arXiv:2302.00819v1 [cs.IT] 2 Feb 2023



引言 算术编码——理论与实践[1]

Amir Said
成像系统实验室，惠普实验室，美国加利福尼亚州帕洛阿尔托

HPL-2004-76,
4月 21, 2004[2]

熵编码、压缩、复杂度

本算术编码入门分为两部分。第一部分解释算术编码的原理及其有效原因。我们先用非常一般的方式介绍它，以免其简洁性被层层实现细节所掩盖。接着，我们介绍它的一些基本性质，这些性质将在后文用于实际实现所需的计算技术。

第二部分介绍实际实现方面的内容，包括低精度算术运算、编码与建模的细分，以及自适应编码器的实现。我们还分析算术编码的计算复杂度，以及降低复杂度的技术。

有些章节会先介绍符号表示和大部分数学定义。读者起初若不明白这些内容的用意，也不必因此感到畏惧：后面总会有示例和说明。

[1]作为章节发表于《无损压缩手册》，编者：Khalid Sayood，©学术出版社[2]仅供内部访问，已获准对外发布

目录

1 算术编 码原理 1
1.1 数据压缩与算术编码.1 1.2 符号表示.2 1.3 码值.4 1.4 算术编码.5 1.4.1 编码过程.5 1.4.2 解码过程.10 1.5 算术编码的最优性.12 1.6 算术编码性质.14 1.6.1 动态信源.14 1.6.2 编码器与解码器的同步决策.14 1.6.3 编码与信源建模的分离.15 1.6.4 区间重缩放.15 1.6.5 近似算术.17 1.6.6 正确解码的条件.20

2 算术编 码实现 23
2.1 定精度算术编码.23 2.1.1 缓冲进位实现.25 2.1.2 整数运算实现.29 2.1.3 高效输出.32 2.1.4 进位处理注意事项.34 2.1.5 其他重归一化方法.35 2.2 自适应编码.36 2.2.1 符号分布的计算策略.36 2.2.2 累积分布的直接更新.37 2.2.3 二进制算术编码.39 2.2.4 基于树的累积分布更新.45 2.2.5 累积分布的周期性更新.47 2.3 复杂度分析.48 2.3.1 区间重归一化与压缩数据输入和输出 . . . . . 49 2.3.2 符号搜索 .50 2.3.3 累积分布估计 .54 2.3.4 算术运算 .55 2.4 延伸阅读 .56

第 1 章

算术编码原理

1.1 数据压缩与算术编码

压缩应用采用多种多样的技术，复杂度各不相同，但都包含一些共同的处理过程。图 1.1 展示了数据压缩中常见处理过程的示意图。这些处理过程取决于数据类型，图 1.1 中的模块也可能以不同顺序排列或组合使用。数值处理（如预测编码和线性变换）通常用于波形信号，例如图像和音频 [20, 35, 36, 48, 55]。逻辑处理则是将数据转换为更适合压缩的形式，例如游程、零树、集合分割信息和字典条目 [3, 20, 38, 40, 41, 44, 47, 55]。下一阶段是信源建模，用于处理数据统计特性的变化。它负责收集统计数据并识别数据上下文，从而使源模型更加准确可靠 [14, 28, 29, 45, 46, 49, 53]。

大多数压缩系统有一个共同点：最后一个处理环节都是熵编码，即以最紧凑的形式表示信息。它可能承担大部分压缩工作，也可能只是补充前序阶段已经完成的工作。

在考察各种熵编码方法及其在压缩应用中的潜在用途时，算术编码以其优雅、高效和通用性脱颖而出，因为它能够在种类最多的情形和用途下实现最高效的编码。它最令人称道的特点包括以下几点。

- 对于独立同分布的信源，算术编码能够以可证明的最优方式对每个符号进行压缩（第 1.5 节）。
- 它适用于各种情形和压缩比。同一种算术编码实现，就能有效地对图 1.1 所示不同处理过程产生的各种数据进行编码，例如建模参数、变换系数、信令等（第 1.6.1 节）。
- 它简化了复杂信源的自动建模，使非独立同分布（i.i.d.）信源的压缩达到近乎最优的效果或获得显著提升（第 1.6.3 节）。

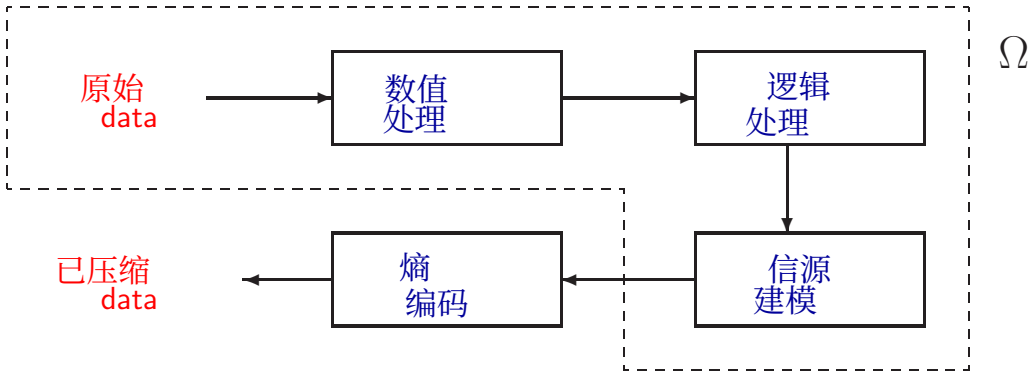

图 1.1：数据压缩的典型处理流程系统。算术编码通常是最后一个阶段，其他阶段可以建模为一个单一的数据源 Ω。

- 其主要运算是算术运算，所有通用处理器或数字信号处理器（中央处理器、数字信号处理器）对这一运算的支持效率都在不断提高（第 2.3 节）。
- 它适合不懂编码或不想自行实现编码算法的人，作为“压缩黑箱”使用。

尽管具有这些优点，算术编码仍不如其他方法普及，也鲜为人知。一些实际问题阻碍了它的应用。

- 算术运算的复杂度对于编码应用而言过高。
- 最有效的实现受到专利保护。专利使用费以及对专利侵权的担忧，阻碍了算术编码在商业产品中的应用。
- 高效的实现难以理解。

不过，这些问题现在大多已经得到解决。首先，计算机算术的相对效率大幅提升，新的技术也避免了开销最大的运算。其次，一些专利已经到期（例如 [11, 16]），或已过时。最后，我们不必过分关注那些会掩盖该方法内在简洁性的复杂度优化细节。如今的计算资源使我们能够实现简单、高效且免专利使用费的算术编码。

1.2 符号表示

设 Ω 为一个数据源，它输出的符号 s_k 编码为集合$\{0, 1, \ldots, M-1\}$中的整数，并设 $S = \{s_1, s_2, \ldots, s_N\}$ 为 Ω [1, 4, 5, 21, 55, 56]输出的 N随机符号序列。目前，我们假设源符号是独立同分布的 [22], ，其概率为

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

我们还假设，对于所有符号都有 $p(m) \neq 0$，并定义 $c(m)$ 为累积分布，

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

注意，$c(0) \equiv 0$，$c(M) \equiv 1$，并且

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

我们用粗体字母表示所有 $p(m)$ 和 $c(m)$ 值的向量，即

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

我们假设压缩数据（编码器的输出）保存在向量（缓冲区）$\mathbf{d}$ 中。输出字母表有 D 个符号，即 $\mathbf{d}$ 中的每个元素都属于集合 $\{0, 1, \ldots, D-1\}$。

在上述假设下，最优编码方法 [1] 对 Ω 中的每个符号 s 进行编码，平均所需比特数为

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

Example 1

◁ 数据源 Ω 可以是一个英语文本文件：该信源中的每个符号都是表示一个字符的单字节。此数据字母表包含 $M = 256$ 个符号，符号编号由 ASCII 标准定义。可以通过收集大量英语文本的统计数据来估计各符号的概率。表 1.1 展示了一些字符、其 ASCII 符号值以及估计概率。表中还展示了以最优方式编码符号 s 所需的比特数，$-\log_2 p(s)$。由这些数字可知，如果英语文本中的数据符号是独立同分布的，那么可能达到的最佳文本压缩比约为 2:1（4 比特/符号）。专用文本压缩方法 [8, 10, 29, 41] 能实现显著更高的压缩比，因为它们利用了字母之间的统计依赖。 ▷

