,\,0.7428\,)\,,
\end{aligned}$$

The list with all the encoder intervals is shown in the first four columns of Table 1.2. Since the intervals quickly become quite small, in Figure 1.4 we have to graphically magnify them (twice) so that we can see how the coding process continues. Note that even though the intervals are shown in different magnifications, the intervals values do not change, and the process to subdivide intervals continues in exactly the same manner. ▷

The final task in arithmetic encoding is to define a code value $\hat{v}(S)$ that will represent data sequence S. In the next section we show how the decoding process works correctly for *any* code value $\hat{v} \in \Phi_N(S)$. However, the code value cannot be provided to the decoder as a pure real number. It has to be stored or transmitted, using a conventional number representation. Since we have the freedom to choose any value in the final interval, we want to choose the values with the shortest representation. For instance, in Example 3, the shortest decimal representation comes from choosing $\hat{v} = 0.7427$, and the shortest binary representation is obtained with $\hat{v} = 0.10111110001_2 = 0.74267578125$.

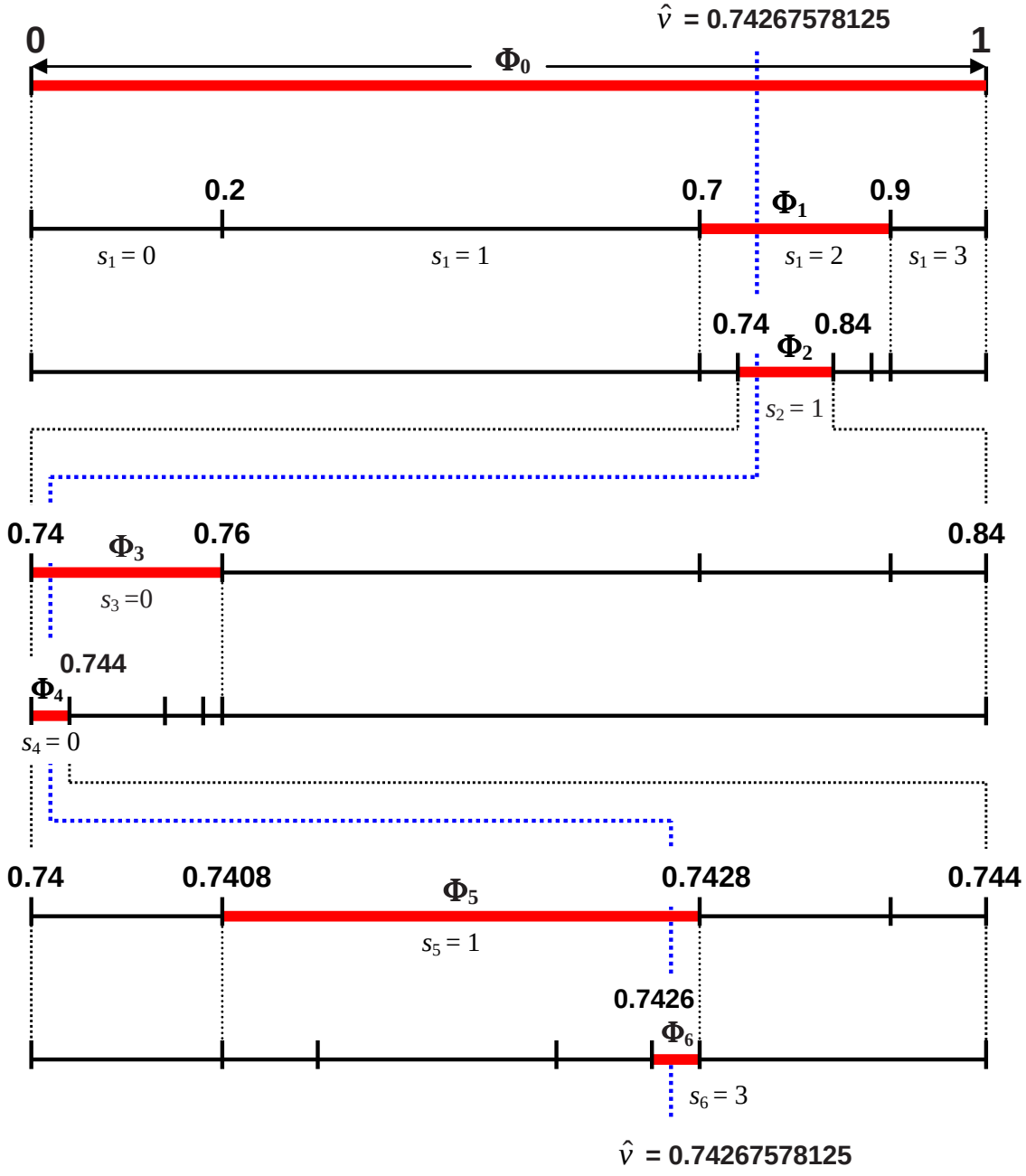

Figure 1.4: Graphical representation of the arithmetic coding process of Example 3: the interval $\Phi_0 = [0, 1)$ is divided in nested intervals according to the probability of the data symbols. The selected intervals, corresponding to data sequence $S = \{2, 1, 0, 0, 1, 3\}$ are indicated by thicker lines.

The process to find the best binary representation is quite simple and best shown by induction. The main idea is that for relatively large intervals we can find the optimal value by testing a few binary sequences, and as the interval lengths are halved, the number of sequences to be tested has to double, increasing the number of bits by one. Thus, according to the interval length l_N, we use the following rules:

- If $l_N \in [0.5,\ 1)$, then choose code value $\hat{v} \in \{0, 0.5\} = \{0.0_2, 0.1_2\}$ for a 1-bit representation.
- If $l_N \in [0.25,\ 0.5)$, then choose value $\hat{v} \in \{0, 0.25, 0.5, 0.75\} = \{0.00_2, 0.01_2, 0.10_2, 0.11_2\}$ for a 2-bit representation.
- If $l_N \in [0.125,\ 0.25)$, then choose value $\hat{v} \in \{0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875\} = \{0.000_2, 0.001_2, 0.010_2, 0.011_2, 0.100_2, 0.101_2, 0.110_2, 0.111_2\}$ for a 3-bit representation.

By observing the pattern we conclude that the minimum number of bits required for representing $\hat{v} \in \Phi_N(S)$ is

$$\mathrm{B}_{\min} = \lceil -\log_2(l_N) \rceil \quad \text{bits}, \tag{1.10}$$

where $\lceil x \rceil$ represents the smallest integer greater than or equal to x.

We can test this conclusion observing the results for Example 3 in Table 1.2. The final interval is $l_N = 0.0002$, and thus $\mathrm{B}_{\min} = \lceil -\log_2(0.0002) \rceil = 13$ bits. However, in Example 3 we can choose $\hat{v} = 0.10111110001_2$, and it requires only 11 bits!

The origin of this inconsistency is the fact that we can choose binary representations with the number of bits given by (10), and then remove the trailing zeros. However, with optimal coding the *average* number of bits that can be saved with this process is only one bit, and for that reason, it is rarely applied in practice.

1.4.2 Decoding Process

In arithmetic coding, the decoded sequence is determined solely by the code value $\hat{v}$ of the compressed sequence. For that reason, we represent the decoded sequence as

$$\hat{S}(\hat{v}) = \{\hat{s}_1(\hat{v}), \hat{s}_2(\hat{v}), \ldots, \hat{s}_N(\hat{v})\}\,. \tag{1.11}$$

We now show the decoding process by which any code value $\hat{v} \in \Phi_N(S)$ can be used for decoding the correct sequence (i.e., $\hat{S}(\hat{v}) = S$). We present the set of recursive equations that implement decoding, followed by a practical example that provides an intuitive idea of how the decoding process works, and why it is correct.

The decoding process recovers the data symbols in the same sequence that they were coded. Formally, to find the numerical solution, we define a sequence of normalized code values $\{\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_N\}$. Starting with $\tilde{v}_1 = \hat{v}$, we sequentially find $\hat{s}_k$ from $\tilde{v}_k$, and then we compute $\tilde{v}_{k+1}$ from $\hat{s}_k$ and $\tilde{v}_k$.

The recursion formulas are

$$\tilde{v}_1 = \hat{v}, \tag{1.12}$$

$$\hat{s}_k(\hat{v}) = \{\, s :\ c(s) \le \tilde{v}_k < c(s+1)\}\,, \quad k = 1, 2, \ldots, N, \tag{1.13}$$

$$\tilde{v}_{k+1} = \frac{\tilde{v}_k - c(\hat{s}_k(\hat{v}))}{p(\hat{s}_k(\hat{v}))}, \quad k = 1, 2,\ \ldots, N-1. \tag{1.14}$$

(In equation (1.13) the colon means "s that satisfies the inequalities.")

A mathematically equivalent decoding method—which later we show to be necessary when working with fixed-precision arithmetic—recovers the sequence of intervals created by the encoder, and searches for the correct value $\hat{s}_k(\hat{v})$ in each of these intervals. It is defined by

$$\Phi_0(\hat{S}) = |\, b_0,\, l_0 \,\rangle = |\, 0,\, 1 \,\rangle\,, \tag{1.15}$$

$$\hat{s}_k(\hat{v}) = \left\{ s:\ c(s) \le \frac{\hat{v} - b_{k-1}}{l_{k-1}} < c(s+1) \right\}, \quad k = 1, 2, \ldots, N, \tag{1.16}$$

$$\Phi_k(\hat{S}) = |\, b_k,\, l_k \,\rangle = |\, b_{k-1} + c(\hat{s}_k(\hat{v}))\, l_{k-1},\, p(\hat{s}_k(\hat{v}))\, l_{k-1} \,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.17}$$

The combination of recursion (1.14) with recursion (1.17) yields

$$\tilde{v}_k = \frac{\hat{v} - \sum_{i=1}^{k-1} c(\hat{s}_i) \prod_{j=1}^{i-1} p(\hat{s}_j)}{\prod_{i=1}^{k-1} p(\hat{s}_i)} = \frac{\hat{v} - b_{k-1}}{l_{k-1}}. \tag{1.18}$$

showing that (1.13) is equivalent to (1.16).

Example 4

◁ Let us apply the decoding process to the data obtained in Example 3. In Figure 1.4, we show graphically the meaning of $\hat{v}$: it is a value that belongs to all nested intervals created during coding. The dotted line shows that its position moves as we magnify the graphs, but the value remains the same. From Figure 1.4, we can see that we can start decoding from the first interval $\Phi_0(S) = [0,\, 1)$: we just have to compare $\hat{v}$ with the cumulative distribution $\mathbf{c}$ to find the only possible value of $\hat{s}_1$

$$\hat{s}_1(\hat{v}) = \{\, s:\ c(s) \le \hat{v} = 0.74267578125 < c(s+1)\} = 2.$$

We can use the value of $\hat{s}_1$ to find out interval $\Phi_1(S)$, and use it for determining $\hat{s}_2$. In fact, we can "remove" the effect of $\hat{s}_1$ in $\hat{v}$ by defining the normalized code value

$$\tilde{v}_2 = \frac{\hat{v} - c(\hat{s}_1)}{p(\hat{s}_1)} = 0.21337890625.$$

Note that, in general, $\tilde{v}_2 \in [0,\, 1)$, i.e., it is a value normalized to the initial interval. In this interval we can use the same process to find

$$\hat{s}_2(\hat{v}) = \{\, s:\ c(s) \le \tilde{v}_2 = 0.21337890625 < c(s+1)\} = 1.$$

The last columns of Table 1.2 show how the process continues, and the updated values computed while decoding. We could say that the process continues until $\hat{s}_6$ is decoded. However, how can the decoder, having only the initial code value $\hat{v}$, know that it is time to stop decoding? The answer is simple: it can't. We added two extra rows to Table 1.2 to show that the decoding process can continue normally after the last symbol is encoded. Below we explain what happens. ▷

It is important to understand that arithmetic encoding maps intervals to *sets of sequences*. Each real number in an interval corresponds to one infinite sequence. Thus, the sequences corresponding to $\Phi_6(S) = [0.7426, 0.7428)$ are all those that *start* as $\{2, 1, 0, 0, 1, 3, \ldots\}$. The code value $\hat{v} = 0.74267578125$ corresponds to one such infinite sequence, and the decoding process can go on forever decoding that particular sequence.

There are two practical ways to inform that decoding should stop:

1. Provide the number of data symbols (N) in the beginning of the compressed file.

2. Use a special symbol as "end-of-message," which is coded only at the end of the data sequence, and assign to this symbol the smallest probability value allowed by the encoder/decoder.

As we explained above, the decoding procedure will always produce a decoded data sequence. However, how do we know that it is the right sequence? This can be inferred from the fact that if S and S' are sequences with N symbols then

$$S \neq S' \Leftrightarrow \Phi_N(S) \cap \Phi_N(S') = \emptyset. \tag{1.19}$$

This guarantees that different sequences cannot produce the same code value. In Section 1.6.6 we show that, due to approximations, we have incorrect decoding if (1.19) is not satisfied.

1.5 Optimality of Arithmetic Coding

Information theory [1, 4, 5, 21, 32, 55, 56] shows us that the average number of bits needed to code each symbol from a stationary and memoryless source Ω cannot be smaller than its entropy $H(\Omega)$, defined by

$$H(\Omega) = -\sum_{m=0}^{M-1} p(m) \log_2 p(m) \quad \text{bits/symbol.} \tag{1.20}$$

We have seen that the arithmetic coding process generates code values that are uniformly distributed across the interval [0, 1). This is a necessary condition for optimality, but not a sufficient one. In the interval $\Phi_N(S)$ we can choose values that require an arbitrarily large number of bits to be represented, or choose code values that can be represented with the minimum number of bits, given by equation (1.10). Now we show that the latter choice satisfies the sufficient condition for optimality.

To begin, we have to consider that there is some overhead in a compressed file, which may include

- Extra bits required for saving $\hat{v}$ with an integer number of bytes.

- A fixed or variable number of bits representing the number of symbols coded.

- Information about the probabilities ($\mathbf{p}$ or $\mathbf{c}$).

Assuming that the total overhead is a positive number σ bits, we conclude from (1.10) that the number of bits per symbol used for coding a sequence S should be bounded by

$$\mathrm{B}_S \leq \frac{\sigma - \log_2(l_N)}{N} \quad \text{bits/symbol.} \tag{1.21}$$

It follows from (1.9) that

$$l_N = \prod_{k=1}^{N} p(s_k), \tag{1.22}$$

and thus

$$\mathrm{B}_S \leq \frac{\sigma - \sum\limits_{k=1}^{N} \log_2 p(s_k)}{N} \quad \text{bits/symbol.} \tag{1.23}$$

Defining $E\{\cdot\}$ as the expected value operator, the expected number of bits per symbol is

$$\begin{aligned} \bar{\mathrm{B}} = E\{\mathrm{B}_S\} &\leq \frac{\sigma - \sum\limits_{k=1}^{N} E\{\log_2 p(s_k)\}}{N} = \frac{\sigma - \sum\limits_{k=1}^{N} \sum\limits_{m=0}^{M-1} p(m) \log_2 p(m)}{N} \\ &\leq H(\Omega) + \frac{\sigma}{N} \end{aligned} \tag{1.24}$$

Since the average number of bits per symbol cannot be smaller than the entropy, we have

$$H(\Omega) \leq \bar{\mathrm{B}} \leq H(\Omega) + \frac{\sigma}{N}, \tag{1.25}$$

and it follows that

$$\lim_{N\to\infty} \left\{\bar{\mathrm{B}}\right\} = H(\Omega), \tag{1.26}$$

which means that arithmetic coding indeed achieves optimal compression performance.

At this point we may ask why arithmetic coding creates intervals, instead of single code values. The answer lies in the fact that arithmetic coding is optimal not only for binary output—but rather for any output alphabet. In the final interval we find the different code values that are optimal for each output alphabet. Here is an example of use with non-binary outputs.

Example 5

◁ Consider transmitting the data sequence of Example 3 using a communications system that conveys information using three levels, {–V, 0, +V} (actually used in radio remote controls). Arithmetic coding with ternary output can simultaneously compress the data and convert it to the proper transmission format.

The generalization of (1.10) for a D-symbol output alphabet is

$$\mathrm{B}_{\min}(l_N, D) = \lceil -\log_D(l_N) \rceil \quad \text{symbols.} \tag{1.27}$$

Thus, using the results in Table 1.2, we conclude that we need $\lceil -\log_3(0.0002) \rceil = 8$ ternary symbols. We later show how to use standard arithmetic coding to find that the shortest ternary representation is $\hat{v}_3 = 0.20200111_3 \approx 0.742722146$, which means that the sequence $S = \{2, 1, 0, 0, 1, 3\}$ can be transmitted as the sequence of electrical signals {+V, 0, +V, 0, 0, –V, –V, –V}. ▷

1.6 Arithmetic Coding Properties

1.6.1 Dynamic Sources

In Section 1.2 we assume that the data source Ω is stationary, so we have one set of symbol probabilities for encoding and decoding all symbols in the data sequence S. Now, with an understanding of the coding process, we generalize it for situations where the probabilities change for each symbol coded, i.e., the k-th symbol in the data sequence S is a random variable with probabilities $\mathbf{p}_k$ and distribution $\mathbf{c}_k$.

The only required change in the arithmetic coding process is that instead of using (1.9) for interval updating, we should use

$$\Phi_k(S) = |\, b_k,\, l_k \,\rangle = |\, b_{k-1} + c_k(s_k)\, l_{k-1},\, p_k(s_k)\, l_{k-1} \,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.28}$$

To understand the changes in the decoding process, remember that the process of working with updated code values is equivalent to "erasing" all information about past symbols, and decoding in the [0, 1) interval. Thus, the decoder only has to use the right set of probabilities for that symbol to decode it correctly. The required changes to (1.16) and (1.17) yield

$$\hat{s}_k(\hat{v}) = \left\{ s :\ c_k(s) \le \frac{\hat{v} - b_{k-1}}{l_{k-1}} < c_k(s+1) \right\}, \quad k = 1, 2, \ldots, N, \tag{1.29}$$

$$\Phi_k(S) = |\, b_k,\, l_k \,\rangle = |\, b_{k-1} + c_k(\hat{s}_k(\hat{v}))\, l_{k-1},\, p_k(\hat{s}_k(\hat{v}))\, l_{k-1} \,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.30}$$

Note that the number of symbols used at each instant can change. Instead of having a single input alphabet with M symbols, we have a sequence of alphabet sizes $\{M_1, M_2, \ldots, M_N\}$.

1.6.2 Encoder and Decoder Synchronized Decisions

In data compression an encoder can change its behavior (parameters, coding algorithm, etc.) while encoding a data sequence, as long as the decoder uses the same information and the same rules to change its behavior. In addition, these changes must be "synchronized," not in time, but in relation to the sequence of data source symbols.

For instance, in Section 1.6.1, we assume that the encoder and decoder are synchronized in their use of varying sets of probabilities. Note that we do not have to assume that all the probabilities are available to the decoder when it starts decoding. The probability vectors can be updated with any rule based on symbol occurrences, as long as $\mathbf{p}_k$ is computed from the data already available to the decoder, i.e., $\{\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_{k-1}\}$. This principle is used for adaptive coding, and it is covered in Section 2.2.

This concept of synchronization is essential for arithmetic coding because it involves a nonlinear dynamic system (Figure 1.3), and error accumulation leads to incorrect decoding, unless the encoder and decoder use *exactly* the same implementation (same precision, number of bits, rounding rules, equations, tables, etc.). In other words, we can make arithmetic coding work correctly even if the encoder makes coarse approximations, as long as the decoder makes exactly the same approximations. We have already seen an example of a choice based on numerical stability: equations (1.16) and (1.17) enable us to synchronize the encoder and decoder because they use the same interval updating rules used by (1.9), while (1.13) and (1.14) use a different recursion.

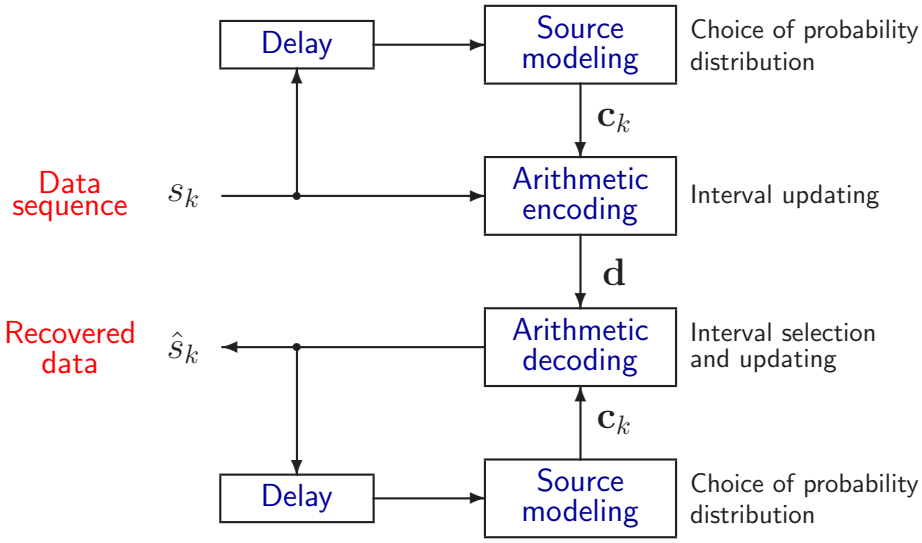

Figure 1.5: Separation of coding and source modeling tasks. Arithmetic encoding and decoding process intervals, while source modeling chooses the probability distribution for each data symbol.

1.6.3 Separation of Coding and Source Modeling

There are many advantages for separating the source modeling (probabilities estimation) and the coding processes [14, 25, 29, 38, 45, 51, 53]. For example, it allows us to develop complex compression schemes without worrying about the details in the coding algorithm, and/or use them with different coding methods and implementations.

Figure 1.5 shows how the two processes can be separated in a complete system for arithmetic encoding and decoding. The coding part is responsible only for updating the intervals, i.e., the arithmetic encoder implements recursion (1.28), and the arithmetic decoder implements (1.29) and (1.30). The encoding/decoding processes use the probability distribution vectors as input, but do not change them in any manner. The source modeling part is responsible for choosing the distribution $\mathbf{c}_k$ that is used to encode/decode symbol s_k. Figure 1.5 also shows that a delay of one data symbol before the source-modeling block guarantees that encoder and decoder use the same information to update $\mathbf{c}_k$.

Arithmetic coding simplifies considerably the implementation of systems like Figure 1.5 because the vector $\mathbf{c}_k$ is used directly for coding. With Huffman coding, changes in probabilities require re-computing the optimal code, or using complex code updating techniques [9, 24, 26].

1.6.4 Interval Rescaling

Figure 1.4 shows graphically one important property of arithmetic coding: the actual intervals used during coding depend on the initial interval and the previously coded data, but the proportions within subdivided intervals do not. For example, if we change the initial interval to $\Phi_0 = |\,1,\,2\,\rangle = [\,1,\,3\,)$ and apply (1.9), the coding process remains the same, except that

all intervals are scaled by a factor of two, and shifted by one.

We can also apply rescaling in the middle of the coding process. Suppose that at a certain stage m we change the interval according to

$$b'_m = \gamma\,(b_m - \delta), \quad l'_m = \gamma\, l_m, \tag{1.31}$$

and continue the coding process normally (using (1.9) or (1.28)). When we finish coding we obtain the interval $\Phi'_N(S) = |\, b'_N,\, l'_N \,\rangle$ and the corresponding code value v'. We can use the following equations to recover the interval and code value that we would have obtained without rescaling:

$$b_N = \frac{b'_N}{\gamma} + \delta, \quad l_N = \frac{l'_N}{\gamma}, \quad \hat{v} = \frac{v'}{\gamma} + \delta. \tag{1.32}$$

The decoder needs the original code value $\hat{v}$ to start recovering the data symbols. It should also rescale the interval at stage m, and thus needs to know m, δ, γ. Furthermore, when it scales the interval using (1.31), it must scale the code value as well, using

$$v' = \gamma\,(\hat{v} - \delta). \tag{1.33}$$

We can generalize the results above to rescaling at stages $m \le n \le \ldots \le p$. In general, the scaling process, including the scaling of the code values is

$$\begin{array}{lll} b'_m = \gamma_1\,(b_m - \delta_1), & l'_m = \gamma_1\, l_m, & v' = \gamma_1\,(\hat{v} - \delta_1), \\ b''_n = \gamma_2\,(b'_n - \delta_2), & l''_n = \gamma_2\, l'_n, & v'' = \gamma_2\,(v' - \delta_2), \\ \vdots & \vdots & \vdots \\ b_p^{(T)} = \gamma_T\,(b_p^{(T-1)} - \delta_T), & l_p^{(T)} = \gamma_T\, l_p^{(T-1)}, & v^{(T)} = \gamma_T\,(v^{(T-1)} - \delta_T). \end{array} \tag{1.34}$$

At the end of the coding process we have interval $\bar{\Phi}_N(S) = \left|\, \bar{b}_N,\, \bar{l}_N \,\right\rangle$ and code value $\bar{v}$. We recover original values using

$$\Phi_N(S) = |\, b_N,\, l_N \,\rangle = \left| \delta_1 + \frac{1}{\gamma_1}\left(\delta_2 + \frac{1}{\gamma_2}\left(\delta_3 + \frac{1}{\gamma_3}\left(\cdots\left(\delta_T + \frac{\bar{b}_N}{\gamma_T}\right)\right)\right)\right),\ \frac{\bar{l}_N}{\prod_{i=1}^{T}\gamma_i} \right\rangle, \tag{1.35}$$

and

$$\hat{v} = \delta_1 + \frac{1}{\gamma_1}\left(\delta_2 + \frac{1}{\gamma_2}\left(\delta_3 + \frac{1}{\gamma_3}\left(\cdots\left(\delta_T + \frac{\bar{v}}{\gamma_T}\right)\right)\right)\right). \tag{1.36}$$

These equations may look awfully complicated, but in some special cases they are quite easy to use. For instance, in Section 2.1 we show how to use scaling with $\delta_i \in \{0, 1/2\}$ and $\gamma_i \equiv 2$, and explain the connection between δ_i and the binary representation of b_N and $\hat{v}$. The next example shows another simple application of interval rescaling.

Example 6

◁ Figure 1.6 shows rescaling applied to Example 3. It is very similar to Figure 1.4, but instead of having just an enlarged view of small intervals, in Figure 1.6 the intervals also change. The rescaling parameters $\delta_1 = 0.74$ and $\gamma_1 = 10$ are used after coding two

symbols, and $\delta_2 = 0$ and $\gamma_2 = 25$ after coding two more symbols. The final interval is $\bar{\Phi}_6(S) = |\, 0.65,\ 0.05\,\rangle$, that corresponds to

$$\Phi_6(S) = \left|\, 0.74 + \frac{1}{10}\left(\frac{0.65}{25}\right),\ \frac{0.05}{10 \times 25}\,\right\rangle = |\, 0.7426,\ 0.0002\,\rangle\,,$$

and which is exactly the interval obtained in Example 3. ▷

1.6.5 Approximate Arithmetic

To understand how arithmetic coding can be implemented with fixed-precision we should note that the requirements for addition and for multiplication are quite different. We show that if we are willing to lose some compression efficiency, then we do not need exact multiplications. We use the double brackets ($[[\,\cdot\,]]$) around a multiplication to indicate that it is an approximation, i.e., $[[\alpha \cdot \beta]] \approx \alpha \cdot \beta$. We define truncation as any approximation such that $[[\alpha \cdot \beta]] \leq \alpha \cdot \beta$. The approximation we are considering here can be rounding or truncation to any precision. The following example shows an alternative way to interpret inexact multiplications.

Example 7

◁ We can see in Figure 1.3 that the arithmetic coding multiplications always occur with data from the source model—the probability p and the cumulative distribution c. Suppose we have $l = 0.04$, $c = 0.317$, and $p = 0.123$, with

$$\begin{aligned} l \times c &= 0.04 \times 0.317 = 0.01268, \\ l \times p &= 0.04 \times 0.123 = 0.00492. \end{aligned}$$

Instead of using exact multiplication we can use an approximation (e.g., with table look-up and short registers) such that

$$\begin{aligned} [[l \times c]] &= [[0.04 \times 0.317]] = 0.012, \\ [[l \times p]] &= [[0.04 \times 0.123]] = 0.0048. \end{aligned}$$

Now, suppose that instead of using p and c, we had used another model, with $c' = 0.3$ and $p' = 0.12$. We would have obtained

$$\begin{aligned} l \times c' &= 0.04 \times 0.3 = 0.012, \\ l \times p' &= 0.04 \times 0.12 = 0.0048, \end{aligned}$$

which are exactly the results with approximate multiplications. This shows that inexact multiplications are mathematically equivalent to making approximations in the source model and then using exact multiplications. ▷

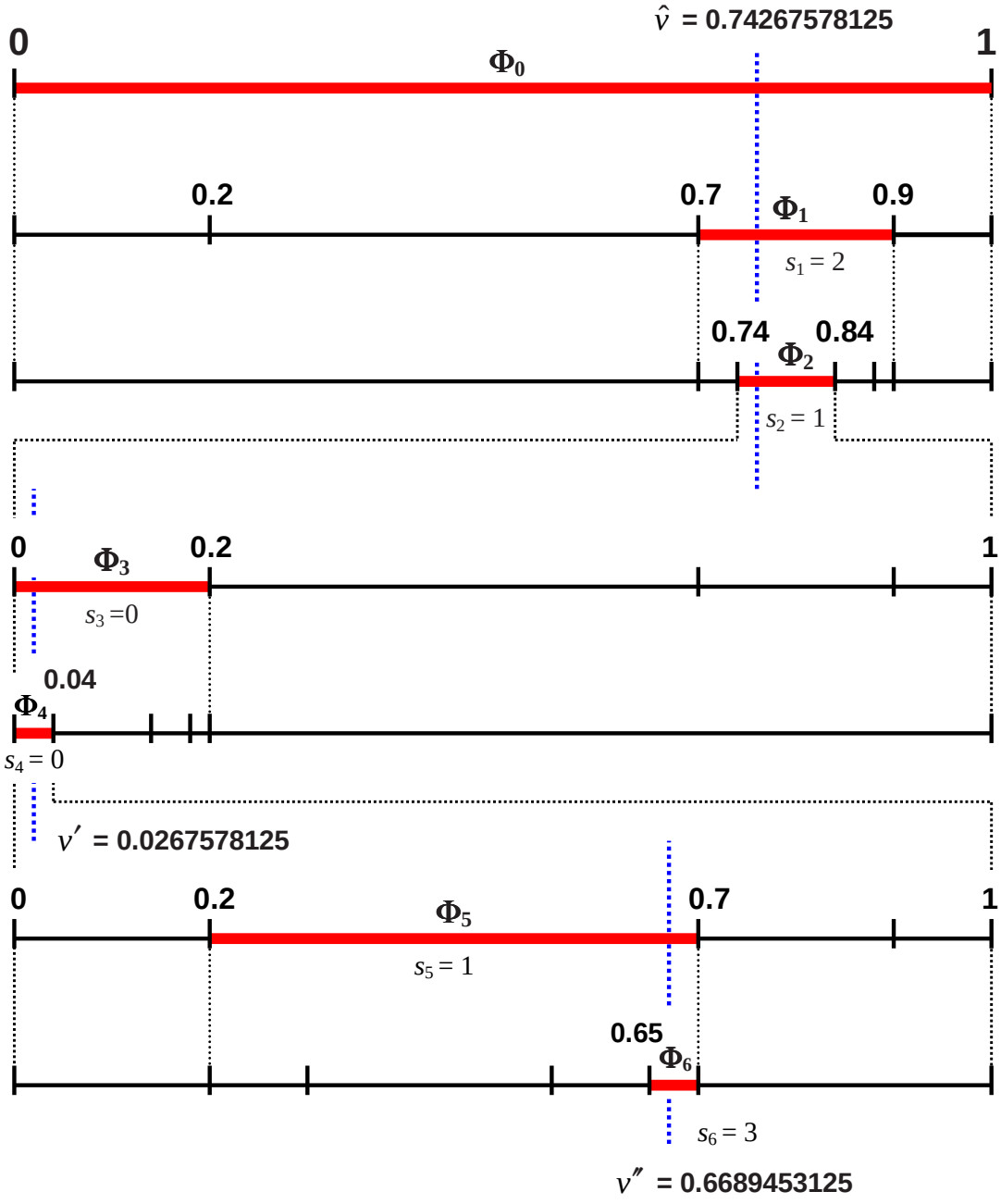

Figure 1.6: Graphical representation of the arithmetic coding process of Example 3 (Figure 1.4) using numerical rescaling. Note that the code value changes each time the intervals are rescaled.

What we have seen in this example is that whatever the approximation used for the multiplications we can always assume that exact multiplications occur all the time, but with inexact distributions. We do not have to worry about the exact distribution values as long as the decoder is synchronized with the encoder, i.e., if the decoder is making exactly the same approximations as the encoder, then the encoder and decoder distributions must be identical (just like having dynamic sources, as explained in Section 1.6.1).

The version of (1.9) with inexact multiplications is

$$\Phi_k(S) = |\, b_k,\, l_k \,\rangle = |\, b_{k-1} + [[c(s_k) \cdot l_{k-1}]],\, [[p(s_k) \cdot l_{k-1}]] \,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.37}$$

We must also replace (1.16) and (1.17) with

$$\hat{s}_k(\hat{v}) = \{s : b_{k-1} + [[c(s) \cdot l_{k-1}]] \le \hat{v} < b_{k-1} + [[c(s+1) \cdot l_{k-1}]]\}\,,\ k = 1, 2, \ldots, N, \tag{1.38}$$

$$\Phi_k(\hat{v}) = |\, b_k,\, l_k \,\rangle = |\, b_{k-1} + [[c(\hat{s}_k(\hat{v})) \cdot\ l_{k-1}]],\, [[p(\hat{s}_k(\hat{v})) \cdot l_{k-1}]] \,\rangle\,,\ k = 1, 2, \ldots, N. \tag{1.39}$$

In the next section we explain which conditions must be satisfied by the approximate multiplications to have correct decoding.

In equations (1.37) to (1.39) we have one type of approximation occurring from the multiplication of the interval length by the cumulative distribution, and another approximation resulting from the multiplication of the interval length by the probability. If we want to use only one type of approximation, and avoid multiplications between length and probability, we should update interval lengths according to

$$l_k = (b_{k-1} + [[c(s_k + 1) \cdot l_{k-1}]]) - (b_{k-1} + [[c(s_k) \cdot l_{k-1}]])\,. \tag{1.40}$$

The price to pay for inexact arithmetic is degraded compression performance. Arithmetic coding is optimal only as long as the source model probabilities are equal to the true data symbol probabilities; any difference reduces the compression ratios.

A quick analysis can give us an idea of how much can be lost. If we use a model with probability values $\mathbf{p}'$ in a source with probabilities $\mathbf{p}$, the average loss in compression is

$$\Delta = \sum_{n=0}^{M-1} p(n) \log_2 \left[\frac{p(n)}{p'(n)}\right], \quad \text{bits/symbol.} \tag{1.41}$$

This formula is similar to the relative entropy [32], but in this case $\mathbf{p}'$ represents the values that would result from the approximations, and it is possible to have $\sum_{n=0}^{M-1} p'(n) \neq 1$.

Assuming a relative multiplication error within ε, i.e.,

$$1 - \varepsilon \le \frac{p(n)}{p'(n)} \le 1 + \varepsilon, \tag{1.42}$$

we have

$$\Delta \le \sum_{n=0}^{M-1} p(n) \log_2(1 + \varepsilon) \approx \frac{\varepsilon}{\ln(2)} \approx 1.4\,\varepsilon \quad \text{bits/symbol.} \tag{1.43}$$

This is not a very tight bound, but it shows that if we can make multiplication accurately to, say 4 digits, the loss in compression performance can be reasonably small.

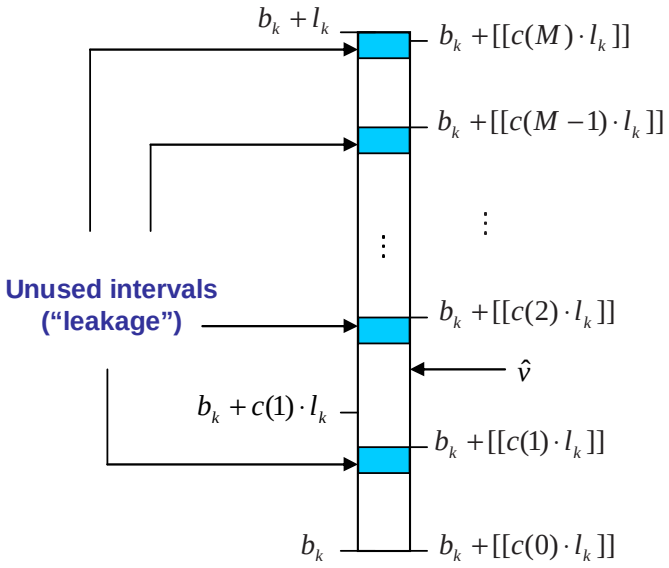

Figure 1.7: Subdivision of a coding interval with approximate multiplications. Due to the fixed-precision arithmetic, we can only guarantee that all coding intervals are disjoint if we leave small regions between intervals unused for coding.

1.6.6 Conditions for Correct Decoding

Figure 1.7 shows how an interval is subdivided when using inexact multiplications. In the figure we show that there can be a substantial difference between, say, $b_k + c(1) \cdot l_k$ and $b_k + [[c(1) \cdot l_k]]$, but this difference does not lead to decoding errors if the decoder uses the same approximation.

Decoding errors occur when condition (1.19) is not satisfied. Below we show the constraints that must be satisfied by approximations, and analyze the three main causes of coding error to be avoided.

(a) The interval length must be positive and intervals must be disjoint.
The constraints that guarantee that the intervals do not collapse into a single point, and that the interval length does not become larger than the allowed interval are

$$0 < l_{k+1} = [[p(s) \cdot l_k]] \leq (b_k + [[c(s+1) \cdot l_k]]) - (b_k + [[c(s) \cdot l_k]]), \; s = 0, 1, \ldots, M-1. \quad (1.44)$$

For example, if the approximations can create a situation in which $[[c(s+1) \cdot l_k]] < [[c(s) \cdot l_k]]$, there would be an non-empty intersection of subintervals assigned for $s+1$ and s, and decoder errors would occur whenever a code value belongs to the intersection.