这个例子首先表明，我们对数据源的初始假设在实际情况中很少成立。更常见的是，我们会遇到以下问题。

1. 源符号并非服从相同的分布。
2. 数据序列中的符号并非相互独立（即使它们不相关）[22]。
3. 我们只能估计概率值、符号之间的统计依赖关系，以及这些因素随时间的变化情况。

不过，在接下来的章节中，我们将说明，将算术编码推广到时变数据源是很直接的，并解释如何处理所有这些实际问题。

字符	ASCII 符号 s	概率 $p(s)$	最优数量 比特的 –对数$_2$ $p(s)$
空格	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

表 1.1：英语中一些字母和标点符号的概率估计。符号按照 ASCII 标准编号。

1.3 编码值

算术编码不同于其他编码方法；对于那些方法，我们知道编码符号与实际写入文件的比特之间的确切关系。算术编码一次编码一个数据符号，并为每个符号分配一个实数数量的比特（参见表 1.1 最后一列中的示例）。要弄清楚这是如何实现的，我们需要理解码值表示：将编码后的消息映射到区间 $[0, 1)$ 内的实数。

压缩数据序列的编码值 v 是一个实数，其小数位等于该序列中的符号。只需在编码序列开头加上“0.”，再将结果解释为以 D 为基数的数，即可将序列转换为编码值，其中 D 是编码序列字母表中的符号数量。例如，如果一种编码方法生成的比特序列为 0011000101100，那么我们有

$$\begin{aligned} &\text{Code sequence} \quad \mathbf{d} = \quad [\underbrace{0011000101100}] \\ &\text{Code value} \qquad\quad v = \quad 0.\overbrace{0011000101100}_2 \quad = 0.19287109375 \end{aligned} \tag{1.5}$$

其中下标“2”表示二进制表示法。与通常做法一样，十进制表示法省略下标。

这种构造在由 D 个符号组成的字母表所形成的无限符号序列与区间 $[0, 1)$ 内的实数之间建立了一种便捷的映射：任何数据序列都可以用一个实数表示，反之亦然。码值表示可用于任何编码系统，并提供了一种通用方式来表示大量

信息，而不受编码所用符号集（二进制、三进制、十进制等）的影响。例如，在 (1.5) 中，我们可以看到同一编码的二进制和十进制表示。

我们可以通过分析压缩方法生成的编码值分布，来评估该方法的效能。根据香农信息论 [1]，如果一种编码方法是最优的，那么其编码值的累积分布 [22] 必定是从点 (0, 0) 到点 (1, 1) 的一条直线。

Example 2

◁ 假设独立同分布信源 Ω 有四个符号，数据符号的概率为 $\mathbf{p} = [0.65 \;\; 0.2 \;\; 0.1 \;\; 0.05]$。如果我们用每个符号两比特的方式对来自该信源的随机数据序列进行编码，所得编码值将形成如图 1.2 中“未压缩”标签所示的累积分布。请注意，该分布呈偏斜状，表明有可能实现显著压缩。

同样的序列也可以使用 Ω 的霍夫曼编码 [2, 4, 21, 55, 56],进行编码，其中符号“0”使用一比特，符号“1”使用两比特，符号“2”和“3”使用三比特。图 1.2 中相应的编码值累积分布表明，与未压缩的情况相比已有显著改进，但这种编码方法显然仍非最优。图 1.2 中的第三条线显示，通过算术编码模拟压缩后的序列，其编码值分布与最优分布几乎完全相同。 ▷

直线型分布意味着，如果一种编码方法是最优的，那么压缩序列中就不再存在统计依赖或冗余，因此其编码值在区间 $[0, 1)$ 上均匀分布。这一事实对于理解算术编码的工作原理至关重要。此外，编码值是算术编码／解码过程不可或缺的一部分，其中的算术运算作用于与编码值直接相关的实数。

关于编码值，最后补充一点：两个无限长且不同的序列可能对应同一个编码值。这是因为对于任意 $D > 1$，都有

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

例如，如果 $D = 10$ 和 $k = 2$，那么 (1.6) 就是等式 $0.09999999\ldots = 0.1$。这一事实对于编码而言并无重要的实际意义，但在研究算术编码的某些理论性质时，我们需要将其考虑在内。

1.4 算术编码

1.4.1 编码过程

本节首先介绍描述算术编码的符号表示和方程，随后给出一个详细示例。从根本上说，算术编码过程包括创建一系列嵌套区间，其形式为 $\Phi_k(S) = [\alpha_k, \beta_k) \quad k = 0, 1, \ldots, N$,

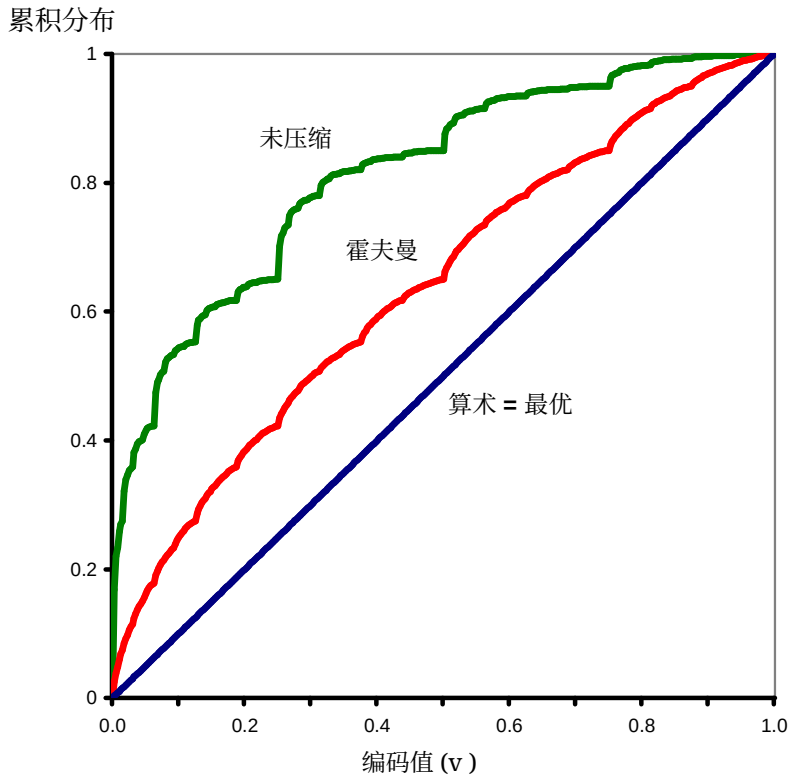

图 1.2：将不同编码方法应用于例2的信源时所生成码值的累积分布。

其中，S 是源数据序列，α_k 和 β_k 是满足 $0 \le \alpha_k \le \alpha_{k+1}$ 的实数，且 $\beta_{k+1} \le \beta_k \le 1$。为了更简洁地描述算术编码，我们用 $|\, b,\, l\,\rangle$ 的形式表示区间，其中 b 称为区间的基数或起点，l 为区间的长度。传统区间表示法与新表示法之间的关系为

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

在这种新符号表示法下，算术编码过程中使用的区间由以下递归方程组定义 [5, 13]

$$\Phi_0(S) \;=\; |\, b_0,\, l_0\,\rangle = |\, 0,\, 1\,\rangle\,, \tag{1.8}$$
$$\Phi_k(S) \;=\; |\, b_k,\, l_k\,\rangle = |\, b_{k-1} + c(s_k)\, l_{k-1},\; p(s_k)\, l_{k-1}\,\rangle\,, \quad k = 1, 2, \ldots, N. \tag{1.9}$$

这些区间的性质保证了 $0 \le b_k \le b_{k+1} < 1$，并且 $0 < l_{k+1} < l_k \le 1$。图 1.3 展示了与递归方程组 (1.9) 相对应的动态系统。我们将在后文说明，如何在编码过程结束时，从最终区间中选择一个编码值，即 $\hat{v}(S) \in \Phi_N(S)$。