If $[[c(s+1) \cdot l_k]] = [[c(s) \cdot l_k]]$ then the interval length collapses to zero, and stays as such, independently of the symbols coded next. The interval length may become zero due to arithmetic underflow, when both l_k and $p(s) = c(s+1) - c(s)$ are very small. In Section 2.1 we show that interval rescaling is normally used to keep l_k within a certain range to avoid

this problem, but we also have to be sure that all symbol probabilities are larger than a minimum value defined by the arithmetic precision (see Sections (2.5) and (A.1)).

Besides the conditions defined by (1.44), we also need to have

$$[[c(0) \cdot l_k]] \geq 0, \quad \text{and} \quad [[c(M) \cdot l_k]] \leq l_k. \tag{1.45}$$

These two condition are easier to satisfy because $c(0) \equiv 0$ and $c(M) \equiv 1$, and it is easy to make such multiplications exact.

(b) Sub-intervals must be nested.
We have to be sure that the accumulation of the approximation errors, as we continue coding symbols, does not move the interval base to a point outside all the previous intervals. With exact arithmetic, as we code new symbols, the interval base increases within the interval assigned to s_{k+1}, but it never crosses the boundary to the interval assigned to $s_{k+1}+1$, i.e.,

$$b_{k+n} = b_k + \sum_{i=k}^{k+n-1} c(s_{i+1}) \cdot l_i < b_k + c(s_{k+1}+1) \cdot l_k, \quad \text{for all} \quad n \geq 0. \tag{1.46}$$

The equivalent condition for approximate arithmetic is that for every data sequence we must have

$$b_k + [[c(s_{k+1}+1) \cdot l_k]] > b_k + [[c(s_{k+1}) \cdot l_k]] + \sum_{i=k+1}^{\infty} [[c(s_{i+1}) \cdot l_i]]. \tag{1.47}$$

To determine when (1.47) may be violated we have to assume some limits on the multiplication approximations. There should be a non-negative number ε such that

$$[[c(s_{i+1}) \cdot l_i]](1-\varepsilon) < c(s_{i+1}) \cdot l_i. \tag{1.48}$$

We can combine (1.40), (1.47) and (1.48) to obtain

$$(1-\varepsilon) \cdot l_k > \sum_{i=k+1}^{\infty} c(s_{i+1}) \cdot l_i, \tag{1.49}$$

which is equal to

$$1-\varepsilon > c(s_{k+2}) + p(s_{k+3})\left(c(s_{k+3}) + p(s_{k+3})\left(c(s_{k+4}) + p(s_{k+4})\left(\cdots\right)\right)\right). \tag{1.50}$$

To find the maximum for the right-hand side of (1.50) we only have to consider the case $s_{k+2} = s_{k+3} = \ldots = M-1$ to find

$$1-\varepsilon > c(M-1) + p(M-1)\left(c(M-1) + p(M-1)\left(c(M-1) + p(M-1)\left(\cdots\right)\right)\right), \tag{1.51}$$

which is equivalent to

$$1-\varepsilon > c(M-1) + p(M-1). \tag{1.52}$$

But we know from (1.3) that by definition $c(M-1)+p(M-1) \equiv 1$! The answer to this contradiction lies in the fact that with exact arithmetic we would have equality in (1.46)

only after an infinite number of symbols. With inexact arithmetic it is impossible to have semi-open intervals that are fully used and match perfectly, so we need to take some extra precautions to be sure that (1.47) is always satisfied. What equation (1.52) tells us is that we solve the problem if we artificially decrease the interval range assigned for $p(M-1)$. This is equivalent to setting aside small regions, indicated as gray areas in Figure 1.7, that are not used for coding, and serve as a "safety net."

This extra space can be intentionally added, for example, by replacing (1.40) with

$$l_k = (b_{k-1} + [[c(s_k+1)\cdot l_{k-1}]]) - (b_{k-1} + [[c(s_k)\cdot l_{k-1}]]) - \zeta \tag{1.53}$$

where $0 < \zeta \ll 1$ is chosen to guarantee correct coding and small compression loss.

The loss in compression caused by these unused subintervals is called "leakage" because a certain fraction of bits is "wasted" whenever a symbol is coded. This fraction is on average

$$\Delta_s = p(s)\log_2\left(\frac{p(s)}{p'(s)}\right) \quad \text{bits}, \tag{1.54}$$

where $p(s)/p'(s) > 1$ is the ratio between the symbol probability and the size of interval minus the unused region. With reasonable precision, leakage can be made extremely small. For instance, if $p(s)/p'(s) = 1.001$ (low precision) then leakage is less than 0.0015 bits/symbol.

(c) Inverse arithmetic operations must not produce error accumulation.
Note that in (1.38) we define decoding assuming only the additions and multiplications used by the encoder. We could have used

$$\hat{s}_k(\hat{v}) = \left\{ s : c(s) \le \left[\left[\frac{\hat{v} - b_{k-1}}{l_{k-1}}\right]\right] < c(s+1) \right\}, \quad k = 1, 2, \ldots, N. \tag{1.55}$$

However, this introduces approximate subtraction and division, which have to be consistent with the encoder's approximations. Here we cannot possibly cover all problems related to inverse operations, but we should say that the main point is to observe error accumulation.

For example, we can exploit the fact that in (1.16) decoding only uses the difference $\varpi_k \equiv \hat{v} - b_k$, and use the following recursions.

$$|\, \varpi_0,\, l_0 \,\rangle = |\, \hat{v},\, 1 \,\rangle, \tag{1.56}$$

$$\hat{s}_k = \{ s : [[c(s)\cdot l_{k-1}]] \le \varpi_k < [[c(s+1)\cdot l_{k-1}]] \}, \quad k = 1, 2, \ldots, N. \tag{1.57}$$

$$|\, \varpi_k,\, l_k \,\rangle = |\, \varpi_{k-1} - [[c(\hat{s}_k)\cdot l_{k-1}]],\, [[p(\hat{s}_k)\cdot l_{k-1}]] \,\rangle, \quad k = 1, 2, \ldots, N. \tag{1.58}$$

However, because we are using a sequence of subtractions in (1.58), this technique works with integer arithmetic implementations (see Appendix A), but it may not work with floating-point implementations because of error accumulation.

Chapter 2

Arithmetic Coding Implementation

In this second part, we present the practical implementations of arithmetic coding. We show how to exploit all the arithmetic coding properties presented in the previous sections and develop a version that works with fixed-precision arithmetic. First, we explain how to implement binary extended-precision additions that exploit the arithmetic coding properties, including the carry propagation process. Next, we present complete encoding and decoding algorithms based on an efficient and simple form of interval rescaling. We provide the description for both floating-point and integer arithmetic, and present some alternative ways of implementing the coding, including different scaling and carry propagation strategies. After covering the details of the coding process, we study the symbol probability estimation problem, and explain how to implement adaptive coding by integrating coding and source modeling. At the end, we analyze the computational complexity of arithmetic coding.

2.1 Coding with Fixed-Precision Arithmetic

Our first practical problem is that the number of digits (or bits) required to represent the interval length exactly grows when a symbol is coded. For example, if we had $p(0) = 0.99$ and we repeatedly code symbol 0, we would have

$$l_0 = 1, \quad l_1 = 0.99, \quad l_2 = 0.9801, \quad l_3 = 0.970299, \quad l_4 = 0.96059601, \quad \ldots$$

We solve this problem using the fact we do not need exact multiplications by the interval length (Section 1.6.5). Practical implementations use P-bit registers to store approximations of the mantissa of the interval length and the results of the multiplications. All bits with significance smaller than those in the register are assumed to be zero.

With the multiplication precision problem solved, we still have the problem of implementing the additions in (1.37) when there is a large difference between the magnitudes of the interval base and interval length. We show that rescaling solves the problem, simultaneously enabling exact addition, and reducing loss of multiplication accuracy. For a binary output, we can use rescaling in the form of (1.31), with $\delta \in \{0, 1/2\}$ and $\gamma = 2$ whenever the length of the interval is below $1/2$. Since the decoder needs to know the rescaling parameters, they are saved in the data buffer $\mathbf{d}$, using bits "0" or "1" to indicate whether $\delta = 0$ or $\delta = 1/2$.

Special case $\delta = 1$ and $\gamma = 1$, corresponding to a carry in the binary representation, is explained later.

To simplify the notation we represent the rescaled intervals simply as $|\, b,\, l\,\rangle$ (no subscripts), and the rescaled code value as v.

$$\begin{aligned} l &= 2^{t(l_k)} l_k, \\ b &= \mathrm{frac}(2^{t(l_k)} b_k) = 2^{t(l_k)} b_k - \left\lfloor 2^{t(l_k)} b_k \right\rfloor, \\ v &= \mathrm{frac}(2^{t(l_k)} \hat{v}), \end{aligned} \tag{2.1}$$

where $\mathrm{frac}(\cdot)$ is the fractional part of a number and

$$t(x) = \{n : 2^{-n-1} < x \le 2^{-n}\} = \lfloor -\log_2(x) \rfloor . \tag{2.2}$$

Note that under these conditions we have $b \in [0,\, 1)$ and $l \in (0.5,\, 1]$, and for that reason the rescaling process is called *renormalization*. Of course, l, b, and v change with k, but this new notation is more convenient to represent variables in algorithm descriptions or computer programs.

The binary representations of the interval base and length have the following structure:

$$\begin{array}{rcccccc} & & & & \overbrace{}^{(L=2^P l)} & & \\ l_k = & 0.0000\ldots00 & 0000\ldots00 & & 1aaa\ldots aa & 000000\ldots_2 & \\ b_k = & \underbrace{0.aaaa\ldots aa}_{\text{settled}} & \underbrace{0111\ldots11}_{\text{outstanding}} & & \underbrace{aaaa\ldots aa}_{\substack{(B=2^P b)\\ \text{active}}} & \underbrace{000000\ldots_2}_{\text{trailing zeros}} & \end{array} \tag{2.3}$$

where symbol a represents an arbitrary bit value.

We can see in (2.3) that there is "window" of P *active* bits, forming integers L and B, corresponding to the nonzero bits of l_k, and the renormalized length l. Because the value of l is truncated to P-bit precision, there is a set of trailing zeros that does not affect the additions. The bits to the left of the active bits are those that had been saved in the data buffer **d** during renormalization, and they are divided in two sets.

The first set to the left is the set of *outstanding* bits: those that can be changed due to a carry from the active bits when new symbols are encoded. The second is the set of bits that have been *settled*, i.e., they stay constant until the end of the encoding process. This happens because intervals are nested, i.e., the code value cannot exceed

$$b_{k+n} < b_k + l_k \le \underbrace{0.aaaa\ldots aa}_{\text{settled}} \quad \underbrace{1000\ldots00}_{\text{changed by carry}} \quad \underbrace{aaaa\ldots aa}_{\text{active}} \quad 00000\ldots_2 \tag{2.4}$$

This equation shows that only the outstanding bits may change due to a carry from the active bits. Furthermore, inequality (2.4) also shows that there can be only one carry that would change these bits. If there is a carry, or when it is found that there can be no carry, these bits become settled. For that reason, the set of outstanding bits always start with 0, and is possibly followed only by 1s. As new symbols are encoded, all sets move to the right.

Algorithm 1

Function Arithmetic_Encoder $(N, S, M, \mathbf{c}, \mathbf{d})$

1. set { $b \leftarrow 0$; $l \leftarrow 1$;
   $t \leftarrow 0$; } ⋆ *Initialize interval*
   ⋆ *and bit counter*
2. for $k = 1$ to N do ⋆ *Encode N data symbols*
   2.1. Interval_Update $(s_k, b, l, M, \mathbf{c})$; ⋆ *Update interval according to symbol*
   2.2. if $b \geq 1$ then ⋆ *Check for carry*
      2.2.1. set { $b \leftarrow b - 1$; ⋆ *Shift interval base*
      Propagate_Carry $(t, \mathbf{d})$; } ⋆ *Propagate carry on buffer*
   2.3. if $l \leq 0.5$ then ⋆ *If interval is small enough*
      2.3.1. Encoder_Renormalization $(b, l, t, \mathbf{d})$; ⋆ *then renormalize interval*
3. Code_Value_Selection $(b, t, \mathbf{d})$; ⋆ *Choose final code value*
4. return t. ⋆ *Return number of code bits*

2.1.1 Implementation with Buffer Carries

Combining all we have seen, we can present an encoding algorithm that works with fixed-precision arithmetic. Algorithm 1 shows a function Arithmetic_Encoder to encode a sequence S of N data symbols, following the notation of Section 1.2. This algorithm is very similar to the encoding process that we used in Section 1.4, but with a renormalization stage after each time a symbol is coded, and the settled and outstanding bits being saved in the buffer $\mathbf{d}$. The function returns the number of bits used to compress S.

In Algorithm 1, Step 1 sets the initial interval equal to [0, 1), and initializes the bit counter t to zero. Note that we use curly braces ({ }) to enclose a set of assignments, and use symbol "⋆" before comments. In Step 2, we have the sequential repetition of interval resizing and renormalizations. Immediately after updating the interval we find out if there is a carry, i.e., if $b \geq 1$, and next we check if further renormalization is necessary. The encoding process finishes in Step 3, when the final code value v that minimizes the number of code bits is chosen. In all our algorithms, we assume that functions receive references, i.e., variables can be changed inside the function called. Below we describe each of the functions used by Algorithm 1.

There are many mathematically equivalent ways of updating the interval $|\, b,\, l\,\rangle$. We do not need to have both vectors $\mathbf{p}$ and $\mathbf{c}$ stored to use (1.9). In Algorithm 2 we use (1.40) to update length as a difference, and we avoid multiplication for the last symbol ($s = M - 1$), since it is more efficient to do the same at the decoder. To simplify notation, we do not use double brackets to indicate inexact multiplications, but it should be clear that here all numbers represent the content of CPU registers.

In Step 2.2.1 of Algorithm 1, the function to propagate the carry in the buffer $\mathbf{d}$ is called, changing bits that have been added to $\mathbf{d}$ previously, and we shift the interval to have $b < 1$. Figure 2.1 shows the carry propagation process. Active bits are shown in bold and outstanding bits are underlined. Whenever there is a carry, starting from the most recent bits added to buffer $\mathbf{d}$, we complement all bits until the first 0-bit is complemented, as in

Algorithm 2

Procedure Interval_Update $(s, b, l, M, \mathbf{c})$

1. if $s = M - 1$ ⋆ *Special case for last symbol*
   then set $y \leftarrow b + l$; ⋆ *end of interval*
   else set $y \leftarrow b + l \cdot c(s+1)$; ⋆ *base of next subinterval*
2. set $\{\, b \leftarrow b + l \cdot c(s)$; ⋆ *Update interval base*
   $l \leftarrow y - b;\ \}$ ⋆ *Update interval length as difference*
3. return.

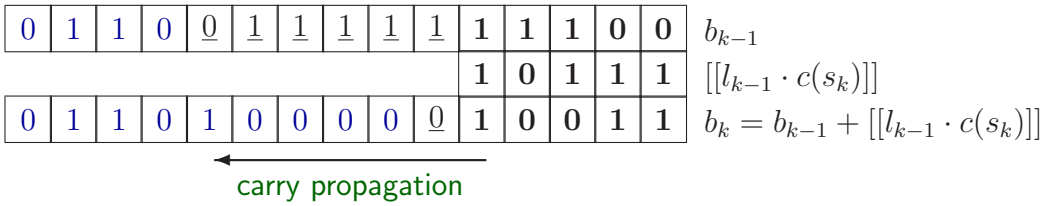

Figure 2.1: Carry propagation process. Bold letters indicate the active bits, outstanding bits are underlined, and leftmost bits are settled.

Algorithm 3.

Algorithm 4 implements interval renormalization, where we test if active bits became outstanding or settled. While the interval length is smaller than 0.5, the interval is rescaled by a factor of two, and a bit is added to the bit buffer **d**.

The final encoding stage is the selection of the code value. We use basically the same process explained at the end of Section 1.4.1, but here we choose a code value belonging to the rescaled interval. Our choice is made easy because we know that, after renormalization, we always have $0.5 < l \leq 1$ (see (2.1)), meaning that we only need an extra bit to define the final code value. In other words: all bits that define the code value are already in buffer **d**, and we only need to choose one more bit. The only two choices to consider in the rescaled interval are $v = 0.5$ or $v = 1$.

The decoding procedure, shown in Algorithm 6, gets as input the number of compressed symbols, N, the number of data symbols, M, and their cumulative distribution **c**, and the array with the compressed data bits, **d**. Its output is the recovered data sequence $\hat{S}$. The decoder must keep the P-bit register with the code value updated, so it will read P extra bits at the end of **d**. We assume that this can be done without problems, and that those bits had been set to zero.

The interval selection is basically an implementation of (1.38): we want to find the subinterval that contains the code value v. The implementation that we show in Algorithm 7 has one small shortcut: it combines the symbol decoding with interval updating (1.39) in a single function. We do a sequential search, starting from the last symbol ($s = M - 1$), because we assume that symbols are sorted by increasing probability. The advantages of

Algorithm 3

Procedure Propagate_Carry $(t, \mathbf{d})$

1. set $n \leftarrow t$; ⋆ *Initialize pointer to last outstanding bit*
2. while $d(n) = 1$ do ⋆ *While carry propagation*
   2.1. set { $d(n) \leftarrow 0$; ⋆ *complement outstanding 1-bit and*
   $n \leftarrow n - 1$; } ⋆ *move to previous bit*
3. set $d(n) \leftarrow 1$; ⋆ *Complement outstanding 0-bit*
4. return.

•

Algorithm 4

Procedure Encoder_Renormalization $(b, l, t, \mathbf{d})$

1. while $l \leq 0.5$ do ⋆ *Renormalization loop*
   1.1. set { $t \leftarrow t + 1$; ⋆ *Increment bit counter and*
   $l \leftarrow 2l$; } ⋆ *scale interval length*
   1.2. if $b \geq 0.5$ ⋆ *Test most significant bit of interval base*
   then set { $d(t) \leftarrow 1$; ⋆ *Output bit 1*
   $b \leftarrow 2(b - 0.5)$; } ⋆ *shift and scale interval base*
   else set { $d(t) \leftarrow 0$; ⋆ *Output bit 0*
   $b \leftarrow 2b$; } ⋆ *scale interval base*
2. return.

•

Algorithm 5

Procedure Code_Value_Selection $(b, t, \mathbf{d})$

1. set $t \leftarrow t + 1$; ⋆ *Increment bit counter*
2. if $b \leq 0.5$ ⋆ *Eenormalized code value selection*
   then set $d(t) \leftarrow 1$; ⋆ *Choose $v = 0.5$: output bit 1*
   else set { $d(t) \leftarrow 0$; ⋆ *Choose $v = 1.0$: output bit 0 and*
   Propagate_Carry $(t - 1, \mathbf{d})$; } ⋆ *propagate carry*
3. return.

•

Algorithm 6

Procedure Arithmetic_Decoder $(N, M, \mathbf{c}, \mathbf{d}, \hat{S})$

1. set { $b \leftarrow 0$; $l \leftarrow 1$;
   $v = \sum_{n=1}^{P} 2^{-n} d(n)$;
   $t \leftarrow P$; } ⋆ *Initialize interval* ⋆ *Read P bits of code value* ⋆ *Initialize bit counter*
2. for $k = 1$ to N do ⋆ *Decode N data symbols*
   2.1. $\hat{s}_k$ = Interval_Selection $(v, b, l, M, \mathbf{c})$; ⋆ *Decode symbol and update interval*
   2.2. if $b \geq 1$ then ⋆ *Check for "overflow"*
      2.2.1. set { $b \leftarrow b - 1$; ⋆ *shift interval base*
      $v \leftarrow v - 1$; } ⋆ *shift code value*
   2.3. if $l \leq 0.5$ then ⋆ *If interval is small enough*
      2.3.1. Decoder_Renormalization $(v, b, l, t, \mathbf{d})$; ⋆ *then renormalize interval*
3. return.

•

Algorithm 7

Function Interval_Selection $(v, b, l, M, \mathbf{c})$

1. set { $s \leftarrow M - 1$;
   $x \leftarrow b + l \cdot c(M-1)$;
   $y \leftarrow b + l$; } ⋆ *Start search from last symbol* ⋆ *Base of search interval* ⋆ *End of search interval*
2. while $x > v$ do ⋆ *Sequential search for correct interval*
   2.1. set { $s \leftarrow s - 1$; ⋆ *Decrement symbol by one*
   $y \leftarrow x$; ⋆ *move interval end*
   $x \leftarrow b + l \cdot c(s)$; } ⋆ *compute new interval base*
3. set { $b \leftarrow x$; ⋆ *Update interval base*
   $l \leftarrow y - b$; } ⋆ *Update interval length as difference*
4. return s.

•

sorting symbols and more efficient searches are explained in Section 2.2.

In Algorithm 7 we use only arithmetic operations that are exactly equal to those used by the encoder. This way we can easily guarantee that the encoder and decoder approximations are exactly the same. Several simplifications can be used to reduce the number of arithmetic operations (see Appendix A).

The renormalization in Algorithm 8 is similar to Algorithm 4, and in fact all its decisions (comparisons) are meant to be based on exactly the same values used by the encoder. However, it also needs to rescale the code value in the same manner as the interval base (compare (1.35) and (1.36)), and it reads its least significant bit (with value 2^{-P}).

Example 8

◁ We applied Algorithms 1 to 8 to the source and data sequence of Example 3, and the results are shown in Table 2.1. The first column shows what event caused the change in the interval. If a new symbol σ is coded, then we show it as $s = \sigma$, and show

Algorithm 8

Procedure Decoder_Renormalization $(v, b, l, t, \mathbf{d})$

1. while $l \leq 0.5$ do
   1.1. if $b \geq 0.5$
   then set { $b \leftarrow 2(b - 0.5)$;
   $v \leftarrow 2(v - 0.5)$; }
   else set { $b \leftarrow 2b$;
   $v \leftarrow 2v$; }
   1.2. set { $t \leftarrow t + 1$;
   $v \leftarrow v + 2^{-P} d(t)$;
   $l \leftarrow 2l$; }
2. return.

⋆ *Renormalization loop*
⋆ *Remove most significant bit*
⋆ *shift and scale interval base*
⋆ *shift and scale code value*
⋆ *scale interval base*
⋆ *scale code value*
⋆ *Increment bit counter*
⋆ *Set least significant bit of code value*
⋆ *Scale interval length*

•

the results of Algorithm 2. We indicate the interval changes during renormalization (Algorithm 4) by showing the value of δ used for rescaling, according to (1.31).

Comparing these results with those in Table 1.2, we see how renormalization keeps all numerical values within a range that maximizes numerical accuracy. In fact, the results in Table 2.1 are exact, and can be shown with a few significant digits. Table 2.1 also shows the decoder's updated code value. Again, these results are exact and agree with the results shown in Table 1.2. The third column in Table 2.1 shows the contents of the bit buffer $\mathbf{d}$, and the bits that are added to this buffer every time the interval is rescaled. Note that carry propagation occurs twice: when $s = 3$, and when the final code value $v = 1$ is chosen by Algorithm 5. ▷

2.1.2 Implementation with Integer Arithmetic

Even though we use real numbers to describe the principles of arithmetic coding, most practical implementations use only integer arithmetic. The adaptation is quite simple, as we just have to assume that the P-bit integers contain the fractional part of the real numbers, with the following adaptations. (Appendix A has the details.)

- Define $B = 2^P b$, $L = 2^P l$, $V = 2^P v$, and $C(s) = 2^P c(s)$. Products can be computed with $2P$ bits, and the P least-significant bits are discarded. For example, when updating the interval length we compute $L \leftarrow \left\lfloor L \cdot [C(s+1) - C(s)] \cdot 2^{-P} \right\rfloor$. The length value $l = 1$ cannot be represented in this form, but this is not a real problem. We only need to initialize the scaled length with $L \leftarrow 2^P - 1$, and apply renormalization only when $l < 0.5$ (strict inequality).

- The carry condition $b \geq 1$ is equivalent to $B \geq 2^P$, which can mean integer overflow. It can be detected accessing the CPU's carry flag, or equivalently, checking when the value of B decreases.

- Since $l > 0$ we can work with a scaled length equal to $L' = 2^P l - 1$. This way we can represent the value $l = 1$ and have some extra precision if P is small. On the other

Event	**Scaled base** b	**Scaled length** l	**Bit buffer** **d**	**Scaled code value** v	**Normalized code value** $(v-b)/l$
—	0	1		0.74267578125	0.74267578125
$s=2$	0.7	0.2		0.74267578125	
$\delta=0.5$	0.4	0.4	1	0.4853515625	
$\delta=0$	0.8	0.8	10	0.970703125	0.21337890625
$s=1$	0.96	0.4	10	0.970703125	
$\delta=0.5$	0.92	0.8	101	0.94140625	0.0267578125
$s=0$	0.92	0.16	101	0.94140625	
$\delta=0.5$	0.84	0.32	1011	0.8828125	
$\delta=0.5$	0.68	0.64	10111	0.765625	0.1337890625
$s=0$	0.68	0.128	10111	0.765625	
$\delta=0.5$	0.36	0.256	101111	0.53125	
$\delta=0$	0.72	0.512	1011110	1.0625	0.6689453125
$s=1$	0.8224	0.256	1011110	1.0625	
$\delta=0.5$	0.6448	0.512	10111101	1.125	0.937890625
$s=3$	1.1056	0.0512	10111101	1.125	
$\delta=1$	0.1056	0.0512	10111110	0.125	
$\delta=0$	0.2112	0.1024	101111100	0.25	
$\delta=0$	0.4224	0.2048	1011111000	0.5	
$\delta=0$	0.8448	0.4096	10111110000	1	
$\delta=0.5$	0.6896	0.8192	101111100001	1	
$v=1$	1	—	101111100001	—	
$\delta=1$	0	—	101111100010	—	
$\delta=0$	0	—	1011111000100	—	

Table 2.1: Results of arithmetic encoding and decoding, with renormalization, applied to source and data sequence of Example 3. Final code value is $\hat{v} = 0.1011111000100_2 = 0.74267578125$.

hand, updating the length using $L' \leftarrow \left\lfloor (L'+1)\cdot[C(s+1)-C(s)]\cdot 2^{-P}-1 \right\rfloor$ requires two more additions.

When multiplication is computed with $2P$ bits, we can determine what is the smallest allowable probability to avoid length underflow. Since renormalization guarantees that $L \geq 2^{P-1}$, we have (1.44) satisfied if

$$\left\lfloor [C(s+1)-C(s)]\, 2^{-P}\, L \right\rfloor \geq \left\lfloor [C(s+1)-C(s)]\, 2^{-1} \right\rfloor \geq 1 \Rightarrow C(s+1)-C(s) \geq 2, \quad (2.5)$$

Meaning that the minimum probability supported by such implementations with P-bit integers is $p(s) \geq 2^{1-P}$.

Input s_k	Encoder Data	Binary Representation	Decimal Representation	8-bit Registers
—	b_o	$0.\mathbf{00000000}_2$	0	00000000
	l_0	$\mathbf{1.00000000}_2$	1	11111111
2	$l_0 \cdot c(2)$	$0.\mathbf{10110011}_2$	0.69921875	10110011
	b_1	$0.1\underline{0}\mathbf{11001100}_2$	0.69921875	11001100
	l_1	$0.00\mathbf{11001100}_2$	0.19921875	11001011
1	$l_1 \cdot c(1)$	$0.00\mathbf{00101000}_2$	0.0390625	00101000
	b_2	$0.1\underline{01}\mathbf{11101000}_2$	0.73828125	11101000
	l_2	$0.000\mathbf{11001100}_2$	0.099609375	11001011
0	$l_2 \cdot c(0)$	$0.000\mathbf{00000000}_2$	0	00000000
	b_3	$0.1\underline{0111}\mathbf{10100000}_2$	0.73828125	10100000
	l_3	$0.00000\mathbf{10100000}_2$	0.01953125	10011111
0	$l_3 \cdot c(0)$	$0.00000\mathbf{00000000}_2$	0	00000000
	b_4	$0.1011110\underline{1}\mathbf{00000000}_2$	0.73828125	00000000
	l_4	$0.0000000\mathbf{11111000}_2$	0.0037841796875	11110111
1	$l_4 \cdot c(1)$	$0.0000000\mathbf{00110001}_2$	0.0007476806640625	00110001
	b_5	$0.1011110\underline{1}\mathbf{01100010}_2$	0.7390289306640625	01100010
	l_5	$0.0000000\mathbf{11111000}_2$	0.00189208984375	11110111
3	$l_5 \cdot c(3)$	$0.0000000\mathbf{11011110}_2$	0.0016937255859375	11011110
	b_6	$0.10111101101\underline{0}\mathbf{00000000}_2$	0.74072265625	00000000
	l_6	$0.000000000000\mathbf{11001000}_2$	0.00019073486328125	11000111
—	$\hat{v}$	0.1011110110101_2	0.7408447265625	—

Table 2.2: Results of arithmetic encoding and decoding for Example 3 using 8-bit precision for the arithmetic operations.

Example 9

◁ Table 2.2 shows the results of an 8-bit register implementation ($P = 8$) applied to the data sequence and source used in Example 3. The table shows the interval $|\, b_k,\ l_k \,\rangle$ in base-2 notation to make clear that even though we use 8-bit arithmetic, we are actually implementing *exact* additions. Following the conventions in (2.3) and Figure 2.1, we used bold letters to indicate active bits, and underlined for outstanding bits. We also show the approximate results of the multiplications $l_{k-1} \cdot c(s_k)$. The last column shows the binary contents of the registers with active bits. We used $L' = 2^P l - 1$ to represent the length. The results are also shown in decimal notation so that they can be compared with the exact results in Table 1.2. Note that the approximations change the code value after only 7 bits, but the number of bits required to represent the final code value is still 13 bits. ▷

Algorithm 9

Procedure Encoder_Renormalization $(b, l, t, \mathbf{d})$

1. while $l \leq 1/D$ do
   1.1. set { $t \leftarrow t + 1$; } ⋆ *Renormalization loop*
   ⋆ *Increment symbol counter*
   $d(t) \leftarrow \lfloor D \cdot b \rfloor$; ⋆ *Output symbol from most significant bits*
   $b \leftarrow D \cdot b - d(t)$; ⋆ *Update interval base*
   $l \leftarrow D \cdot l$; } ⋆ *Scale interval length*
2. return.

•

2.1.3 Efficient Output

Implementations with short registers (as in Example 9) require renormalizing the intervals as soon as possible to avoid losing accuracy and compression efficacy. We can see in Table 2.1 that, as a consequence, intervals may be rescaled many times whenever a symbol is coded. Even though rescaling can be done with bit shifts instead of multiplications, this process still requires many CPU cycles (see Section 2.3).

If we use longer registers (e.g., 16 bits or more), we can increase efficiency significantly by moving more than one bit to the output buffer $\mathbf{d}$ whenever rescaling. This process is equivalent to have an encoder output alphabet with D symbols, where D is a power of two. For example, moving groups of 1, 2, 4, or 8 bits at a time, corresponds to output alphabets with $D = 2$, 4, 16, and 256 symbols, respectively. Carry propagation and the use of the output buffer $\mathbf{d}$ also become more efficient with larger D.

It is not difficult to modify the algorithms in Section 2.1.1 for a D-symbol output. Renormalization is the most important change. The rescaling parameters defined by (1.31) are such that $\delta \in \{0, 1/D, 2/D, \ldots, 1\}$ and $\gamma \equiv D$. Again, $\delta = 1$ corresponds to a carry. Algorithm 9 has the required changes to Algorithm 4, and the corresponding changes in the decoder renormalization are shown in Algorithm 10. In Appendix A we have an integer arithmetic implementation with D-symbol output, with the carry propagation in Algorithm 25.

We also need to change the equations that define the normalized intervals, (2.1) and (2.2), to

$$\begin{aligned} l &= D^{t(l_k)} l_k, \\ b &= \text{frac}(D^{t(l_k)} b_k) = D^{t(l_k)} b_k - \left\lfloor D^{t(l_k)} b_k \right\rfloor, \\ v &= \text{frac}(D^{t(l_k)} \hat{v}), \end{aligned} \tag{2.6}$$

and

$$t(x) = \{n : \; D^{-n-1} < x \leq D^{-n}\} = \lfloor -\log_D(x) \rfloor \,. \tag{2.7}$$

Example 10

◁ Table 2.3 shows an example of how the arithmetic coding renormalization changes when $D = 16$. Again, the source and data sequence are those of Example 3. Comparing these results with those of Example 8 (Table 2.1), we can see the substantial reduction in the number of times the intervals are rescaled. ▷

Algorithm 10

Procedure Decoder_Renormalization $(v, b, l, t, \mathbf{d})$

1. while $l \leq 1/D$ do ⋆ *Renormalization loop*
   1.1. set { $t \leftarrow t + 1$; } ⋆ *Increment symbol counter*
   $a \leftarrow \lfloor D \cdot b \rfloor$; ⋆ *Most significant digit*
   $b \leftarrow D \cdot b - a$; ⋆ *Update interval base*
   $v \leftarrow D \cdot v - a + D^{-P} d(t)$; ⋆ *Update code value*
   $l \leftarrow D \cdot l$; } ⋆ *Scale interval length*
3. return.

•

Event	Scaled base b	Scaled length l	Bit buffer **d**	Scaled code value v
—	0	1		0.74267578125
$s = 2$	0.7	0.2		0.74267578125
$s = 1$	0.74	0.1		0.74267578125
$s = 0$	0.74	0.02		0.74267578125
$\delta = 11/16$	0.84	0.32	B	0.8828125
$s = 0$	0.84	0.064	B	0.8828125
$s = 1$	0.8528	0.032	B	0.8828125
$\delta = 13/16$	0.6448	0.512	BD	1.125
$s = 3$	1.1056	0.0512	BD	1.125
$\delta = 1$	0.1056	0.0512	BE	0.125
$\delta = 1/16$	0.6896	0.8192	BE1	1
$v = 1$	1	—	BE1	—
$\delta = 1$	0	—	BE2	—
$\delta = 0$	—	—	BE20	—

Table 2.3: Results of arithmetic encoding and decoding for Example 3, with renormalization and a 16-symbol (hexadecimal) output alphabet. Final code value is $\hat{v} = 0.\text{BE20}_{16} = 0.1011\,1110\,0010\,0000_2 = 0.74267578125$.

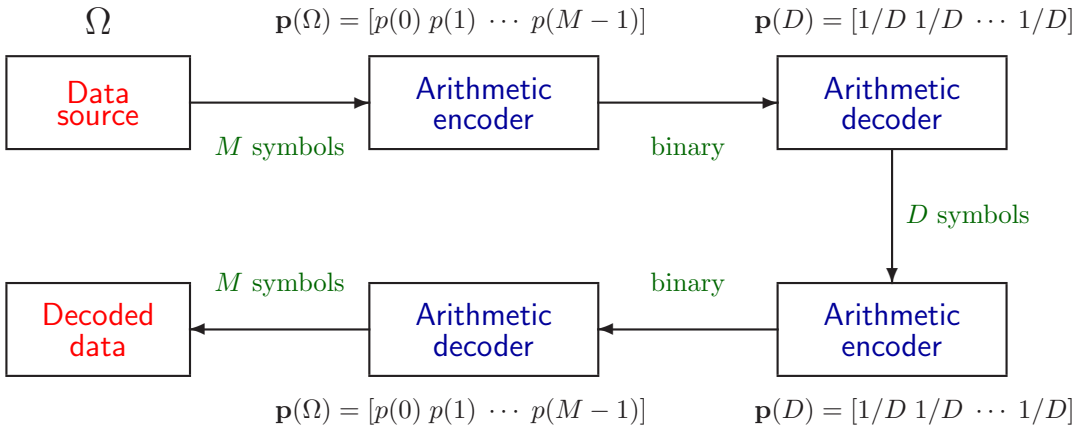

Figure 2.2: Configuration for using a standard (binary output) arithmetic encoder to replace an encoder with D-symbol (ternary, decimal, etc.) output.

Algorithms 9 and 10 can also be used for values of D that are not powers of two, but with inefficient radix-D arithmetic. The problem is that while the multiplications by the interval length can be approximated, rescaling has to be exact. For example, if $D = 2$ then multiplication by 2^{-P} is computed exactly with bit-shifts, but exact multiplication by 3^{-P} requires special functions [39].

A better alternative is to use a small trick, shown in Figure 2.2. We can compress the data sequence using a standard encoder/decoder with binary arithmetic. Next, we "decode" the binary compressed data using a D-symbol alphabet with uniform probability distribution $\mathbf{p} = [\,1/D \quad 1/D \quad \cdots \quad 1/D\,]$. The uniform distribution does not change the distribution of the code values (and thus will not alter compression), but converts the data to the desired alphabet. Since both processes implemented by the decoder are perfectly reversible, the decoder only has to implement the inverse processes. This takes twice the time for encoding and decoding, but is significantly faster than using radix-D arithmetic.

2.1.4 Care with Carries

We have seen in Section 2.1 that carry propagation is applied only to the set of outstanding bits, which always start with a 0-bit, and is followed possibly by 1-bits. Examples of set of outstanding bits are

$$\emptyset, \{0\}, \{0,1\}, \{0,1,1\}, \{0,1,1,1\}, \cdots, \{0,1,1,1,\ldots,1\}.$$

Clearly, with such simple structure we do not have to save the outstanding bits to know what they are. We can just keep a counter with the number of outstanding bits, which is incremented as new bits are defined during renormalization. We can output (or "flush") these bits whenever a carry occurs or when a new outstanding 0-bit comes from renormalization.

Note that not all bits put out by rescaling have to be outstanding before becoming settled. For example, if we have $b < b + l \leq 0.5$, we not only know that the next bit is zero, but we know that it cannot possibly be changed by a carry, and is thus settled. We can disregard these details when implementing carry propagation in buffers, but not when using counters.