由 (1.8) 和 (1.9) 定义的编码过程，也称为埃利亚斯编码，最早由 [5] 描述。我们采用基数和长度来表示区间的约定，这种做法一直沿用

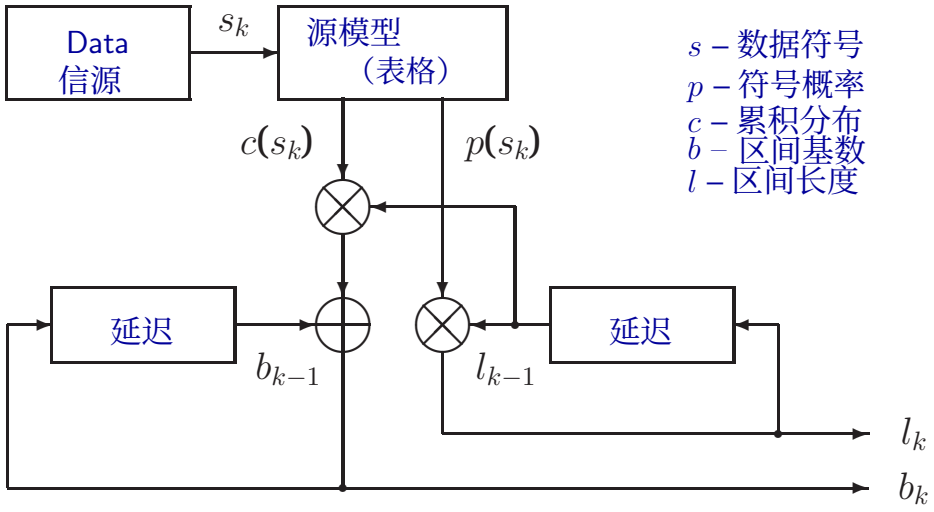

图 1.3：用于更新算术编码区间的动态系统。

至最早的算术编码论文 [12, 13]。其他作者则用区间的两个端点表示区间，例如 [基数、基数+长度），但这两种符号表示在数学上并无区别。

Example 3

◁ 假设信源 Ω 有四个符号（$M = 4$），这些符号的概率和分布分别为 $\mathbf{p} = [0.2 \;\; 0.5 \;\; 0.2 \;\; 0.1]$ 和 $\mathbf{c} = [0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1]$，要编码的符号序列（$N = 6$）为 $S = \{2, 1, 0, 0, 1, 3\}$。

图1.4以图示方式展示了编码过程如何对应于实数轴上区间的选择。我们从图的顶部开始，区间为 $[0, 1)$，它被划分为四个子区间，每个子区间的长度等于相应数据符号的概率。具体而言，区间 $[0, 0.2)$ 对应于 $s_1 = 0$，区间 $[0.2, 0.7)$ 对应于 $s_1 = 1$，区间 $[0.7, 0.9)$ 对应于 $s_1 = 2$，最后，区间 $[0.9, 1)$ 对应于 $s_1 = 3$。下一组允许的嵌套子区间的长度也与符号的概率成正比，但其长度还与所属区间的长度成正比。此外，这些子区间表示的不止一个符号值。例如，区间 $[0, 0.04)$ 对应于 $s_1 = 0$, $s_2 = 0$，区间 $[0.04, 0.14)$ 对应于 $s_1 = 0$, $s_2 = 1$，依此类推。

区间长度会按照符号概率相应缩小，从而获得在区间 $[0, 1$ 中均匀分布的编码值（这是最优性的必要条件，正如第1.3节所述）。例如，如果20%的序列以符号“0”开头，那么20%的编码值必须位于分配给这些序列的区间内；而要实现这一点，只能为第一个符号“0”分配一个长度等于其概率的区间，即0.2。对子区间长度的分配也适用同样的原理：每次出现符号“0”，都必须将区间长度缩小到其当前长度的20%。这样，在编码了

迭代 k	输入符号 s_k	区间 base b_k	区间长度 l_k	解码器更新后的值 $\tilde{v}_k = \frac{\hat{v}-b_{k-1}}{l_{k-1}}$	输出符号 $\hat{s}_k$
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

表 1.2：示例 3 和 4 的算术编码与解码结果。最后两行展示了在最后一个符号之后继续解码时会发生什么。

若干个符号后，编码值的分布应当非常接近均匀分布。

公式 (1.8) 和 (1.9) 给出了区间顺序计算的公式。将它们应用于我们的示例，可得：

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

所有编码器区间的列表列于表 1.2 的前四列中。由于区间很快就会变得非常小，因此在图1.4中，我们必须将它们（两次）放大显示，才能看清编码过程如何继续。请注意，尽管区间以不同的放大倍数显示，但区间的数值并未改变，区间细分过程仍以完全相同的方式继续。 ▷

算术编码的最后一项任务是定义一个编码值 $\hat{v}(S)$，用于表示数据序列 S。在下一节中，我们将展示，对于任意编码值 $\hat{v} \in \Phi_N(S)$，解码过程都能正确运行。然而，不能将编码值作为纯实数提供给解码器。它必须使用常规的数值表示进行存储或传输。由于我们可以在最终区间内任意选择一个值，因此希望选择表示形式最短的值。例如，在示例 3 中，选择 $\hat{v} = 0.7427$ 可得到最短的十进制表示，而选择 $\hat{v} = 0.10111110001_2 = 0.74267578125$ 则可得到最短的二进制表示。

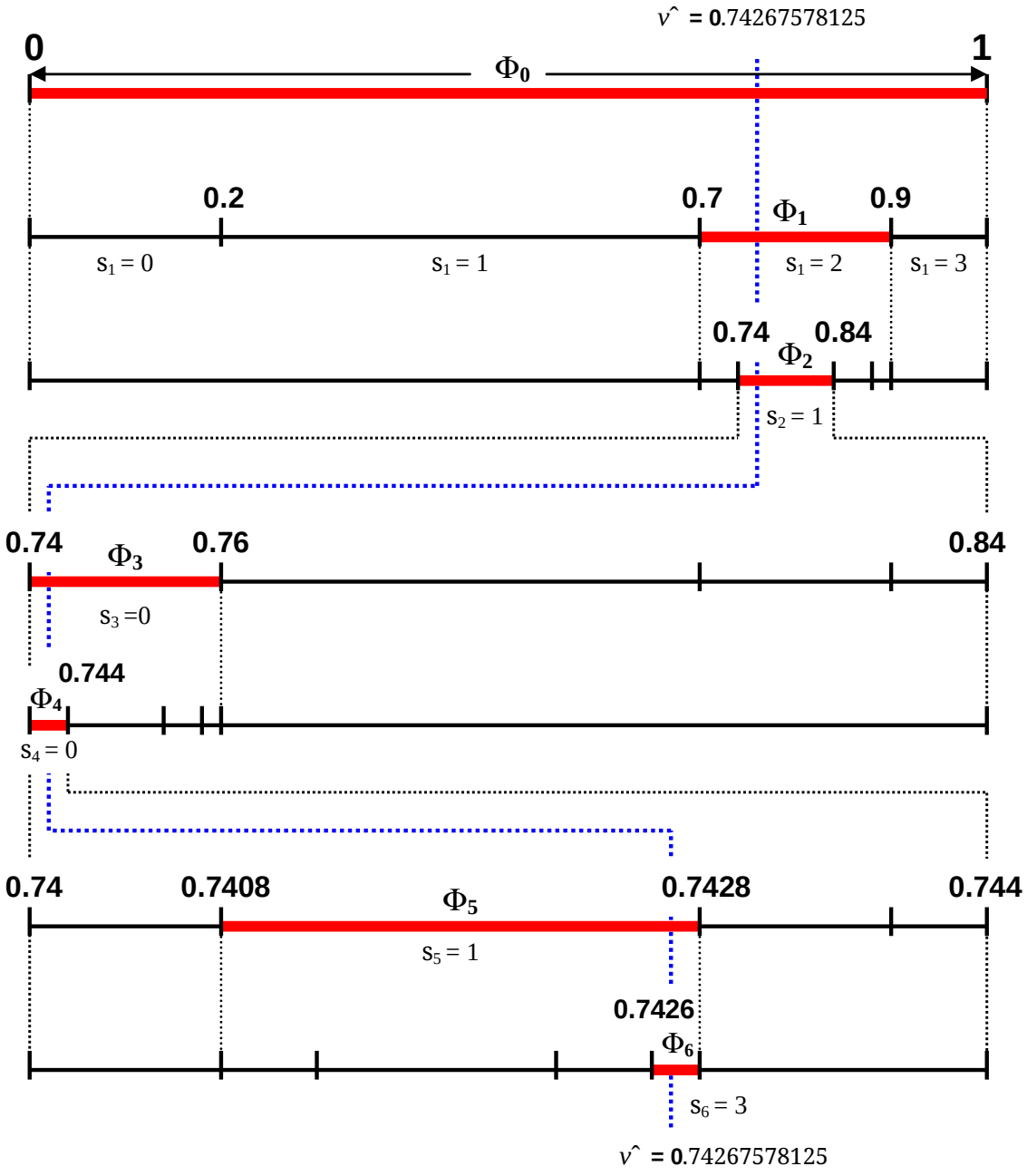

图1.4：示例 3 的算术编码过程图示：区间 $\Phi_0 = [0, 1$ 根据数据符号的概率被划分为嵌套区间。与数据序列 $S = \{2, 1, 0, 0, 1, 3\}$ 对应的所选区间以较粗的线条表示。