With D-symbol output the set of outstanding symbols starts with a symbol $\sigma \neq D-1$, and is possibly followed by several occurrences of the symbol $D-1$, as shown below.

$$\emptyset, \{\sigma\}, \{\sigma, D-1\}, \{\sigma, D-1, D-1\}, \cdots, \{\sigma, D-1, D-1, \ldots, D-1\}.$$

In this case we can only keep the first symbol σ and a counter with the number of outstanding symbols.

It is important to know that the number of outstanding symbols can grow indefinitely, i.e., we can create distributions and infinitely long data sequences such that bits never become settled. The final code value selection (Algorithm 5) settles all outstanding symbols, so that we can periodically restart the encoder to limit the number of outstanding symbols.

There are many choices for dealing with carry propagation. The most common are:

1. Save the outstanding symbols temporarily to a buffer, and then implement carry propagation in the buffer. This simple technique is efficient when working with bytes ($D = 256$), but can be inefficient when D is small. It can only be used if we know that the buffer is large enough to fit all the compressed symbols, since all of them can be outstanding.

2. Use a counter to keep track of the outstanding symbols, saving all symbols to a buffer or file as soon as they become settled. There is chance of overflow depending on the number of bits used by the counter (mantissas with 32, 53, and 64 bits allow, respectively, $4 \cdot 10^9$, $9 \cdot 10^{15}$, and $2 \cdot 10^{19}$ outstanding symbols). In practical applications a counter overflow is extremely improbable, specially with adaptive coders.

3. Use carry propagation in a buffer, plus "bit-stuffing" [15, 27] in the following form. Outstanding bits are moved out from the buffer as soon as they become settled. Whenever the number of 1-bits exceeds a threshold (e.g., 16), an artificial zero is added to the compressed bit sequence, forcing the outstanding bits to become settled. The decoder can identify when this happens by comparing the number of consecutive 1-bits read. When it exceeds the threshold, the decoder interprets the next bit not as data, but as carry information. If it is 1, then a carry is propagated in the decoder buffer, and the decoding process continues normally. This technique is used by the Q-coder [27].

2.1.5 Alternative Renormalizations

We show in Section 1.6.4 that there is a great deal of flexibility in rescaling the intervals, and we present in Section 2.1.1 an implementation based on a particular form of renormalization. Other choices lead to renormalized intervals with distinct properties, which had been exploited in several manners by different arithmetic coding implementations. Below we show a few examples.

- We have chosen renormalization (2.6), which produces intervals such that $b \in [0,\ 1)$, $l \in (1/D,\ 1]$, and $b + l \in (1/D,\ 2)$. Its main advantage is that it simplifies carry detection and renormalization, especially when $D > 2$. Note that it has a criterion for when to renormalize that is based only on the interval length.

- The decision of when to renormalize can be based on settled symbols. For example, the method by Rubin [13] keeps intervals such that $b \in [0, 1)$, and $b + l \in (0, 1]$, which are renormalized when the most significant symbol becomes settled, i.e., $\lfloor Db \rfloor = \lfloor D(b + l) \rfloor$, meaning that the outstanding symbols are kept in the registers. To avoid the interval length eventually collapsing to zero, the encoder and decoder rescale and shift the interval when its length gets too small, forcing outstanding symbols to become settled.

- Witten, Neal, and Cleary [25, 51] proposed an arithmetic coding implementation that became quite popular. Instead of the base length, it uses the extremes points to represent the interval, and keeps it renormalized in such a manner that $b + l \in (0, 1]$. Renormalization occurs whenever bits are settled, and also when $0.25 \leq b < b + l \leq 0.75$. Thus, it avoids the precision loss that occurs in [13], and uses counters to keep track of the number of outstanding bits. This technique can be adapted to D-symbol output, but it is not as simple as what we have in Appendix A.

- The binary Q-coder developed by IBM [27, 28], keeps intervals with $l \in (0.75, 1.5]$. This way we normally have $l \approx 1$, and we can approximate multiplications with $p \cdot l \approx p$, and $(1 - p) \cdot l \approx l - p$. Variations include the QM [36] and MQ-coder [62].

2.2 Adaptive Coding

Since typical information sources tend to be quite complex, we must have a good model of the data source to achieve optimal compression. There are many techniques for modeling complex sources [14, 25, 29, 44, 45, 46, 49] that decompose the source data in different categories, under the assumption that in each category the source symbols are approximately independent and identically distributed, and thus well suited to be compressed with arithmetic coding. In general, we do not know the probabilities of the symbols in each category. Adaptive coding is the estimation of the probabilities of the source symbols during the coding process. In this section we study techniques to efficiently combine arithmetic coding with dynamic probability estimation.

2.2.1 Strategies for Computing Symbol Distributions

The most efficient technique for computing distributions depends on the data type. When we are dealing with completely unknown data we may want adaptation to work in a completely automatic manner. In other cases, we can use some knowledge of the data properties to reduce or eliminate the adaptation effort. Below we explain the features of some of the most common strategies for estimating distributions.

- Use a constant distribution that is available before encoding and decoding, normally estimated by gathering statistics in a large number of typical samples. This approach can be used for sources such as English text, or weather data, but it rarely yields the best results because few information sources are so simple as to be modeled by a single distribution. Furthermore, there is very little flexibility (e.g., statistics for English text

do not fit well Spanish text). On the other hand, it may work well if the source model is very detailed, and in fact it is the only alternative in some very complex models in which meaningful statistics can only be gathered from a very large amount of data.

- Use pre-defined distributions with adaptive parameter estimation. For instance, we can assume that the data has Gaussian distribution, and estimate only the mean and variance of each symbol. If we allow only a few values for the distribution parameters, then the encoder and decoder can create several vectors with all the distribution values, and use them according to their common parameter estimation. See ref. [49] for an example.

- Use two-pass encoding. A first pass gathers the statistics of the source, and the second pass codes the data with the collected statistics. For decoding, a scaled version of vectors **p** or **c** must be included at the beginning of the compressed data. For example, a book can be archived (compressed) together with its particular symbol statistics. It is possible to reduce the computational overhead by sharing processes between passes. For example, the first pass can simultaneously gather statistics and convert the data to run-lengths.

- Use a distribution based on the occurrence of symbols previously coded, updating **c** with each symbol encoded. We can start with a very approximate distribution (e.g., uniform), and if the probabilities change frequently, we can reset the estimates periodically. This technique, explained in the next section, is quite effective and the most convenient and versatile. However, the constant update of the cumulative distribution can increase the computational complexity considerably. An alternative is to update only the probability vector **p** after each encoded symbol, and update the cumulative distribution **c** less frequently (Section 2.2.5).

2.2.2 Direct Update of Cumulative Distributions

After encoding/decoding k symbols the encoder/decoder can estimate the probability of a symbol as

$$p(m) = \frac{\tilde{P}(m)}{k+M}, \quad m = 0, 1, 2, \ldots, M-1, \tag{2.8}$$

where $\tilde{P}(m) > 0$ is the number of times symbol m was encoded/decoded plus one (added to avoid zero probabilities). The symbol occurrence counters are initialized with $\tilde{P}(m) = 1$, and incremented after a symbol is encoded. We define the cumulative sum of occurrences as

$$\tilde{C}(m) = \sum_{i=0}^{m-1} \tilde{P}(i), \quad m = 0, 1, 2, \ldots, M, \tag{2.9}$$

and the cumulative distribution as

$$c(m) = \frac{\tilde{C}(m)}{k+M} = \frac{\tilde{C}(m)}{\tilde{C}(M)}, \quad m = 0, 1, 2, \ldots, M \tag{2.10}$$

Algorithm 11

Procedure Interval_Update $(s, b, l, M, \tilde{\mathbf{C}})$

1. set $\gamma = l/\tilde{C}(M)$ ⋆ *Compute division result*
2. if $s = M - 1$ ⋆ *Special case for last symbol*
   then set $y \leftarrow b + l$; ⋆ *end of interval*
   else set $y \leftarrow b + \gamma \cdot \tilde{C}(s+1)$; ⋆ *base of next subinterval*
3. set { $b \leftarrow b + \gamma \cdot \tilde{C}(s)$; ⋆ *Update interval base*
   $l \leftarrow y - b$; } ⋆ *Update interval length as difference*
4. return.

•

Algorithm 12

Function Interval_Selection $(v, b, l, M, \tilde{\mathbf{C}})$

1. set { $s \leftarrow M - 1$; ⋆ *Start search from last symbol*
   $\gamma = l/\tilde{C}(M)$ ⋆ *Compute division result*
   $x \leftarrow b + \gamma \cdot \tilde{C}(M-1)$; ⋆ *Base of search interval*
   $y \leftarrow b + l$; } ⋆ *End of search interval*
2. while $x > v$ do ⋆ *Sequential search for correct interval*
   2.1. set { $s \leftarrow s - 1$; ⋆ *Decrement symbol by one*
   $y \leftarrow x$; ⋆ *move interval end*
   $x \leftarrow b + \gamma \cdot \tilde{C}(s)$; } ⋆ *compute new interval base*
3. set { $b \leftarrow x$; ⋆ *Update interval base*
   $l \leftarrow y - b$; } ⋆ *Update interval length as difference*
4. return s.

•

Only a few changes to the algorithms of Section 2.1.1 are sufficient to include the ability to dynamically update the cumulative distribution. First, we may use (2.10) to change all multiplications by $c(s)$, as follows.

$$b \leftarrow b + \frac{l \cdot \tilde{C}(s)}{\tilde{C}(M)} \tag{2.11}$$

Similarly, the changes to the integer arithmetic version in Appendix A may be in the form.

$$B \leftarrow B + \left\lfloor \frac{L \cdot \tilde{C}(s)}{\tilde{C}(M)} \right\rfloor \tag{2.12}$$

However, since divisions are much more expensive than additions and multiplications, it is better to compute $\gamma = l/\tilde{C}(M)$ once, and implement interval updating as shown in Algorithm 11 [50, 51]. The corresponding changes to Algorithm 7 is shown in Algorithm 12.

In Algorithms 7 and 12 we may need to compute several multiplications, under the assumption they are faster than a single division (see Algorithm 21 for a more efficient search method). If this is not true, Algorithm 13 shows how to replace those multiplications by one extra division.

Algorithm 13

Function Interval_Selection $(v, b, l, M, \tilde{\mathbf{C}})$

1. set { $s \leftarrow M - 1$; — ⋆ *Initialize search from last symbol*
   $\gamma = l/\tilde{C}(M)$ — ⋆ *Compute division result*
   $W \leftarrow (v - b)/\gamma$; } — ⋆ *Code value scaled by $\tilde{C}(M)$*
2. while $\tilde{C}(s) > W$ do — ⋆ *Look for correct interval*
   2.1. set $s \leftarrow s - 1$; — ⋆ *decrement symbol by one*
3. set { $b \leftarrow b + \gamma \cdot \tilde{C}(s)$; — ⋆ *Update interval base*
   $l \leftarrow \gamma \cdot [\tilde{C}(s+1) - \tilde{C}(s)]$; } — ⋆ *Update interval length*
4. return s.

•

Algorithm 14

Procedure Update_Distribution $(s, M, \tilde{\mathbf{C}})$

1. for $m = s + 1$ to M do — ⋆ *For all symbols larger than s*
   1.1 set $\tilde{C}(s) \leftarrow \tilde{C}(s) + 1$; — ⋆ *increment cumulative distribution*
2. return.

•

After the modifications above, Algorithms 1 and 6 are made adaptive by adding the following line:

2.4. Update_Distribution $(s_k, M, \tilde{\mathbf{C}})$;

The procedure to update the cumulative distribution is shown in Algorithm 14. Note that in this algorithm it is necessary to compute up to M additions. Similarly, in Step 2 of Algorithm 13 we have to perform up to M comparisons and subtractions. Since the number of operations in both cases decreases with the symbol number, it is good to sort the symbol by increasing probability. Reference [25] presents an implementation that simultaneously updates the distribution while keeping it sorted.

2.2.3 Binary Arithmetic Coding

Binary arithmetic coders work only with a binary source alphabet ($M = 2$). This is an important type of encoder because it helps to solve many of the complexity issues created with the dynamic update of the cumulative distributions (Algorithm 14). When $M = 2$ the cumulative distribution vector is simply $\mathbf{c} = [\,0\ p(0)\ 1\,]$, which makes coding and updating the cumulative distribution much simpler tasks.

However, it is important to observe that there is a performance trade-off here. While binary arithmetic coding greatly simplifies coding each binary data symbol, its final throughput of information (actual information bits) cannot be larger than one bit per coded symbol, which normally means one bit per few CPU clock cycles [63]. Consequently, they are not as attractive for fast coding as they used to be, but there is no reason not to use their special properties when coding binary sources.

Algorithms 15, 16, and 17, have the procedures for, respectively, binary encoding (inter-

Algorithm 15

Procedure Binary_Interval_Update $(s, b, l, \tilde{C}(1), \tilde{C}(2))$

1. set $x \leftarrow l \cdot \tilde{C}(1)/\tilde{C}(2)$; ⋆ *Point for interval division*
2. if $s = 0$ ⋆ *If symbol is zero*
   then set $l \leftarrow x$; ⋆ *Update interval length,*
   else set { $b \leftarrow b + x$; ⋆ *move interval base and*
   $l \leftarrow l - x$; } ⋆ *update interval length*
3. return.

•

Algorithm 16

Function Binary_Interval_Selection $(v, b, l, \tilde{C}(1), \tilde{C}(2))$

1. set $x \leftarrow l \cdot \tilde{C}(1)/\tilde{C}(2)$; ⋆ *Point for interval division*
2. if $b + x > v$ ⋆ *Look for correct interval*
   then set { $s \leftarrow 0$; ⋆ *Symbol is 0: no change to interval base*
   $l \leftarrow x$; } ⋆ *update interval length*
   else set { $s \leftarrow 1$; ⋆ *Symbol is 1:*
   $b \leftarrow b + x$; ⋆ *move interval base and*
   $l \leftarrow l - x$; } ⋆ *update interval length*
3. return s.

•

val update), decoding (interval selection and update), and distribution update. Note that instead of using the full vector $\tilde{\mathbf{C}}$ we use only $\tilde{C}(1) = \tilde{P}(0)$ and $\tilde{C}(2) = \tilde{P}(0) + \tilde{P}(1)$. The renormalization procedures do not have to be changed for binary arithmetic coding.

Example 11

◁ Binary arithmetic coding has universal application because, just as any number can be represented using bits, data symbols from any alphabet can be coded as a sequence of binary symbols. Figure 2.3 shows how the process of coding data from a 6-symbol source can be decomposed in a series of binary decisions, which can be represented as a *binary search tree* [30]. The leaf nodes correspond to the date source symbols, and intermediate nodes correspond to the decisions shown below them.

Underlined numbers are used for the intermediate nodes, and their value corresponds to the number used for the comparison (they are the "*keys*" for the binary search tree [30]). For instance, node $\underline{m}$ corresponds to test "$s < m$?" Symbols are coded starting from the root of the tree, and continue until a leaf node is encountered. For example, if we want to code the symbol $s = 2$, we start coding the information "$s < 3$?" indicated by node $\underline{3}$; next we go to node $\underline{1}$, and code the information "$s < 1$?", and finally move to node $\underline{2}$ to code "$s < 2$?". At each node the information is coded with a different set of probabilities, which in Figure 2.3 are shown below the tree nodes. These probabilities, based on the number of symbol occurrences, are updated with Algorithm 17. An alphabet with M symbols needs $M - 1$ probability

Algorithm 17

Procedure Update_Binary_Distribution $(s, \tilde{\mathbf{C}})$

1. if $s = 0$ then set $\tilde{C}(1) \leftarrow \tilde{C}(1) + 1$; ⋆ *If $s = 0$ then increment 0-symbol counter*
2. set $\tilde{C}(2) \leftarrow \tilde{C}(2) + 1$; ⋆ *Increment symbol counter*
3. return.

estimates for the intermediate nodes. The decoder follows the same order, using the same set of probabilities. (Note that this is equivalent to using the scheme explained in Section 1.6.3, and shown in Figure 1.5.)

There is no loss in compression in such scheme. For example, when coding symbol $s = 2$ we can compute the symbol probability as a product of conditional probabilities [22].

$$\begin{aligned} \text{Prob}(s = 2) &= \text{Prob}(s < 3) \cdot \text{Prob}(s = 2 | s < 3) \\ &= \text{Prob}(s < 3) \cdot \text{Prob}(s \geq 1 | s < 3) \cdot \text{Prob}(s = 2 | 1 \leq s < 3) \\ &= \text{Prob}(s < 3) \cdot \text{Prob}(s \geq 1 | s < 3) \cdot \text{Prob}(s \geq 2 | s \geq 1, s < 3) \end{aligned} \quad (2.13)$$

This means that

$$\begin{aligned} \log_2 [\text{Prob}(s = 2)] &= \log_2 [\text{Prob}(s < 3)] + \log_2 [\text{Prob}(s \geq 1 | s < 3)] + \\ &\quad + \log_2 [\text{Prob}(s \geq 2 | 1 \leq s < 3)] \end{aligned} \quad (2.14)$$

The left-hand-side of (2.14) is the number of bits required to code symbol $s = 2$ directly with a 6-symbol model, which is equal to the sum of the bits used to code the same symbol by successively coding the binary symbols in the binary-search tree (see Figure 2.3). We do not have to worry about computing explicit values for the conditional probabilities because, when we use a different adaptive binary model for each node, we automatically get the estimate of the proper conditional probabilities. This property is valid for binary-tree decompositions of any data alphabet. ▷

A binary-search tree as in Figure 2.3 can be automatically generated using, for example, the bisection algorithm [17, 19, 39]. Algorithms 18 and 19 show such implementations for encoding, decoding, and overall probability estimation updates. Note that they use the binary coding and decoding functions of Algorithms 15 and 16, and use only one vector, $\bar{\mathbf{C}}$, with dimension $M-1$, to store the tree-branch occurrence counters. Each component $\bar{C}(m)$, $0 < m < M$, contains the number of the number of times we had "$s < m$?" on tree node $\underline{m}$, and $\bar{C}(0)$ contains the total number of symbols coded. This vector is initialized with the number of leaf nodes reachable from the left branch of the corresponding node.

The binary conditional probabilities estimates are computed during the encoding and decoding directly from $\bar{\mathbf{C}}$. For example, in the example of Figure 2.3 we have

$$\bar{\mathbf{C}} = [\,121\ 4\ 8\ 22\ 15\ 19\,].$$

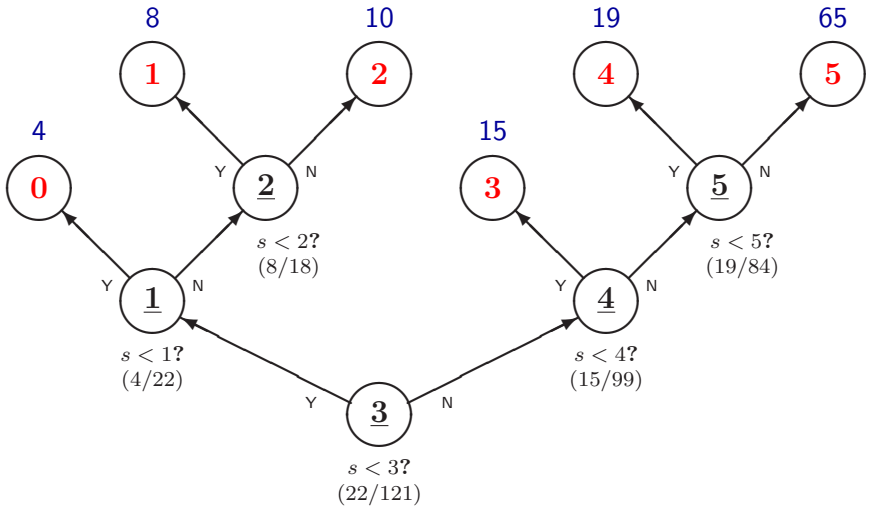

Figure 2.3: Example of a binary search tree with the sequential decisions required for coding data from a 6-symbol alphabet using binary encoders. Leaf nodes represent data symbols, and the numbers above them represent their number of occurrences, $\tilde{P}(s)$. The binary information indicated by each question mark is coded with the probability estimate shown in parenthesis.

In order to code the decision at node $\underline{3}$ we use probability estimate 22/121, which is defined by counters $\bar{C}(0) = 121$ and $\bar{C}(3) = 22$. If we move to node $\underline{1}$ then the next probability estimate is $\bar{C}(1)/\bar{C}(3) = 4/22$. On the other hand, if we move node $\underline{4}$ we need to use probability estimate $\bar{C}(4)/[\bar{C}(0) - \bar{C}(3)] = 15/99$, which is also readily computed from components of $\bar{\mathbf{C}}$. See Algorithms 18 and 19 for details.

Since we use bisection search in Algorithms 18 and 19, the number of times the binary encoding and decoding functions are called is between $\lfloor \log_2 M \rfloor$ and $\lceil \log_2 M \rceil$, for all data symbols. Thus, by using binary-tree searches and binary arithmetic coding we greatly reduce the *worst-case* complexity required to update probability estimates.

Example 12

◁ Figure 2.4 shows a second example of a binary-search tree that can be used to code the 6-symbol data of Example 10. Different search algorithms can be used to create different trees. For example, we could have used a sequential search, composed of tests, "$s < 5$?", "$s < 4$?", "$s < 3$?", ..., "$s < 1$?"

The trees created from bisection search minimize the maximum number of binary symbols to be coded, but not the average. The tree of Figure 2.4 was designed so that the most frequent symbols are reached with the smallest number of tests. Table 2.4 shows the average number of symbols required for coding symbols from the source of

Algorithm 18

Procedure Interval_Update $(s, b, l, M, \bar{\mathbf{C}})$

1. set { $u \leftarrow 0$; $n \leftarrow M$; — ⋆ *Initialize bisection search limits*
   $k \leftarrow \bar{C}(0)$; — ⋆ *First divisor = symbol counter*
   $\bar{C}(0) \leftarrow \bar{C}(0) + 1$; } — ⋆ *Increment symbol counter*
2. while $n - u > 1$ do — ⋆ *Bisection search loop*
   2.1. set $m \leftarrow \lfloor (u + n)/2 \rfloor$; — ⋆ *Compute middle point*
   2.2. if $s < m$ — ⋆ *If symbol is smaller than middle*
   then set { $n \leftarrow m$; — ⋆ *then update upper limit*
   Binary_Interval_Update $(0,\ b,\ l,\ \bar{C}(m),\ k)$; — ⋆ *code symbol 0*
   $k \leftarrow \bar{C}(m)$; — ⋆ *set next divisor*
   $\bar{C}(m) \leftarrow \bar{C}(m) + 1$; } — ⋆ *increment 0-symbol counter*
   else set { $u \leftarrow m$; — ⋆ *else update lower limit*
   Binary_Interval_Update $(1,\ b,\ l,\ \bar{C}(m),\ k)$; — ⋆ *code symbol 1*
   $k \leftarrow k - \bar{C}(m)$; } — ⋆ *set next divisor*
3. return.

•

Algorithm 19

Function Interval_Selection $(v, b, l, M, \bar{\mathbf{C}})$

1. set { $s \leftarrow 0$; $n \leftarrow M$; — ⋆ *Initialize bisection search bounds*
   $k \leftarrow \bar{C}(0)$; — ⋆ *First divisor = symbol counter*
   $\bar{C}(0) \leftarrow \bar{C}(0) + 1$; } — ⋆ *Increment symbol counter*
2. while $n - s > 1$ do — ⋆ *Bisection search loop*
   2.1. set $m \leftarrow \lfloor (s + n)/2 \rfloor$; — ⋆ *Compute middle point*
   2.2. if Binary_Interval_Selection $(v,\ b,\ l,\ \bar{C}(m),\ k) = 0$ — ⋆ *If symbol is smaller than middle*
   then set { $n \leftarrow m$; — ⋆ *then update upper limit*
   $k \leftarrow \bar{C}(m)$; — ⋆ *set next divisor*
   $\bar{C}(m) \leftarrow \bar{C}(m) + 1$; } — ⋆ *increment 0-symbol counter*
   else set { $s \leftarrow m$; — ⋆ *else update lower limit*
   $k \leftarrow k - \bar{C}(m)$; } — ⋆ *set next divisor*
3. return s.

•

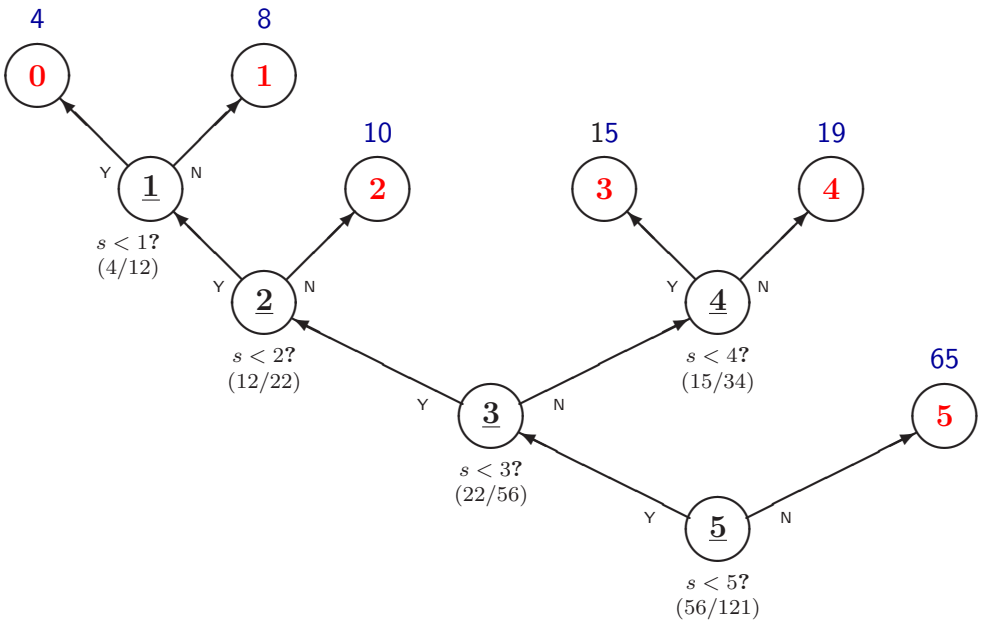

Figure 2.4: Another example of a binary-search tree with for coding data from a 6-symbol source. The number of symbol occurrences is the same as shown in Figure 2.3. This tree has been designed to minimize the average number of binary symbols coded.

Example 11 (considering probability estimates as the actual probabilities), for different trees created from different search methods.

Because we have the symbols sorted by increasing probability, the performance of the tree defined by sequential search, starting from the most probable symbol, is quite good. The tree of Figure 2.4 is the one that minimizes the average number of coded binary symbols. Below we explain how it is designed. ▷

Prefix coding [4, 5, 21, 32, 55, 56] is the process of coding information using decision trees as defined above. The coding process we have shown above is identical, except that we call a binary arithmetic encoding/decoding function at each node. Thus, we can use all the known facts about prefix coding to analyze the computational complexity of binary arithmetic encoding/decoding, if we measure complexity by the number of coded binary symbols.

Since the optimal trees for prefix coding are created using the Huffman algorithm [2], these trees are also optimal for binary arithmetic encoding/decoding [23]. Strictly speaking, if the data symbols are not sorted according to their probability, the optimal Huffman tree does not satisfy the requirements for binary-search trees, i.e., "keys" are not properly sorted, and we cannot define a node with a simple comparison of the type "$s < m$?" This problem is solved by storing the paths from the root node to leaf nodes, i.e., the Huffman codewords.

Data Symbol s	Probability estimate $p(s)$	Number of binary symbols coded					
		Sequential search		Bisection search		Optimal search	
		$N(s)$	$p(s)N(s)$	$N(s)$	$p(s)N(s)$	$N(s)$	$p(s)N(s)$
0	0.033	6	0.198	2	0.066	4	0.132
1	0.066	5	0.331	3	0.198	4	0.264
2	0.083	4	0.331	3	0.248	3	0.248
3	0.124	3	0.372	2	0.248	3	0.372
4	0.157	2	0.314	3	0.471	3	0.471
5	0.537	1	0.537	3	1.612	1	0.537
Sum	1.000	21	2.083	16	2.843	18	2.025

Table 2.4: Number binary symbols coded using trees created from different types of binary searches, applied to data source of Example 10. The trees corresponding to bisection and optimal searches are shown in Figures 2.3 and 2.4, respectively.

2.2.4 Tree-based Update of Cumulative Distributions

In this section, we show that we can use binary-search trees (Section 2.2.3) to efficiently combine computing the cumulative distribution, updating it, encoding and decoding, without having to use a binary arithmetic encoder. We present techniques similar to the methods proposed by Moffat [31] and Fenwick [42]. We start with an example of how to compute the cumulative distribution vector $\tilde{\mathbf{C}}$ from the statistics $\bar{\mathbf{C}}$ gathered while using the binary search trees.

Example 13

◁ Let us consider the binary search tree shown in Figure 2.3. Let us assume that we had been using Algorithms 18 and 19 to compute the number of symbols occurrences in the tree, $\bar{\mathbf{C}}$, and we want to compute the cumulative distribution $\tilde{\mathbf{C}}$ from $\bar{\mathbf{C}} = [\,121\ 4\ 8\ 22\ 15\ 19\,]$.

From the tree structure we can find out that, except for the root node, the counter at each node has the number of occurrences of all symbols found following the left branch, i.e.,

$$\begin{aligned}
\bar{C}(0) \equiv\ & \tilde{P}(0)+\tilde{P}(1)+\tilde{P}(2)+\tilde{P}(3)+\tilde{P}(4)+\tilde{P}(5) && = 121\\
\bar{C}(1) =\ & \tilde{P}(0) && = 4\\
\bar{C}(2) =\ & \tilde{P}(1) && = 8\\
\bar{C}(3) =\ & \tilde{P}(0)+\tilde{P}(1)+\tilde{P}(2) && = 22\\
\bar{C}(4) =\ & \tilde{P}(3) && = 15\\
\bar{C}(5) =\ & \tilde{P}(4) && = 19
\end{aligned}$$

where $\tilde{P}(s)$ is the number of time symbol s has occurred. From these equations we

can compute the vector with cumulative distributions as

$$\begin{array}{lll}
\tilde{C}(0) \equiv & 0 & \\
\tilde{C}(1) = & \tilde{P}(0) & = \bar{C}(1) = 4 \\
\tilde{C}(2) = & \tilde{P}(0) + \tilde{P}(1) & = \bar{C}(1) + \bar{C}(2) = 12 \\
\tilde{C}(3) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) & = \bar{C}(3) = 22 \\
\tilde{C}(4) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) & = \bar{C}(3) + \bar{C}(4) = 37 \\
\tilde{C}(5) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) & = \bar{C}(3) + \bar{C}(4) + \bar{C}(5) = 56 \\
\tilde{C}(6) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) & = \bar{C}(0) = 121
\end{array}$$

We can do the same with the tree of Figure 2.4, and find different sets of equations. In this case the counters are

$$\begin{array}{ll}
\bar{C}(0) \equiv & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) = 121 \\
\bar{C}(1) = & \tilde{P}(0) = 4 \\
\bar{C}(2) = & \tilde{P}(0) + \tilde{P}(1) = 12 \\
\bar{C}(3) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) = 22 \\
\bar{C}(4) = & \tilde{P}(3) = 15 \\
\bar{C}(5) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) = 56
\end{array}$$

and their relationship with the cumulative distribution is given by

$$\begin{array}{lll}
\tilde{C}(0) \equiv & 0 & \\
\tilde{C}(1) = & \tilde{P}(0) & = \bar{C}(1) = 4 \\
\tilde{C}(2) = & \tilde{P}(0) + \tilde{P}(1) & = \bar{C}(2) = 12 \\
\tilde{C}(3) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) & = \bar{C}(3) = 22 \\
\tilde{C}(4) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) & = \bar{C}(3) + \bar{C}(4) = 37 \\
\tilde{C}(5) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) & = \bar{C}(5) = 56 \\
\tilde{C}(6) = & \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) & = \bar{C}(0) = 121
\end{array}$$

▷

The equations that we obtain from any decision tree are linearly independent, and thus it is always possible to compute the cumulative distribution $\tilde{\mathbf{C}}$ from the counters $\bar{\mathbf{C}}$. We show next how to use the tree structure to efficiently compute the components of $\tilde{\mathbf{C}}$ required for encoding symbol s, $\tilde{C}(s)$ and $\tilde{C}(s+1)$.

In order to compute $\tilde{C}(s)$, when we move from the root of the tree, up to the leaf representing symbol s, we simply add the value of $\bar{C}(n)$ for each node $\underline{n}$ that does not have its condition satisfied (i.e., we move up its right-side branch). For example, to compute $\tilde{C}(2)$ using the tree of Figure 2.3, we start from node $\underline{3}$, and move left to node $\underline{1}$, right to node $\underline{2}$, and right to leaf 2. Since we move along the right branch of nodes $\underline{1}$ and $\underline{2}$, we conclude that $\tilde{C}(2) = \bar{C}(1) + \bar{C}(2) = 12$.

The value of $\tilde{C}(s+1)$ can be computed together with $\tilde{C}(s)$: we just have to add the running sum for $\tilde{C}(s)$ and $\bar{C}(n)$ at the last left-side branch taken before reaching leaf s. For example, when computing $\tilde{C}(2)$, the last left-side branch is taken at root node $\underline{3}$, and thus $\tilde{C}(3)$ is equal to the running sum (zero) plus $\bar{C}(3) = 22$.

Algorithm 20

Procedure Interval_Update $(s, b, l, M, \bar{\mathbf{C}})$

1. set { $u \leftarrow 0$; $n \leftarrow M$;
   $E \leftarrow 0$; $F \leftarrow \bar{C}(0)$;
   $\gamma \leftarrow l/\bar{C}(0)$; }
2. while $n - u > 1$ do
   2.1. set $m \leftarrow \lfloor (u+n)/2 \rfloor$;
   2.2. if $s < m$
   then set { $n \leftarrow m$;
   $F \leftarrow E + \bar{C}(m)$;
   $\bar{C}(m) \leftarrow \bar{C}(m) + 1$; }
   else set { $u \leftarrow m$;
   $E \leftarrow E + \bar{C}(m)$; }
3. if $s = M - 1$
   then set $y \leftarrow b + l$;
   else set $y \leftarrow b + \gamma \cdot F$;
4. set { $b \leftarrow b + \gamma \cdot E$;
   $l \leftarrow y - b$;
   $\bar{C}(0) \leftarrow \bar{C}(0) + 1$; }
5. return.

⋆ *Initialize bisection search limits*
⋆ *Initialize cumulative sum bounds*
⋆ *Compute result of division*
⋆ *Bisection search loop*
  ⋆ *Compute middle point*
  ⋆ *If symbol is smaller than middle*
    ⋆ *then update bisection upper limit*
    ⋆ *set upper bound on cum. sum*
    ⋆ *Increment node occurrence counter*
    ⋆ *else update bisection lower limit*
    ⋆ *increment lower bound on cum. sum*
⋆ *Set interval end according to symbol*
  ⋆ *Exact multiplication*
  ⋆ *Base of next subinterval*
⋆ *Update interval base*
⋆ *Update interval length as difference*
⋆ *Increment symbol counter*

Algorithms 20 and 21 show how to combine all the techniques above to simultaneously compute and update the cumulative distribution, and use the computed values for encoding and decoding. They use a tree defined by bisection search, but it is easy to modify them to other tree structures. After Step 2 of Algorithm 20 we have $E = \tilde{C}(s)$ and $F = \tilde{C}(s+1)$ computed and updated with a number of additions proportional to $\log_2(M)$.

2.2.5 Periodic Updates of the Cumulative Distribution

We have seen in Section 2.2.2 that adaptive coding can increase the arithmetic coding computational complexity significantly, because of the effort to update the cumulative distributions. In Sections 2.2.3 and 2.2.4 we present tree-based updating techniques that reduce this complexity very significantly. However, adaptation can still be a substantial fraction of the overall coding computational complexity, and it happens that there is not much we can do if we insist on the assumption that the probability model has to be updated immediately after each encoded/decoded symbol, i.e., estimates are in the form given by (2.10), with a division by the factor $\tilde{C}(M)$.

However, with accurate and fast source modeling we can avoid having probabilities changing so quickly that we need to refresh estimates on a symbol-by-symbol basis. For example, an image encoder may use different probabilities depending on a classification of the part being encoded, which can be something like "constant," "smooth," "edge," "high-frequency pattern," etc. With these techniques, we can assume that the source state may change quickly, but the source properties (symbol probabilities) for each state change slowly.

Algorithm 21

Function Interval_Selection $(v, b, l, M, \bar{\mathbf{C}})$

1. set { $s \leftarrow 0$; $n \leftarrow M$;
   $x \leftarrow b$; $y \leftarrow b + l$;
   $E \leftarrow 0$;
   $\gamma \leftarrow l/\bar{C}(0)$; }
2. while $n - s > 1$ do
   2.1. set { $m \leftarrow \lfloor (s+n)/2 \rfloor$;
   $z \leftarrow b + \gamma \cdot [E + \bar{C}(m)]$; }
   2.2. if $z > v$
   then set { $n \leftarrow m$;
   $y \leftarrow z$;
   $\bar{C}(m) \leftarrow \bar{C}(m) + 1$; }
   else set { $s \leftarrow m$;
   $x \leftarrow z$;
   $E \leftarrow E + \bar{C}(m)$; }
3. set { $b \leftarrow x$;
   $l \leftarrow y - x$;
   $\bar{C}(0) \leftarrow \bar{C}(0) + 1$; }
4. return s.

⋆ *Initialize bisection search symbol limits*
⋆ *Set search interval bounds*
⋆ *Initialize cumulative sum*
⋆ *Compute result of division*
⋆ *Bisection search loop*
⋆ *Compute middle point*
⋆ *Value at middle point*
⋆ *If symbol is smaller than middle*
⋆ *then update bisection upper limit*
⋆ *new interval end*
⋆ *increment node occurrence counter*
⋆ *else update bisection lower limit*
⋆ *new interval base*
⋆ *increment lower bound on cum. sum*
⋆ *Update interval base*
⋆ *Update interval length as difference*
⋆ *Increment symbol counter*

Under these assumptions, and unless the number of data symbols is very large (thousands or millions [51]), a significantly more efficient form of adaptive arithmetic coding updates only the vector with symbol occurrence counters ($\tilde{\mathbf{P}}$) after each symbol, and updates the distribution estimate ($\mathbf{c}$) periodically, or following some special events. For example, the Q-coder updates its probability estimation only during renormalization [27]. For periodic updates, we define R as number of symbol coded between updates of $\mathbf{c}$. Typically, the period R is a small multiple of M (e.g., $R = 4M$), but to improve the accuracy while minimizing the computations, we can start with frequent updates, and then decrease their frequency.