寻找最佳二进制表示的过程相当简单，用归纳法来说明最为合适。主要思路是，对于相对较大的区间，我们可以通过测试少量二进制序列来找到最优值；随着区间长度减半，需要测试的序列数量必须加倍，因此比特数增加一位。因此，根据区间长度 l_N，我们采用以下规则：

- 如果 $l_N \in [0.5,\ 1)$，则选择编码值 $\hat{v} \in \{0, 0.5\} = \{0.0_2, 0.1_2\}$，采用 1 比特表示。

- 如果 $l_N \in [0.25,\ 0.5)$，则选择值 $\hat{v} \in \{0, 0.25, 0.5, 0.75\} = \{0.00_2, 0.01_2, 0.10_2, 0.11_2\}$，采用2比特表示。

- 如果 $l_N \in [0.125,\ 0.25)$，那么对于 3比特表示，选择值 $\hat{v} \in \{0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875\} = \{0.000_2, 0.001_2, 0.010_2, 0.011_2, 0.100_2, 0.101_2, 0.110_2, 0.111_2\}$。

观察这一规律后，我们得出结论，表示 $\hat{v} \in \Phi_N(S)$ 所需的最少比特数为

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

其中，$\lceil x \rceil$ 表示大于或等于 x 的最小整数。

我们可以通过观察表 1.2 中示例 3 的结果来检验这一结论。最终区间为 $l_N = 0.0002$，因此 $\mathrm{B}_{\min} = \lceil -\log_2(0.0002) \rceil = 13$ 比特。然而，在示例 3 中，我们可以选择 $\hat{v} = 0.10111110001_2$，而且只需 11 比特！

这种不一致的根源在于，我们可以选择位数由 (10) 给出的二进制表示，然后去掉尾随零。不过，采用最优编码时，这种做法平均只能节省一比特，因此在实践中很少使用。

1.4.2 解码过程

在算术编码中，解码序列完全由压缩序列的编码值 $\hat{v}$ 决定。因此，我们将解码序列表示为

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

现在，我们将介绍解码过程，说明如何使用任意编码值 $\hat{v} \in \Phi_N(S)$ 解码出正确的序列（即 $S(\hat{v}) = S$）。我们先给出实现解码的递归方程，然后通过一个实际示例，直观说明解码过程的工作原理及其正确性。

解码过程会按照数据符号被编码时的相同顺序恢复它们。具体而言，为了求得数值解，我们定义一个归一化码值序列 $\{\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_N\}$。从 $\tilde{v}_1 = \hat{v}$ 开始，我们依次根据 $\tilde{v}_k$ 求出 $\hat{s}_k$，然后根据 $\hat{s}_k$ 和 $\tilde{v}_k$ 计算 $\tilde{v}_{k+1}$。

递推公式为

$$\begin{aligned}
\tilde{v}_1 &= \hat{v}, && (1.12)\\
\hat{s}_k(\hat{v}) &= \{\, s :\ c(s) \le \tilde{v}_k < c(s+1)\}, \quad k = 1, 2, \ldots, N, && (1.13)\\
\tilde{v}_{k+1} &= \frac{\tilde{v}_k - c(\hat{s}_k(\hat{v}))}{p(\hat{s}_k(\hat{v}))}, \quad k = 1, 2, \ldots, N-1. && (1.14)
\end{aligned}$$

（在公式（1.13）中，冒号表示“满足这些不等式的 s ”。）

一种在数学上等价的解码方法——我们稍后将证明，在使用定精度算术时这是必不可少的——会还原编码器生成的区间序列，并在每个区间中搜索正确的值 $\hat{s}_k(\hat{v})$。其定义如下：

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

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

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

将递归式 (1.14) 与递归式 (1.17) 结合，可得

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

这表明 (1.13) 与 (1.16) 等价。

Example 4

◁ 我们将对示例 3 中得到的数据应用解码过程。在图1.4中，我们以图示方式展示了 $\hat{v}$ 的含义：它是编码过程中形成的所有嵌套区间都包含的一个值。虚线表明，随着我们放大图形，它的位置会移动，但该值本身保持不变。从图1.4可以看出，我们可以从第一个区间 $\Phi_0(S) = [0,\ 1)$ 开始解码：只需将 $\hat{v}$ 与累积分布 c 进行比较，就能找到 $\hat{s}_1$ 唯一可能的取值

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

我们可以利用 $\hat{s}_1$ 的值来找到区间 $\Phi_1(S)$，并用它来确定 $\hat{s}_2$。实际上，我们可以通过定义归一化码值，来“消除” $\hat{s}_1$ 对 $\hat{v}$ 的影响

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

请注意，一般来说，$\tilde{v}_2 \in [0,\ 1)$，也就是说，它是相对于初始区间归一化后的值。在这个区间中，我们可以使用相同的过程来找到

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

表 1.2 最后几列展示了这一过程如何继续，以及解码过程中计算出的更新后的值。我们可以说，这一过程会一直持续到解码出 $\hat{s}_6$ 为止。然而，解码器只有初始编码值 $\hat{v}$，怎么知道该在何时停止解码呢？答案很简单：它无法知道。我们在表 1.2 中增加了两行，以展示在编码出最后一个符号之后，解码过程仍可照常继续。下面我们将解释其中的情况。 ▷

重要的是要理解，算术编码将区间映射为一组序列。区间中的每个实数都对应一个无限序列。因此，与 $\Phi_6(S) = [0.7426, 0.7428)$ 对应的序列，都是以 $\{2, 1, 0, 0, 1, 3, \ldots\}$ 开头的序列。编码值 $\hat{v} = 0.74267578125$ 对应其中一个这样的无限序列，而解码过程可以永远继续，对该特定序列进行解码。

有两种实用的方法可以告知解码应当停止：

1. 在压缩文件开头注明数据符号的数量（N）。

2. 使用一个特殊符号作为“消息结束符”，仅在数据序列末尾对其进行编码，并为该符号分配编码器/解码器允许的最小概率值。

如上所述，解码过程总会生成一个解码数据序列。然而，我们如何知道它是正确的序列呢？我们可以根据以下事实推断：如果 S 和 S' 是包含 N 个符号的序列，那么

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

这保证了不同的序列不会产生相同的编码值。在第 1.6.6 节中，我们将说明，由于近似值，如果（1.19）不成立，就会出现解码错误。

1.5 算术编码的最优性

信息论 [1, 4, 5, 21, 32, 55, 56] 告诉我们，对平稳无记忆信源 Ω 中的每个符号进行编码所需的平均比特数不能小于其熵 $H(\Omega)$，熵定义为

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

我们已经看到，算术编码过程会在区间 $[0, 1)$ 内生成均匀分布的码值。这是实现最优性的必要条件，但并非充分条件。在区间 $\Phi_N(S)$ 内，我们可以选择需要任意多比特才能表示的值，也可以选择能够用最少比特表示的码值，如公式 (1.10) 所示。现在我们将证明，后一种选择满足最优性的充分条件。