One immediate consequence of this approach is that while coding we can use the simpler procedures of Section 2.1.1 and Appendix A, and can completely avoid the divisions in equations (2.10) to (2.12). Furthermore, if the period R is large enough, then it is feasible to do many complex tasks while updating $\mathbf{c}$, in order to increase the encoding/decoding speed. For instance, in Section 2.3.2 we explain how to make decoding faster by sorting the symbols according to their probability, finding the optimal decoding sequence (Huffman tree), or computing the data for fast table look-up decoding.

2.3 Complexity Analysis

Large computational complexity had always been a barrier to the adoption of arithmetic coding. In fact, for many years after arithmetic coding was invented, it was considered little more than a mathematical curiosity because the additions made it slow, multiplications made it very expensive, and divisions made it impractical. In this section, we analyze the

complexity of arithmetic coding and explain how the technological advances that gives us fast arithmetic operations change the complexity analysis.

Many of the conclusions in this section are based on experimental tests designed for analyzing the arithmetic coding complexity, which are described in reference [63]. Another earlier experimental evaluation of complexity is in reference [43].

We have to observe that the *relative* computational complexity of different coding operations changed dramatically. Arithmetic operations today are much more efficient, and not only due the great increase in the CPU clock frequency. In the past, the ratio between clock cycles used by some simple operations (comparisons, bit shifts, table look-up) and arithmetic operations (specially division) was quite large. Today, this ratio is much smaller [58, 59, 60, 61], invalidating previous assumptions for complexity reduction. For instance, a set of comparisons, bit shifts, and table look-up now takes significantly more time than a multiplication.

The speed of arithmetic coding needs to be measured by the rate of data (true information) bits coming out of the encoder, or into the decoder (*information throughput*). For example, binary coders can be very simple and fast, but their throughput is limited to a fraction of bits per CPU clock cycle. Arithmetic coders with larger alphabets, on the other hand, process information in a significantly more efficient manner, and can easily yield throughputs of many bits per CPU clock cycle [63].

In this section we use the "big-O" notation of [30], where $O(\cdot)$ indicates asymptotic upper bounds on the computational complexity.

The main factors that influence complexity are

- Interval renormalization and compressed data input and output.
- Symbol search.
- Statistics estimation (adaptive models only).
- Arithmetic operations.

In the next sections we analyze each of these in some detail. However, we will not consider special hardware implementations (ASICs) for three reasons: (a) it is definitely outside the scope of this text; (b) our model also applies to some specialized hardware, like DSP chips; (c) many optimization techniques for general CPUs also apply to efficient hardware.

2.3.1 Interval Renormalization and Compressed Data Input and Output

In integer arithmetic implementations the interval renormalization can be calculated with only bit shifts. However, when $D = 2$, renormalization occurs quite frequently, consuming many clock cycles. Using larger output alphabets in the form $D = 2^r$ reduces the frequency of the renormalization by a factor of r (see Example 10), and thus may reduce the number of clock cycles used for renormalization very significantly. Floating-point implementations may need multiplication during renormalization, but if D is large enough then these operations do not add much to the overall complexity.

Data input and output, which occurs during renormalization, also has significant impact on the encoding and decoding speed. Since computers and communication systems work with groups of 8 bits (bytes), processing one bit at a time requires extra clock cycles to properly align the data. Thus, there is substantial speedup when renormalization produces bits that can be easily aligned in bytes (e.g., $D = 2^4$, $D = 2^8$, or $D = 2^{16}$). The case $D = 256$ has been shown to be particularly efficient [52, 54].

2.3.2 Symbol Search

The computational complexity of the arithmetic decoder may be many times larger than the encoder's because the decoder needs to find out in which subinterval the current code value is, i.e., solve

$$\hat{s}(v) = \{\, s :\ c(s)l \le v - b < c(s+1)l \}\,, \tag{2.15}$$

or

$$\hat{s}(v) = \left\{ s :\ c(s) \le \frac{v-b}{l} < c(s+1) \right\}. \tag{2.16}$$

Mathematically these are equivalent, but we use (2.16) to represent the case in which the division $(v-b)/l$ is computed first (e.g., Algorithm 13).

This is a line-search problem [17, 30], where we need to minimize both the number of points tested and the computations used for determining these points. The possible difference in complexity between the encoder and decoder grows with the number of data symbols, M. We can see from Algorithms 15 and 16 that when $M = 2$, the complexity is practically the same for the encoder and decoder.

There are five main schemes for symbol search that are described next.

(a) Sequential search on sorted symbols

This search method, used in Algorithms 7 and 12, in the worst-case searches $M-1$ intervals to find the decoded symbol. We can try to improve the average performance by sorting the symbols according to their probability. Assuming that the symbols are sorted with increasing probability, the average number of tests is

$$\bar{N}_s(\Omega) = \sum_{m=0}^{M-1} p(m)(M-m) = M - \sum_{m=0}^{M-1} m\, p(m) = M - \bar{s}(\Omega) \le M, \tag{2.17}$$

where $\bar{s}(\Omega)$ is the average symbol number (after sorting) put out by data source Ω. Thus, sequential search can be efficient only if the symbol distribution is very skewed ($\bar{s}(\Omega) \approx M-1$).

(b) Bisection search

With this type of search, shown in Algorithms 19, 21 and 28, the number of tests is bounded by

$$\lfloor \log_2(M) \rfloor \le \bar{N}_b(\Omega) \le \lceil \log_2(M) \rceil\,, \tag{2.18}$$

independently of the probabilities of the data symbols. Note that the *encoder* may also have to implement the bisection search when it is used with binary coding (Section 2.2.3), or when using trees for updating the probability estimates (Section 2.2.4).

(c) Optimal tree-based search

We show in Section 2.2.4 how the process of encoding and decoding data from a M-symbol alphabet can be decomposed in a set of binary decisions, and in Example 12 we show its similarities to prefix coding, and an optimal decision tree designed with the Huffman algorithm. The interval search process during decoding is also defined by binary decisions, and the optimal set of decisions is also computed with the Huffman algorithm [2, 23]. Thus, we can conclude that the average number of tests in such scheme is bounded as the average number of bits used to code the source with Huffman coding, which is given by [9]

$$\max\{1, H(\Omega)\} \leq \bar{N}_o(\Omega) \leq H(\Omega) + 0.086 + \max_{m=0,1,\ldots,M-1}\{p(m)\}. \tag{2.19}$$

Implementing the optimal binary-search tree requires some extra storage, corresponding to the data normally used for representing a Huffman code, and it is necessary to reorder the symbols according to probability (as in the example of Figure 2.4), or use a modified definition of the cumulative distribution.

With adaptive coding we have the problem that the optimal tree is defined from the symbol probabilities, which are unknown when encoding and decoding start. This problem can be solved by periodically redesigning the tree, together with the distribution updates (Section 2.2.5).

(d) Bisection search on sorted symbols

We can combine the simplicity of bisection with a technique that takes into account the symbol probabilities. When the data symbols are sorted with increasing probability, we can look for the symbol m such that $c(m) \approx 0.5$, and use it as the first symbol (instead of middle point) to divide the interval during bisection. If the distribution is nearly uniform, then this symbol should be near the middle and the performance is similar to standard bisection search. On the other hand, if the distribution is highly skewed, them $m \approx M - 1$, meaning that the most probable symbols are tested first, reducing the average number of tests.

We can extend this technique to find also find the symbols with $c(m)$ near 0.25 and 0.75, and then 0.125, 0.375, 0.625, and 0.825, and so on. Figure 2.5 shows an example of a cumulative distribution, and the process of finding the symbols for initializing the bisection search. The corresponding first levels of the binary-search tree are shown in Figure 2.6. The decoder does not have to use all levels of that tree; it can use only the first level (and store one symbol), or only the first two levels (and store three symbols), or any desired number of levels. In Figure 2.6, the complete binary-search tree is defined by applying standard bisection search following the tests shown in the figure. If the encoder uses up to $V > 1$ levels of the binary-search tree defined by the cumulative distribution, the average number of tests is near the optimal (2.19), and the worst case is not much larger than (2.18).

(e) Bisection search starting from table look-up

The interval search methods that use (2.16) require one division, but its advantage is that it is equivalent to rescaling the current interval to unit length before search. This means that from a quantized version of the fraction $(v - b)/l$, like

$$E(v, b, l) = \left\lfloor \frac{K_t(v - b)}{l} \right\rfloor, \tag{2.20}$$

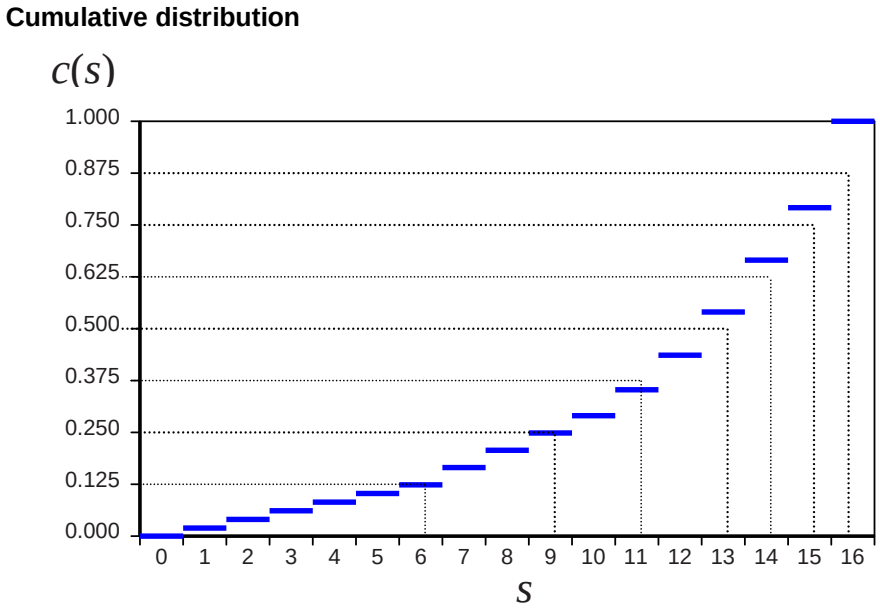

Figure 2.5: Technique to find quasi-optimal points to initialize bisection search on sorted symbols.

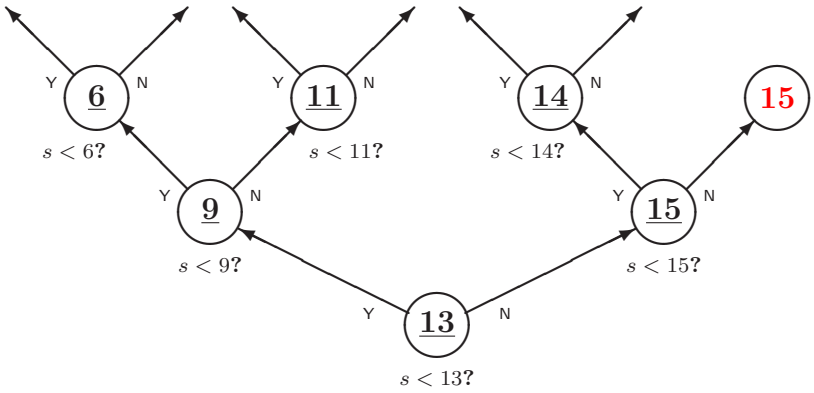

Figure 2.6: Decision process corresponding to the quasi-optimal binary-search initialization of Figure 2.5.

$E(v,b,l)$	$(v-b)/l$	$\hat{s}_{\min}(E)$	$\hat{s}_{\max}(E)$
0	[0.000, 0.125)	0	0
1	[0.125, 0.250)	0	1
2	[0.250, 0.375)	1	1
3	[0.375, 0.500)	1	1
4	[0.500, 0.625)	1	1
5	[0.625, 0.750)	1	2
6	[0.750, 0.875)	2	2
7	[0.875, 1.000)	2	3

Table 2.5: Values of $(v-b)/l$ quantized to 8 integer values to allow fast decoding using table look-up. The minimum and maximum possible values of the decoded symbol (table entries) were computed using $\mathbf{c} = [\,0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1\,]$.

we can know better initial values for the bisection search, which can be stored in tables with K_t elements, enabling significantly faster decoding.

The table entries are

$$\hat{s}_{\min}(E) = \left\{ s : \; c(s) \leq \frac{E}{K_t} < c(s+1) \right\}, \tag{2.21}$$

$$\hat{s}_{\max}(E) = \left\{ s : \; c(s) < \frac{E+1}{K_t} \leq c(s+1) \right\}. \tag{2.22}$$

Note that $\hat{s}_{\max}(E) <= \hat{s}_{\min}(E+1)$, so one table is enough for correct decoding.

Example 14

◁ Let us consider again the source of Example 3, with $\mathbf{c} = [\,0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1\,]$. Table 2.5 shows how each value of $E(v,b,l)$ corresponds to an interval of possible values of $(v-b)/l$. By analyzing $\mathbf{c}$ and these intervals we can identify the range of possible decoded symbols, which correspond to $\hat{s}_{\min}(E)$ and $\hat{s}_{\max}(E)$. For example, if while decoding we have $E(v,b,l) = 0$, then we know that $(v-b)/l < 0.125$, and consequently $\hat{s}(v) = 0$. On the other hand, if $E(v,b,l) = 5$, then $0.625 \leq (v-b)/l < 0.75$, which means that we can only say that $1 \leq \hat{s}(v) \leq 2$, and one more test is necessary for finishing decoding. ▷

Note that, while the number of tests required by the original bisection search does not depend on the probabilities, when we use $\hat{s}_{\min}(E)$ and $\hat{s}_{\max}(E)$ to initialize the search the initial interval depends on the probabilities, and the most probable symbols are found with a smaller number of tests. For instance, we can see in Table 2.5 that most of the table entries correspond to the most probable symbols.

The main constraint for table look-up decoding is that it is practical only for static or periodically updated models (Section 2.2.5). The overhead of computing the table each time the model is updated can be small because table look-up is only for initializing the bisection search, and even small tables (e.g., 16 entries) can make it significantly more efficient.

2.3.3 Cumulative Distribution Estimation

Adaptive arithmetic coding requires an extra effort to update the cumulative distribution estimates after coding each symbol, or in a periodical manner. Section 2.2 covers all the important aspects of how adaptive encoders efficiently update the cumulative distributions $\mathbf{c}$. We analyze two main strategies, depending on whether the cumulative distribution updating is done together or independently of the symbol search (Section 2.3.2).

(a) Combined updating, coding and decoding
In this case, the most efficient implementations use binary-search trees, and updating is applied to vector $\bar{\mathbf{C}}$, instead of $\tilde{\mathbf{C}}$. Binary coding (Section 2.2.3) and tree-based updating (Section 2.2.4) have the same asymptotic complexity, depending on the tree being used. In addition, by comparing Algorithms 18 and 19 and Algorithms 20 and 21, we can see that the encoder and the decoder have very similar computational complexity.

The worst-case effort is minimized when we use bisection search, resulting in a number of operations per coded symbol proportional to $\log_2(M)$ (equation (2.18)). Huffman trees, that optimize the average performance, require an average number of operations proportional to $\bar{N}_o(\Omega)$, defined in equation (2.19).

(b) Independent updating
When source Ω has low entropy, some symbols should occur much more frequently, and it may be efficient to update $\tilde{\mathbf{C}}$ directly, if the symbols are sorted by increasing probabilities. Implementations by Witten *et al.* [25, 51] use this strategy. However, when all symbols are equally probable, the effort for sorting and updating $\tilde{\mathbf{C}}$ is on average proportional to $M/2$, and in the worst-case proportional to $M-1$.

As explained in Section 2.2.5, with periodic updates of $\tilde{\mathbf{C}}$ we can recompute and sort symbols with reasonable worst-case complexity. We assume that the updating period is R coded symbols. One choice is keep the data symbols not sorted, and to update $\tilde{\mathbf{C}}$ using (2.10), which requires $O(M)$ operations per update, and an average complexity of $O(M/R)$ operations per coded symbol. Decoding can use bisection for symbol search. If we choose, for example, $R = M$ then we have $O(1)$ operations per update, which is quite reasonable.

Sorting the symbols according to their probability during each update can be done with $O(M \log(M)/R)$ operations per symbol, in the worst-case [30]. Since symbols are normally already sorted from previous updating passes, insertion sort [30] typically can be done with an average of $O(M/R)$ operations. When the symbols are sorted, we can use the more efficient symbol search of Section 2.3.2(d). Choosing $R = M$ results in $O(log(M))$ operations per symbol.

With periodic updating we can both sort symbols, and find the optimal search strategy (Huffman tree), with a reasonable complexity of $O(M \log(M)/R)$ operations per symbol [30]. However, even though the asymptotic complexity of finding the Huffman code is the same as simply sorting, it requires considerable more effort.

Table 2.6 presents a summary of the results above. Note that for the complexity analysis of optimal searches, we can use $O(H(\Omega) + 1)$, instead of the more complex and tighter bounds given by (2.19). The last columns in Table 2.6 indicates the need to use divisions in the form (2.10). Changing the cumulative distribution every symbol requires using different

Searching & updating methods	Symbol encoding	Decoder interval search	Distribution updating	Divisions
Fixed code optimal search	$O(1)$	$O(H(\Omega)+1)$	NA	NA
Sequential search on sorted symbols	$O(1)$	$O(M)$	$O(M)$	O(1)
Combined updating and coding, bisection tree	$O(\log(M))$			O(1)
Combined updating and coding, optimal tree	$O(H(\Omega)+1)$			O(1)
Periodic update, bisection search	$O(1)$	$O(\log(M))$	$O(M/R)$	O(1/R)
Periodic update, optimal search	$O(1)$	$O(H(\Omega)+1)$	$O(M\log(M)/R)$	O(1/R)

Table 2.6: Computational complexity of symbol search and adaptation (cumulative distribution updating) for fixed and different adaptive arithmetic coding techniques. Typically, $R = 4M$ minimizes the complexity without degrading performance.

divisors for each coded symbol. Periodic updating, on the other hand, allows us to compute the inverse of divisor once, and use it for coding several (R) symbols.

2.3.4 Arithmetic Operations

Presently, additions and multiplications are not much slower than other operations, such as bit shifts and comparison. Divisions, on the other hand, have a much longer latency (number of clock cycles required to perform the operation), and cannot be streamlined with other operations (like special circuitry for multiply-add) [58, 59, 60, 61].