首先，我们必须考虑压缩文件中会有一些开销，其中可能包括

- 保存 $\hat{v}$ 所需的额外比特数，按整数个字节计。
- 一个固定的或可变数量的比特，用于表示已编码符号的数量。
- 关于概率的信息（p 或 c）。

假设总开销为正数 σ 比特，我们由（1.10）得出，用于编码一个序列的每个符号所需的比特数 S 应满足以下界限：

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

由 (1.9) 可得

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

因此

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

将 $E\{\cdot\}$ 定义为期望值算子，则每个符号的期望比特数为

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

由于每个符号的平均比特数不能小于熵，因此有

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

由此可得

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

这意味着算术编码确实实现了最优压缩性能。

此时我们可能会问，为什么算术编码生成的是区间，而不是单个编码值。答案在于，算术编码不仅对二进制输出是最优的，对任何输出字母表都是最优的。在最终区间中，我们可以找到适用于每种输出字母表的最优码值。下面举例说明如何使用非二进制输出。

Example 5

◁ 考虑使用一种通信系统传输示例 3 中的数据序列，该系统使用三个电平传递信息：{–V、0、 +V}（实际用于无线电遥控器）。采用三进制输出的算术编码可以同时压缩数据，并将其转换为合适的传输格式。

D符号输出字母表的 (1.10) 的一般化形式为

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

因此，根据表 1.2 中的结果，我们得出需要 $\lceil -\log_3(0.0002) \rceil = 8$ 个三进制符号。我们稍后会说明如何使用标准算术编码，得出最短的三进制表示为 $\hat{}$ $v_3 = 0.20200111_3 \approx 0.742722146$，这意味着序列 $S = \{2, 1, 0, 0, 1, 3\}$ 可以作为以下电信号序列传输：{+V, 0, +V, 0, 0, –V, –V,–V}。 ▷

1.6 算术编码的性质

1.6.1 动态信源

在第1.2节中，我们假设数据源 Ω 是平稳的，因此，对数据序列 S 中的所有符号进行编码和解码时，使用同一组符号概率。现在，了解了编码过程之后，我们将其推广到每个待编码符号的概率都会变化的情况，即数据序列 S 中的第 k 个符号是一个随机变量，其概率为 p_k，分布为 c_k。

算术编码过程中唯一需要做的改动是：区间更新时不再使用.....，而应使用 (1.9)

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

要理解解码过程中的变化，请记住，使用更新后的码值进行操作，相当于“抹去”关于过去符号的所有信息，并在 $[0, 1)$ 区间内进行解码。因此，解码器只需为该符号使用正确的概率集，就能正确解码。对 (1.16) 和 (1.17) 做相应修改后，得到

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

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

请注意，每个时刻使用的符号数量都可能发生变化。我们不再只有一个包含 M 个符号的输入字母表，而是有一个字母表大小序列 $\{M_1, M_2, \ldots, M_N\}$。

1.6.2 编码器与解码器的同步决策

在数据压缩中，编码器可以在编码数据序列的过程中改变其行为（参数、编码算法等），只要解码器使用相同的信息和相同的规则来改变其行为即可。此外，这些变化必须是“同步的”，但同步的不是时间，而是数据源符号序列。

例如，在第 1.6.1 节中，我们假设编码器和解码器在使用变化的概率集合时是同步的。请注意，我们不必假设解码器开始解码时就能获得所有概率。只要概率向量的更新规则基于符号出现，便可以按任意规则更新；只要 p_k 是根据解码器已经获得的数据计算得出的，即 $\{\hat{}s_1、\hat{}s_2、\ldots、\hat{}s_{k-1}\}$。这一原则用于自适应编码，并在第 2.2 节中介绍。

同步这一概念对算术编码至关重要，因为算术编码涉及一个非线性动态系统（图 1.3），而误差累积会导致解码错误，除非编码器和解码器使用完全相同的实现（相同的精度、比特数、舍入规则、方程、表格等）。换句话说，即使编码器采用粗略的近似值，只要解码器采用完全相同的近似值，算术编码仍能正确运行。我们已经看到一个基于数值稳定性的选择示例：方程 (1.16) 和 (1.17) 使编码器和解码器能够保持同步，因为它们采用了与 (1.9) 相同的区间更新规则，而 (1.13) 和 (1.14) 则采用了不同的递归方式。

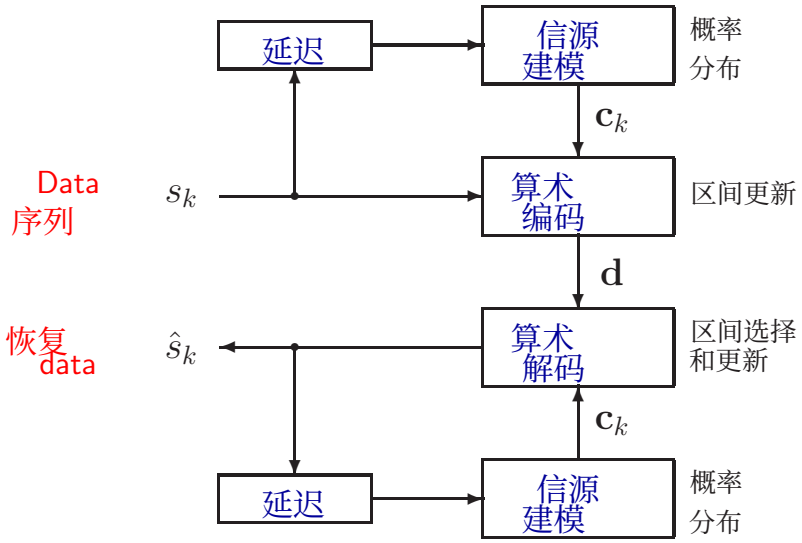

图 1.5：编码与信源建模任务的分离。算术编码和解码过程处理区间，而信源建模则为每个数据符号选择概率分布。

1.6.3 编码与信源建模的分离

将信源建模（概率估计）与编码过程分开有许多优点 [14, 25, 29, 38, 45, 51, 53]。例如，这样我们就可以开发复杂的压缩方案，而不必担心编码算法的具体细节；也可以将这些方案用于不同的编码方法和实现。

图 1.5 展示了如何在完整的算术编码与解码系统中将这两个过程分离。编码部分只负责更新区间，即算术编码器实现递归式 (1.28)，算术解码器实现式 (1.29) 和 (1.30)。编码／解码过程将概率分布向量作为输入，但不会以任何方式改变它们。信源建模部分负责选择分布 $\mathbf{c}_k$，用于对符号 s_k 进行编码／解码。图 1.5 还表明，在信源建模模块之前延迟一个数据符号，可确保编码器和解码器使用相同的信息来更新 $\mathbf{c}_k$。

算术编码大大简化了图 1.5 所示系统的实现，因为向量 $\mathbf{c}_k$ 可直接用于编码。对于霍夫曼编码，概率发生变化时，需要重新计算最优码，或者使用复杂的码更新技术 [9, 24, 26]。

1.6.4 区间重缩放

图1.4以图示方式展示了算术编码的一项重要特性：编码过程中实际使用的区间取决于初始区间和先前编码的数据，但细分区间内的比例并不受影响。例如，如果我们将初始区间改为 $\Phi_0 = |1, 2\rangle = [1, 3)$ 并应用 (1.9)，编码过程保持不变，只是

所有区间都按两倍缩放，并平移一个单位。

我们也可以在编码过程的中途进行重新缩放。假设在某个阶段 m，我们按照以下方式改变区间

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

并继续正常进行编码过程（使用（1.9）或（1.28））。编码完成后，我们得到区间 $\Phi'_N(S) = |\, b'_N,\ l'_N \,\rangle$ 以及相应的编码值 v'。我们可以使用以下方程，恢复在不进行重新缩放的情况下本应得到的区间和编码值：

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

解码器需要原始编码值 $\hat{v}$ 才能开始恢复数据符号。它还应在阶段 m 对区间进行重新缩放，因此需要知道 m、δ、γ。此外，当它使用（1.31）对区间进行缩放时，也必须使用以下方式对编码值进行缩放：

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

我们可以将上述结果推广到阶段 $m \le n \le \ldots \le p$ 的重新缩放。一般而言，包括对编码值进行缩放在内的缩放过程为

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

在编码过程结束时，我们得到区间 $\bar{\Phi}_N(S) = \left|\, \bar{b}_N,\, \bar{l}_N \,\right\rangle$ 和编码值 $\bar{v}$。我们使用以下方法恢复原始值：