First, let us consider the arithmetic required by fixed coders, and adaptive coders with periodic updating. The arithmetic coding recursion in the forms (1.9) and (1.40) require two multiplications and, respectively, one and three additions. Thus, except for processor with very slow multiplication, encoding requirements are quite reasonable. Decoding is more complex due to the effort to find the correct decoding interval. If we use (1.16) for interval search, we have one extra division, plus several comparisons (Section 2.3.2). The multiplication-only version (1.38) requires several multiplications and comparisons, but with reasonably efficient symbol search this is faster than (1.16), and eliminates the need to define division when multiplications are approximated.

Adaptive coding can be considerably slower because of the divisions in (2.11) and (2.12). They can add up to two divisions per interval updating. In Algorithms 11, 12 and 13, we show how to avoid one of the divisions. Furthermore, updating the cumulative distribution may require a significant number of additions. Note that having only periodic updates of the cumulative distribution significantly reduces this adaptation overhead, because all these

divisions and additions are not required for every coded symbol. For example, a single division, for the computation of $1/\tilde{C}(M)$, plus a number of multiplications and additions proportional to M, may be needed per update. If the update occurs every M coded symbols, then the average number of divisions per coded symbol is proportional to $1/M$, and the average number of multiplications and additions are constants.

2.4 Further Reading

We presented the coding method that is now commonly associated with the name arithmetic coding, but the reader should be aware that other types of arithmetic coding had been proposed [6, 7]. Standard implementations of arithmetic coding had been defined in some international standards [33, 36, 38, 62]. Several techniques for arithmetic coding complexity reduction not covered here are in the references [11, 18, 23, 25, 27, 28, 34, 37, 42, 51, 52]. We did not mention the fact that errors in arithmetic coded streams commonly lead to catastrophic error propagation, and thus error-resilient arithmetic coding [57] is very important in some applications.

Appendix A

Integer Arithmetic Implementation

The following algorithms show the required adaptations in the algorithms in Section 2.1.1 for use with integer arithmetic, and with a D-symbol output alphabet. Typically D is small power of two, like 2, 4, 16, or 256. As explained in Section 2.1.2, we assume that all numbers are represented as integers, but here we define $B = D^P b$, $L = D^P l$, and $C(s) = D^P c(s)$. In addition, we assume multiplications computed with $2P$ digits and results truncated to P digits. Renormalization (2.6) sets $L > D^{P-1}$, and thus the minimum probability allowed is defined by

$$\left\lfloor [C(s+1) - C(s)]\, D^{-P}\, L \right\rfloor \geq 1 \Rightarrow C(s+1) - C(s) \geq D, \tag{A.1}$$

i.e., $p(s) \geq D^{1-P}$.

Algorithm 22, for integer arithmetic, is almost identical to Algorithm 1, except for the initialization of L, and the decision for renormalization. The arithmetic operations that update the interval base may overflow the integer registers. To make clear that this is acceptable, we define the results modulo D^P, as in Algorithm 23. The results of the multiplications are multiplied by D^{-P} and truncated, meaning that the least significant bits are discarded. Overflow is here detected by a reduction in the base value. For that reason, we implement carry propagation together with the interval update in Algorithm 23.

Note that in the renormalization of Algorithm 24, we assume that D is a power of two, and all multiplications and divisions are actually implemented with bit shifts. The carry propagation with a D-symbol output, shown in Algorithm 25, is very similar to Algorithm 3.

In Algorithm 5 the code value selection is made to minimize the number of bits, assuming that the decoder pads the buffer $\mathbf{d}$ with sufficient zeros. This inconvenience can be avoided by simply adding a proper extra symbol at the end of the compressed data. Algorithm 26 shows the required modifications. It shifts the interval base by a small amount, and resets the interval length, so that when procedure Encoder_Renormalization is called, it adds the proper two last output symbols to buffer $\mathbf{d}$. This way, correct decoding does not dependent on the value of the symbols that are read by the decoder (depending on register size P) past the last compressed data symbols.

With integer arithmetic the decoder does not have to simultaneously update the interval base and code value (see Algorithm 8). Since decoding is always based on the difference $v - b$, we define $V = D^P(v - b)$ and use only V and L while decoding. The only other required change is that we must subtract from V the numbers that would have been added to B.

ALGORITHM 22

Function Arithmetic_Encoder $(N, S, M, \mathbf{C}, \mathbf{d})$

1. set { $B \leftarrow 0$; $L \leftarrow D^P - 1$;
   $t \leftarrow 0$; }
2. for $k = 1$ to N do
   2.1. Interval_Update $(s_k, M, B, L, \mathbf{C})$;
   2.2. if $L < D^{P-1}$ then
   Encoder_Renormalization $(B, L, t, \mathbf{d})$;
3. Code_Value_Selection $(B, L, t, \mathbf{d})$;
4. return t.

⋆ *Initialize interval*
⋆ *and symbol counter*
⋆ *Encode N data symbols*
⋆ *Update interval according to symbol*
⋆ *If interval is small enough*
⋆ *then renormalize interval*
⋆ *Choose final code value*
⋆ *Return number of code symbols*

•

ALGORITHM 23

Procedure Interval_Update $(s, M, B, L, \mathbf{C})$

1. if $s = M - 1$;
   then set $Y \leftarrow L$;
   else set $Y \leftarrow \left\lfloor [L \cdot C(s+1)] \cdot D^{-P} \right\rfloor$;
2. set { $A \leftarrow B$;
   $X \leftarrow \left\lfloor [L \cdot C(s)] \cdot D^{-P} \right\rfloor$;
   $B \leftarrow (B + X) \bmod D^P$;
   $L \leftarrow Y - X$; }
3. if $A > B$ then Propagate_Carry $(t, \mathbf{d})$;
4. return.

⋆ *If s is last symbol set first product equal*
⋆ *to current interval length*
⋆ *or else equal to computed value*
⋆ *Save current base*
⋆ *Compute second product*
⋆ *Update interval base*
⋆ *Update interval length*
⋆ *Propagate carry if overflow*

•

ALGORITHM 24

Procedure Encoder_Renormalization $(B, L, t, \mathbf{d})$

1. while $L < D^{P-1}$ do
   1.1. set { $t \leftarrow t + 1$;
   $d(t) \leftarrow \left\lfloor B \cdot D^{1-P} \right\rfloor$;
   $L \leftarrow (D \cdot L) \bmod D^P$;
   $B \leftarrow (D \cdot B) \bmod D^P$; }
2. return.

⋆ *Renormalization loop*
⋆ *Increment symbol counter*
⋆ *Output symbol*
⋆ *Update interval length*
⋆ *Update interval base*

•

Algorithm 25

Procedure Propagate_Carry $(t, \mathbf{d})$

1. set $n \leftarrow t$; ⋆ *Initialize pointer to last outstanding symbol*
2. while $d(n) = D - 1$ do ⋆ *While carry propagation*
   2.1. set $\{$ $d(n) \leftarrow 0$; ⋆ *Set outstanding symbol to zero*
   $n \leftarrow n - 1$; $\}$ ⋆ *Move to previous symbol*
3. set $d(n) \leftarrow d(n) + 1$; ⋆ *Increment first outstanding symbol*
4. return.

•

Algorithm 26

Procedure Code_Value_Selection $(B, L, t, \mathbf{d})$

1. set $\{$ $A \leftarrow B$; ⋆ *Save current base*
   $B \leftarrow \left(B + D^{P-1}/2\right) \bmod D^P$; ⋆ *Increase interval base*
   $L \leftarrow D^{P-2} - 1$; $\}$ ⋆ *Set new interval length*
2. if $A > B$ then Propagate_Carry $(t, \mathbf{d})$; ⋆ *Propagate carry if overflow*
3. Encoder_Renormalization $(B, L, t, \mathbf{d})$ ⋆ *Output two symbols*
4. return.

•

Algorithm 27

Procedure Arithmetic_Decoder $(N, M, \mathbf{C}, \mathbf{d})$

1. set $\{$ $L \leftarrow D^P - 1$; ⋆ *Initialize interval length*
   $V = \sum_{n=1}^{P} D^{P-n} d(n)$; ⋆ *Read P digits of code value*
   $t \leftarrow P$; $\}$ ⋆ *Initialize symbol counter*
2. for $k = 1$ to N do ⋆ *Decoding loop*
   2.1. set $s_k =$ Interval_Selection $(V,\ L,\ M,\ \mathbf{C})$; ⋆ *Decode symbol and update interval*
   2.2. if $L < D^{P-1}$ then ⋆ *If interval is small enough*
   Decoder_Renormalization $(V,\ L,\ t,\ \mathbf{d})$; ⋆ *renormalize interval*
4. return.

•

ALGORITHM 28

Function Interval_Selection $(V, L, M, \mathbf{C})$

1. set { $s \leftarrow 0$; $n \leftarrow M$;
   $X \leftarrow 0$; $Y \leftarrow L$;
2. while $n - s > 1$ do
   2.1. set { $m \leftarrow \lfloor (s+n)/2 \rfloor$;
   $Z \leftarrow \left\lfloor [L \cdot C(m)] \cdot D^{-P} \right\rfloor$; }
   2.2. if $Z > V$
   then set { $n \leftarrow m$;
   $Y \leftarrow Z$; }
   else set { $s \leftarrow m$;
   $X \leftarrow Z$; }
3. set { $V \leftarrow V - X$;
   $L \leftarrow Y - X$; }
4. return s.

⋆ *Initialize bisection search limits*
⋆ *Initialize bisection search interval*
⋆ *Binary search loop*
⋆ *Compute middle point*
⋆ *Compute value at middle point*
⋆ *If new value larger than code value*
⋆ *then update upper limit*
⋆ *else update lower limit*
⋆ *Update code value*
⋆ *Update interval length as difference*

•

ALGORITHM 29

Procedure Decoder_Renormalization $(V, L, t, \mathbf{d})$

1. while $L < D^{P-1}$ do
   1.1. set { $t \leftarrow t + 1$;
   $V \leftarrow (D \cdot V) \bmod D^P + d(t)$;
   $L \leftarrow (D \cdot L) \bmod D^P$; }
2. return.

⋆ *Renormalization loop*
⋆ *Increment symbol counter*
⋆ *Update code value*
⋆ *Update interval length*

•

In Algorithm 7 we used sequential search for interval selection during decoding, which in the worst case requires testing $M - 1$ intervals. In Algorithm 28 we use bisection [17, 19, 39] for solving (1.16), which requires testing at most $\lceil \log_2(M) \rceil$ intervals (see Sections 2.2.3 and 2.3.2). Finally, the decoder renormalization is shown in Algorithm 29.

Bibliography

[1] C.E. Shannon, "A mathematical theory of communications," *Bell Syst. Tech. J.,* vol. 27, pp. 379–423, July 1948.

[2] D.A. Huffman, "A method for construction of minimum redundancy codes," *Proc. IRE,* vol. 40, pp. 1098–1101, 1952.

[3] S.W. Golomb, "Run-length encoding," *IEEE Trans. Inform. Theory,* vol. 12, no. 4, pp. 399–401, 1966.

[4] R.G. Gallager, *Information Theory and Reliable Communication,* John Wiley & Sons, New York, 1968.

[5] F. Jelinek, *Probabilistic Information Theory*, McGraw Hill, New York, NY, 1968.

[6] R. Pasco, *Source Coding Algorithms for Fast Data Compression,* Ph.D. Thesis, Dept. of Electrical Engineering, Stanford University, Stanford, CA, 1976.

[7] J.J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM J. Res. Develop.,* vol. 20, no. 3, pp. 198–203, May 1976.

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory,* vol. 23, no. 3, pp. 337–343, 1977.

[9] R.G. Gallagher, "Variations on a theme by Huffman," *IEEE Trans. Inform. Theory*, vol. 24, no. 6, pp. 668–674, Nov. 1978.

[10] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory,* vol. 24, no. 5, pp. 530–536, 1978.

[11] G.G. Langdon and J.J. Rissanen, *Method and Means for Arithmetic String Coding,* United States Patent 4,122,440, Oct. 1978.

[12] G.N.N. Martin, "Range encoding: an algorithm for removing redundancy from a digitized message," *Video and Data Recording Conf.,* Southampton, England, July 1979.

[13] F. Rubin, "Arithmetic stream coding using fixed precision registers," *IEEE Trans. Inform. Theory,* vol. 25, no. 6, pp. 672–675, Nov. 1979.

[14] J.J. Rissanen and G.G. Langdon, "Universal modeling and coding," *IEEE Trans. Inform. Theory,* vol. 27, pp. 12–23, Jan. 1981.

[15] G.G. Langdon and J.J. Rissanen, "Compression of black-white images with arithmetic coding," *IEEE Trans. Commun.,* vol. 29, pp. 858–867, June 1981.

[16] G.G. Langdon and J.J. Rissanen, *Method and Means for Arithmetic Coding Utilizing a Reduced Number of Operations,* United States Patent 4,286,256, Aug. 1981.

[17] P.Gill, M.H. Wright, and W. Murray, *Practical Optimization*, Academic Press, New York, 1982.

[18] G.G. Langdon and J.J. Rissanen, "A simple general binary source code," *IEEE Trans. Inform. Theory,* vol. 28, pp. 800–803, Sept. 1982.

[19] J.R. Rice, *Numerical Methods, Software and Analysis*, McGraw Hill, 1983.

[20] N.S. Jayant and P. Noll, *Digital Coding of Waveforms*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[21] R.J. McEliece, *The Theory of Information and Coding*, Cambridge University Press, New York, NY, 1984.

[22] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill Pub. Co., 1984.

[23] G.G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Develop.,* vol. 28, no. 2, pp. 135–149, March 1984.

[24] D.E. Knuth, "Dynamic Huffman coding," *J. of Algorithms,* vol. 6, pp. 163–180, June 1985.

[25] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Commun. ACM,* vol. 30, no. 6, pp. 520–540, June 1987.

[26] J.S. Vitter, "Design and analysis of dynamic Huffman codes," *J. ACM,* vol. 34, pp. 825–845, Oct. 1987.

[27] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon Jr., R.B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.,* vol. 32, no. 6, pp. 717–726, Nov. 1988.

[28] W.B. Pennebaker and J.L. Mitchell, "Probability estimation for the Q-coder," *IBM J. Res. Develop.,* vol. 32, no. 6, pp. 737–752, Nov. 1988.

[29] T.C. Bell, I.H. Witten, and J. Cleary, *Text Compression*, Prentice Hall, Feb. 1990.

[30] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[31] A. Moffat, "Linear time adaptive arithmetic coding," *IEEE Trans. Inform. Theory,* vol. 36, pp. 401–406, March 1990.

[32] T.M. Cover and J.A. Thomas, *Elements of Information Theory,* John Wiley & Sons, New York, 1991.

[33] International Telecommunication Union (ITU), *Digital Compression and Coding of Continuous-Tone Still Images: Requirements and Guidelines*, ITU-T Recommendation T.81, Geneva, Switzerland, 1992.

[34] P.G. Howard and J.S. Vitter, "Practical implementations of arithmetic coding," in *Image and Text Compression*, J.A. Storer, ed., Kluwer Academic Publishers, Boston, MA, 1992.

[35] A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, Boston, MA, 1992.

[36] W.B. Pennebaker and J.L. Mitchell, *JPEG: Still Image Data Compression Standard*, Von Nostrand Reinhold, New York, 1992.

[37] P.G. Howard and J.S. Vitter, "Analysis of arithmetic coding for data compression," *Inform. Proc. and Management*, vol. 28, no. 6, pp. 749–764, 1992.

[38] International Telecommunication Union (ITU), *Progressive Bi-Level Image Compression*, ITU-T Recommendation T.82, Geneva, Switzerland, 1993.

[39] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, 1993.

[40] J.M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients." *IEEE Trans. on Signal Processing*, vol. 41, pp. 3445–3462, Dec. 1993.

[41] M. Burrows and D.J. Wheeler, *A Block-sorting Lossless Data Compression Algorithm.* Digital Systems Research Center Tehnical Report 124, Digital Equipment Corporation, Palo Alto, CA, 1994.

[42] P.M. Fenwick, "A new data structure for cumulative frequency tables," *Softw. Pract. Exper.*, vol. 24, pp. 327–336, March 1994.

[43] A. Moffat, N. Sharman, I.H. Witten, and T.C. Bell, "An empirical evaluation of coding methods for multi-symbol alphabets," *Information Processing and Management,* 1994.

[44] A. Said and W.A. Pearlman, "Reduced-complexity waveform coding via alphabet partitioning," *Proc. IEEE Int. Symp. Information Theory*, Whistler, Canada, p. 373, Sept. 1995.

[45] M.J. Weinberger, J.J. Rissanen, and R.B. Arps, "Applications of universal context modeling to lossless compression of gray-scale images," *IEEE Trans. on Image Processing*, vol. 5, pp. 575–586, April 1996.

[46] X. Wu and N. Memon, "CALIC – a context based adaptive lossless image codec," *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, vol. 4, pp. 1890–1893, May 1996.

[47] A. Said and W.A. Pearlman, "A new fast and efficient image codec based on set partitioning in hierarchical trees," *IEEE. Trans. Circ. Syst. Video Tech.*, vol. 6, pp. 243–250, June 1996

[48] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, 1997.

[49] S. LoPresto, K. Ramchandran, and M. T. Orchard, "Image coding based on mixture modeling of wavelet coefficients and a fast estimation-quantization framework," *Data Compression Conference*, Snowbird, Utah, pp. 221–230, March 1997.

[50] A. Moffat, "Critique of the paper 'Novel design of arithmetic coding for data compression'," *IEE Proc. – Comput. Digit. Tech.*, vol. 144, pp. 394–396, Nov. 1997.

[51] A. Moffat, R.M. Neal, and I.H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256-294, 1998.

[52] M. Schindler, "A fast renormalisation for arithmetic coding," *Proc. IEEE Data Compression Conf.*, 1998.

[53] I.H. Witten, A. Moffat, and T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed., Morgan Kaufmann Publishers, 1999.

[54] P-C. Wu, "A byte-wise normalization method in arithmetic coding," *Softw. Pract. Exper.*, vol. 29, pp. 299–309, April 1999.

[55] K. Sayood, *Introduction to Data Compression*, 2nd ed., Morgan Kaufmann Publishers, 2000.

[56] D. Salomon, *Data Compression*, 2nd ed., Springer Verlag, New York, NY, 2000.

[57] I. Sodagar, B-B. Chai, and J. Wus, "A new error resilience technique for image compression using arithmetic coding," *Proc. Int. Conf. Acoustics, Speech and Signal Proc.,* vol. 4, pp. 2127–2130, June 2000.

[58] Texas Instruments Incorporated, *TMS320C6000 CPU and Instruction Set Reference Guide*, Literature Number: SPRU189F, Dallas, TX, 2000.

[59] International Business Machines Corporation: *PowerPC 750CX/CXe RISC Microprocessor Users Manual*, (preliminary edition), Hopewell Junction, NY, 2001.

[60] Intel Corporation, *Intel Pentium 4 Processor Optimization*, Reference Manual 248966, Santa Clara, CA, 2001.

[61] Sun Microsystems, Inc., *UltraSPARC III Technical Highlights*, Palo Alto, CA, 2001.

[62] D.S. Taubman and M.W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards and Practice*, Kluwer Academic Publishers, Boston, MA, 2002.

[63] A. Said, *Comparative Analysis of Arithmetic Coding Computational Complexity,* Hewlett-Packard Laboratories Report, HPL–2004–75, Palo Alto, CA, April 2004.