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

and

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

这些公式看起来可能极其复杂，但在某些特殊情况下，使用起来相当简单。例如，在第 2.1 节中，我们将展示如何使用 $\delta_i \in \{0, 1/2\}$ 和$\gamma_i \equiv 2$进行缩放，并说明 δ_i 与 b_N 和 $\hat{v}$的二进制表示之间的联系。下一个示例将展示区间重缩放的另一种简单应用。

Example 6

◁ 图1.6 展示了应用于示例 3 的重新缩放。这与图1.4 非常相似，但图1.6 中的区间不仅仅是放大显示的小区间，它们本身也会发生变化。在编码两个

符号后使用重缩放参数 $\delta_2 = 0$ 和 $\gamma_2 = 25$，编码另外两个符号后使用 $\delta_2 = 0$ 和 $\gamma_2 = 25$。最终区间为 $\Phi_6(S) = |\,0.65,\ 0.05\,\rangle$，它对应于

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

这正是示例 3 中得到的区间。 ▷

1.6.5 近似算术

要理解如何以定精度实现算术编码，我们应该注意到，加法和乘法的要求截然不同。我们将说明，如果愿意牺牲一些压缩效率，就不需要精确乘法。我们用双括号 ($[[\cdot]]$) 标记乘法，以表示这是近似运算，即 $[[\alpha \cdot \beta]] \approx \alpha \cdot \beta$。我们将截断定义为满足 $[[\alpha \cdot \beta]] \leq \alpha \cdot \beta$ 的任何近似运算。这里讨论的近似运算可以是舍入，也可以是截断到任意精度。下面的示例展示了另一种理解不精确乘法的方法。

Example 7

◁ 我们可以从图 1.3 中看到，算术编码中的乘法总是使用源模型中的数据——概率 p 和累积分布 c。假设我们有 $l = 0.04$、$c = 0.317$ 和 $p = 0.123$，其中

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

我们可以不使用精确乘法，而改用近似值（例如通过查表和短寄存器），使得

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

现在，假设我们没有使用 p and c，而是使用了另一个模型，其中为 $c' = 0.3$ 和 $p' = 0.12$。我们将得到

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

这恰好就是采用近似乘法得到的结果。这说明，不精确乘法在数学上等价于先对源模型作近似处理，再使用精确乘法。 ▷

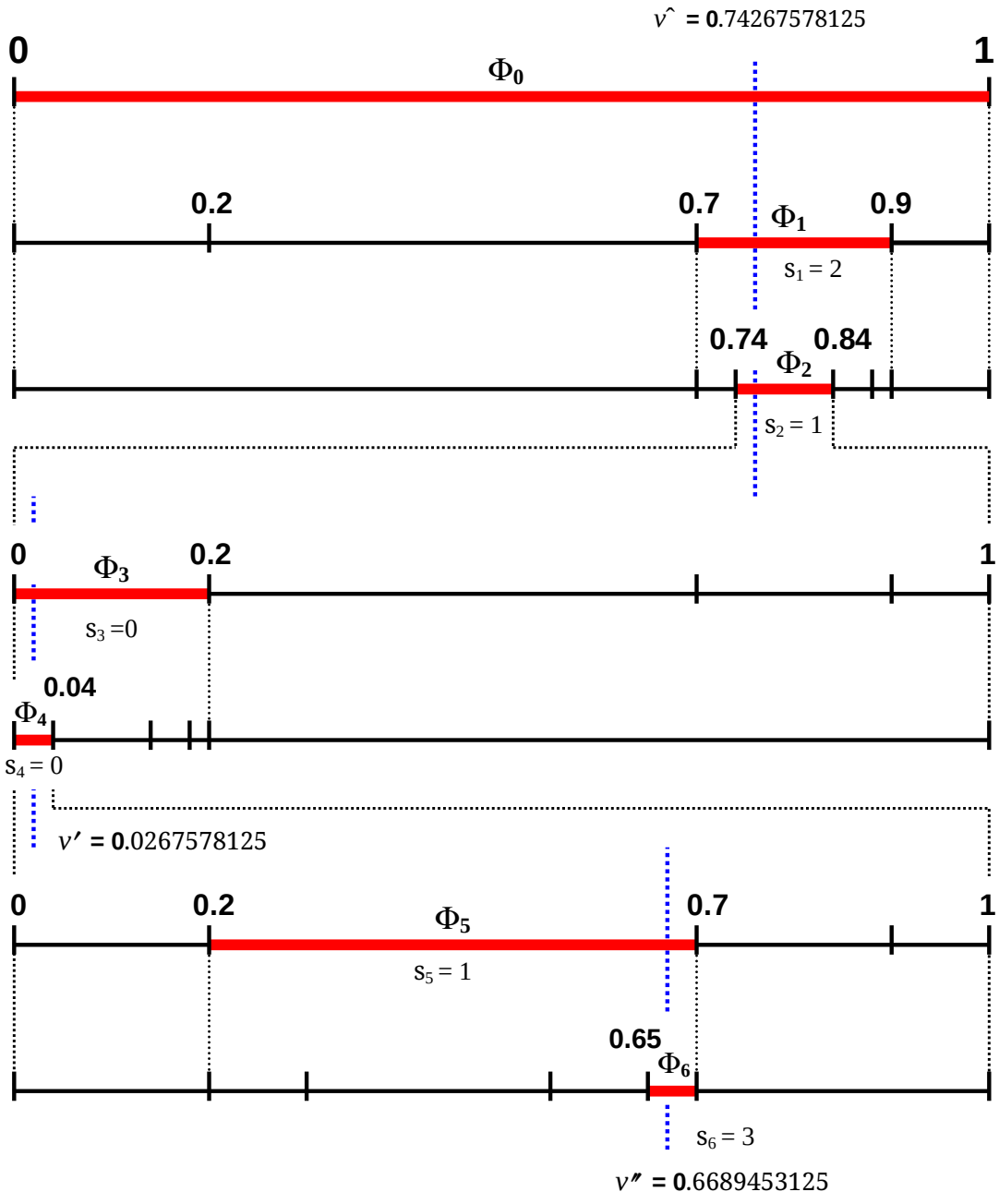

图1.6：示例 3（图1.4）的算术编码过程采用数值重缩放的图形表示。请注意，每次区间重新缩放时，编码值都会发生变化。

从这个例子中我们看到，无论乘法采用何种近似，我们始终可以假设一直进行精确乘法，只是使用不精确的分布。只要解码器与编码器保持同步，我们就不必担心分布值是否精确；也就是说，如果解码器采用的近似与编码器完全相同，那么编码器和解码器的分布必定相同（这与动态信源的情况相同，详见第 1.6.1 节）。

采用不精确乘法的（1.9）式为

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

我们还必须将（1.16）式和（1.17）式替换为

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

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

下一节将说明，近似乘法必须满足哪些条件才能实现正确解码。

在（1.37）至（1.39）式中，一种近似来自区间长度与累积分布的乘法，另一种近似则来自区间长度与概率的乘法。如果我们只想使用一种近似，并避免区间长度与概率之间的乘法，就应按以下方式更新区间长度：

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

不精确算术付出的代价是压缩性能下降。只有当源模型概率与真实的数据符号概率相等时，算术编码才是最优的；任何差异都会降低压缩比。

通过快速分析，我们可以大致了解损失会有多大。如果在概率为 p 的信源中使用概率值为 p′ 的模型，压缩的平均损失为

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

这个公式与相对熵 [32]，类似，但在此情况下，p′ 表示近似值所产生的数值，并且可能出现 $\sum_{n=0}^{M-1} p'(n) \neq 1$。假设相对乘法误差在 ε 以内，即，

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

我们有

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

这并不是一个非常紧的界限，但它表明，如果我们能将乘法精确到比如 4 位数字，那么压缩性能的损失可以相当小。

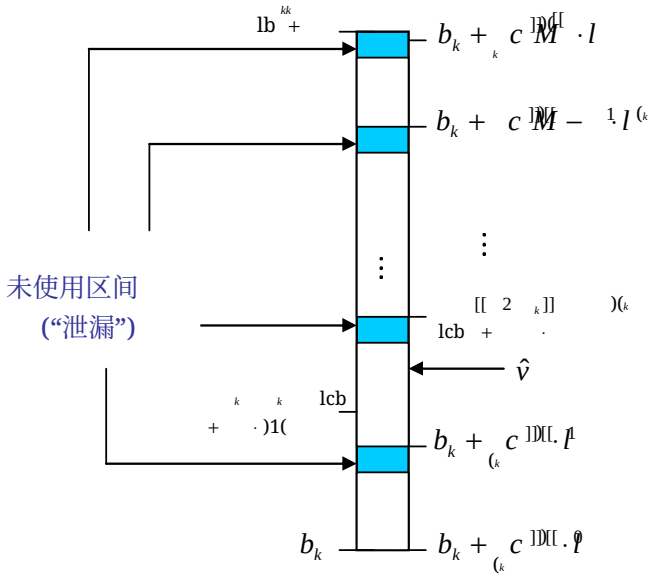

图 1.7：使用近似乘法对编码区间进行细分。由于采用定精度算术，只有在区间之间留出小段不用于编码的区域时，我们才能保证所有编码区间互不重叠。

1.6.6 正确解码的条件

图 1.7 展示了使用不精确乘法时区间如何细分。图中显示，比如说，$b_k + c(1) \cdot l_k$ 与 $b_k + [[c(1) \cdot l_k]]$之间可能存在显著差异，但如果解码器使用相同的近似方法，这种差异不会导致解码错误。

当不满足条件 (1.19) 时，就会发生解码错误。下面我们列出近似值必须满足的约束，并分析需要避免的三种主要编码错误成因。

(a) 区 间长度必须为正，且各区间必须互不相交 .
为确保区间不会收缩为一个点，且区间长度不会超过允许的区间范围，必须满足以下约束：

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

例如，如果近似值可能造成这样的情况： $[[c(s+1) \cdot l_k]] < [[c(s) \cdot l_k]]$，那么分配给 $s+1$ 和s 的子区间就会存在非空交集；每当编码值落入该交集时，就会发生解码错误。

如果 $[[c(s+1) \cdot l_k]] = [[c(s) \cdot l_k]]$，那么区间长度会缩减为零，并且无论接下来编码什么符号，都将保持为零。当 l_k 和 $p(s) = c(s+1) - c(s)$ 都非常小时，可能会因算术下溢而导致区间长度变为零。在第 2.1 节中，我们将说明通常会使用区间重缩放，将 l_k 保持在一定范围内，以避免

这个问题；但我们还必须确保所有符号概率都大于由算术精度决定的最小值（参见第 2.5 节和第 A.1 节）。

除了式（1.44）定义的条件外，我们还需要满足

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

这两个条件较容易满足，因为 $c(0) \equiv 0$ 且 $c(M) \equiv 1$，而且很容易使此类乘法达到精确结果。

（b）子区间必须嵌套。

我们必须确保，随着符号不断编码，近似误差的累积不会使区间基数移至所有先前区间之外的点。采用精确算术时，编码新符号后，区间基数会在分配给 s_{k+1} 的区间内增大，但不会越过边界进入分配给 $s_{k+1}+1$ 的区间，即

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

对于近似算术，相应的条件是：对每个数据序列，我们必须满足

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

要确定何时（1.47）可能不成立，我们必须对乘法近似值作出一些限制。应存在一个非负数 ε，使得

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

结合（1.40）、（1.47）和（1.48），可得

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

这等于

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

要找出（1.50）右侧的最大值，我们只需考虑情形$s_{k+2} = s_{k+3} = \ldots = M-1$，求得

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

这等价于

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

但我们从（1.3）中知道，根据定义 $c(M-1) + p(M-1) \equiv 1$! 这个矛盾的答案在于，采用精确算术时，（1.46）中的等式会成立

但只有经过无限多个符号之后才会如此。采用不精确算术时，不可能让半开区间得到充分利用且完全吻合，因此我们需要采取一些额外的预防措施，以确保（1.47）始终成立。（1.52）告诉我们，只要人为缩小分配给 $p(M-1)$ 的区间范围，就能解决这个问题。这相当于留出一些小区域（图 1.7 中以灰色区域表示），不用于编码，而是充当“安全网”。

这一额外例如，可以通过将（1.40）替换为一个空格来有意添加空格

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

其中 $0 < \zeta \ll 1$ 的选择旨在确保正确编码并减少压缩损失。

这些未使用子区间所造成的压缩损失称为“泄漏”，因为每当对一个符号进行编码时，总有一定比例的比特被“浪费”。这一比例的平均值为

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

其中 $p(s)/p'(s) > 1$ 是符号概率与区间大小减去未使用区域后的值之比。只要精度合理，泄漏就可以极小。例如，如果 $p(s)/p'(s) = 1.001$ （低精度），那么泄漏小于 0.0015 比特/符号。

(c) 逆算术运算不得产生误差累积。
请注意，在 (1.38) 中，我们假设编码器仅使用加法和乘法来定义解码。我们本可以使用

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

然而，这会引入近似减法和除法，而它们必须与编码器使用的近似值保持一致。这里不可能涵盖与逆运算相关的所有问题，但需要指出，关键在于观察误差累积。

例如，我们可以利用这样一个事实：在 (1.16) 中，解码只使用差值 $\varpi_k \equiv \hat{}v - b_k$，并采用以下递推关系。

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

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

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

然而，由于我们在（1.58）中使用了一系列减法，这种技术适用于整数算术实现（参见附录 A），但由于误差累积，可能不适用于浮点实现。

第 2 章

算术编码实现

在第二部分中，我们介绍算术编码的实际实现。我们将展示如何利用前几节介绍的所有算术编码性质，并开发一种适用于定精度算术的版本。首先，我们解释如何实现能够利用算术编码性质的二进制扩展精度加法，包括进位传播过程。接下来，我们介绍基于高效且简单的区间重缩放方法的完整编码和解码算法。我们分别描述浮点运算和整数运算的实现，并介绍一些其他编码实现方法，包括不同的缩放策略和进位传播策略。在介绍完编码过程的细节后，我们将探讨符号概率估计问题，并解释如何通过整合编码与信源建模来实现自适应编码。最后，我们分析算术编码的计算复杂度。

2.1 使用定精度算术进行编码

我们的第一个实际问题是：对符号进行编码时，精确表示区间长度所需的位数（或比特数）会不断增加。例如，如果我们有 $p(0) = 0.99$，并反复对符号 0 进行编码，就会得到

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

我们利用这样一个事实来解决这个问题：无需对区间长度进行精确乘法（第 1.6.5 节）。实际实现使用 P 位寄存器来存储区间长度尾数的近似值以及乘法结果。假设所有有效位数低于寄存器所能表示的位数的比特均为零。

解决了乘法精度问题后，当区间基数与区间长度的数量级相差很大时，我们仍需解决式 (1.37) 中加法的实现问题。我们将说明，重新缩放可以解决这一问题，同时实现精确加法并减少乘法精度损失。对于二进制输出，当区间长度小于 1/2 时，我们可以按照式 (1.31) 进行重新缩放，使用 $\delta \in \{0, 1/2\}$ 和 $\gamma = 2$。由于解码器需要知道重缩放参数，因此会将其保存在数据缓冲区 d 中，并用比特“0”或“1”指示 $\delta = 0$ 或 $\delta = 1/2$。

特殊情况 $\delta = 1$ 和 $\gamma = 1$（对应于二进制表示中的进位）将在后文说明。

为简化符号表示，我们将重缩放区间简写为 $|b, l\rangle$（不带下标），将重缩放编码值简写为 v。

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

其中，frac(·) 表示一个数的小数部分，并且

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

请注意，在这些条件下，我们有 $b \in [0, 1)$ 和 $l \in (0.5, 1]$，因此重新缩放过程称为重归一化。当然，l、b 和 v 会随 k 变化，但这种新符号表示更便于在算法描述或计算机程序中表示变量。

二进制表　　示的区间基数和长度具有如下结构　　：

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

其中，符号 a 表示任意比特值。

从 (2.3) 可以看出，存在一个由 P 个活动位组成的“窗口”，形成整数 L 和 B，它们分别对应于 l_k 的非零位以及重归一化后的长度 l。由于 l 的值被截断为 P 位精度，因此会有一组尾随零不影响加法。活动位左侧的比特是在重归一化过程中保存在数据缓冲区 d 中的，它们分为两组。

左侧的第一组是待定比特：编码新符号时，这些比特可能会因活动位产生的进位而改变。第二组是已确定的比特，也就是说，在编码过程结束之前，它们保持恒定。这是因为区间是嵌套的，也就是说，编码值不能超过

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

该等式表明，只有待定比特可能因活动位产生进位而发生变化。此外，不等式（2.4）也表明，最多只有一次进位会改变这些比特。如果发生进位，或者确定不会发生进位，这些比特就会变为已确定。因此，待定比特集合总是以 0 开头，后面可能只有若干个 1。随着新符号被编码，所有集合都会向右移动。

Algorithm 1

函数算术编码器（N, S, M, c, d）

1. 设置 { $b \leftarrow 0$; $l \leftarrow 1$;
   $t \leftarrow 0$; }
2. 对于 $k = 1$ 到 N, 执行
   2.1. 区间更新 (s_k, b, l, M, c);
   2.2. 如果 $b \geq 1$ 则
      2.2.1. 设置 { $b \leftarrow b - 1$;
         进位传播 (t, d); }
   2.3. 如果 $l \leq 0.5$ 则
      2.3.1. 编码器重归一化 (b, l, t, d);
3. 代码值选择 (b, t, d);
4. return t。

⋆初始化区间
⋆ 和位计数器
⋆ 编码 N 个数据符号
  ⋆ 根据符号更新区间
  ⋆ 检查是否有进位
    ⋆ 移动区间基数
    ⋆ 对缓冲区进行进位传播
  ⋆ 如果区间足够小
    ⋆ 然后对区间重新归一化
⋆ 选择最终码值
⋆ 返回编码比特数

2.1.1 使用缓冲区进位实现

综合我们目前所看到的内容，可以给出一种适用于定点算术的编码算法。算法1展示了函数算术编码器，用于按照第1.2节的符号表示，对S 个 N 数据符号的序列进行编码。该算法与我们在第1.4节中使用的编码过程非常相似，但在每次对符号进行编码后增加了一个重归一化阶段，并将已确定的比特和待定比特保存到缓冲区 d 中。该函数返回用于压缩 S 的比特数。

在算法1中，第1步将初始区间设置为 $[0,\ 1)$，并将位计数器 t 初始化为零。请注意，我们使用花括号（{ }）括起赋值集合，并在注释前使用符号“⋆”。在第2步中，我们依次重复进行区间调整大小和重归一化。区间更新后，我们立即判断是否发生进位，即是否 $b \geq 1$，随后检查是否需要进一步重归一化。当选择使编码比特数量最少的最终码值 v 时，编码过程在第3步结束。在我们的所有算法中，我们假设函数接收引用，即变量可以在被调用的函数内部进行修改。下面将介绍算法1所使用的各个函数。

更新区间 $|b,\ l\rangle$ 在数学上有许多等价的方法。使用 (1.9) 时，无需同时存储向量 p 和 c。在算法 2 中，我们使用 (1.40) 以差值的形式更新长度，并避免对最后一个符号 ($s = M - 1$) 进行乘法，因为解码器采用相同做法会更高效。为简化符号表示，我们不使用双括号来标示不精确乘法，但应当清楚，此处所有数字均表示 CPU寄存器中的内容。

在算法1的步骤 2.2.1 中，会调用缓冲区 d 中的进位传播函数，修改先前已添加到 d 的比特，并移动区间，使其具有$b < 1$。图 2.1 展示了进位传播过程。活动位以粗体显示，待定比特带下划线。每当发生进位时，就从最近添加到缓冲区 d 的比特开始，将所有比特取反，直到第一个 0 比特被取反，如下所示

Algorithm 2

过程区间更新（s, b, l, M, c）

1. 如果 $s = M - 1$
   那么设置 $y \leftarrow b + l$;
   否则设置 $y \leftarrow b + l \cdot c(s+1)$;
2. 设置 $\{\, b \leftarrow b + l \cdot c(s)$;
   $l \leftarrow y - b;\ \}$
3. 返回。

⋆ 最后一个符号的特殊情况
  ⋆ 区间末端
  ⋆ 下一个子区间的基值
⋆ 更新区间基值
⋆ 将更新区间长度设为差值

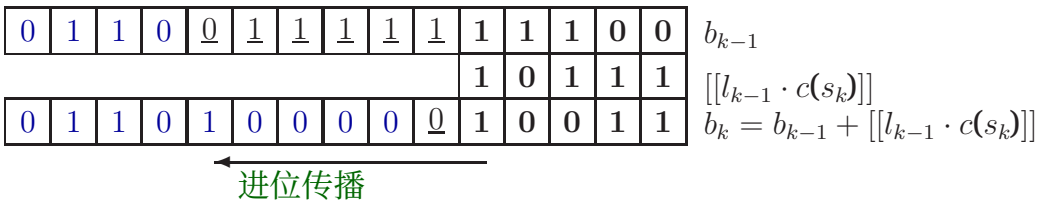

图 2.1：进位传播过程。粗体字母表示活动位，待定比特带下划线，最左侧的比特则已确定。

算法 3。

算法 4 实现了区间重归一化，其中我们会检测活动位是否变成了待定比特或已确定比特。当区间长度小于 0.5 时，将区间按 2 倍缩放，并向比特缓冲区 d 中添加一比特。

最终编码阶段是选择编码值。我们基本采用第 1.4.1 节末尾所述的相同过程，但这里选择属于重缩放区间的编码值。这个选择很容易，因为我们知道，重归一化后始终有 $0.5 < l \leq 1$ （见 (2.1)），这意味着我们只需要额外一比特来确定最终码值。换句话说：定义编码值的所有比特都已在缓冲区 d 中，我们只需再选择一比特。重缩放区间中需要考虑的仅有两种选择是 $v = 0.5$ 或 $v = 1$。

算法 6 所示的解码过程以压缩符号的数量 N、数据符号的数量 M 及其累积分布 c，以及存储压缩数据位的数组 d 作为输入。其输出是恢复的数据序列 S。解码器必须保持存储编码值的 P 位寄存器处于更新状态，因此会在 d 的末尾读取 P 个额外比特。我们假设这不会造成问题，并且这些比特已置为零。

区间选择基本上是对 (1.38) 的一种实现：我们要找到包含编码值 v 的子区间。算法 7 中展示的实现有一个小小的捷径：它将符号解码和区间更新 (1.39) 合并到一个函数中。我们从最后一个符号 ($s = M - 1$) 开始进行顺序搜索，因为我们假设符号按概率递增顺序排列。这样做的优点是

Algorithm 3

进位传播过程（t, d）

1. 设置 $n \leftarrow t$;
2. 当 $d(n) = 1$ 时执行循环
   2.1. 集合 { $d(n) \leftarrow 0$;
   $n \leftarrow n - 1$; }
3. 设置 $d(n) \leftarrow 1$;
4. 返回。

⋆ 将指针初始化为最后一个待处理位
⋆ 当进位传播时
  ⋆ 将尚未处理的 1 位取反，并
  ⋆ 移至前一位
⋆ 将尚未处理的 0 位取反

Algorithm 4

编码器重归一化过程 (b, l, t, d)

1. 当 $l \leq 0.5$ 时执行
   1.1. 设置 { $t \leftarrow t + 1$;
   $l \leftarrow 2l$; }
   1.2. 如果 $b \geq 0.5$
   则设置 { $d(t) \leftarrow 1$;
   $b \leftarrow 2(b - 0.5)$; }
   否则设置 { $d(t) \leftarrow 0$;
   $b \leftarrow 2b$; }
2. 返回。

⋆重归一化循环
  ⋆ 递增位计数器并
  ⋆ 缩放区间长度
  ⋆ 测试区间基数的最高有效位
    ⋆ 输出位 1
    ⋆ 移位并缩放区间基数
    ⋆ 输出位 0
    ⋆ 缩放区间基数

Algorithm 5

过程代码值选择（b, t, d）

1. 设置 $t \leftarrow t + 1$;
2. 如果 $b \leq 0.5$
   则设置 $d(t) \leftarrow 1$;
   否则设置 { $d(t) \leftarrow 0$;
   进位传播 ($t - 1$, d); }
3. return.

⋆ 递增位计数器
⋆ 非归一化码值选择
  ⋆ 选择 $v = 0.5$: 输出位 1
  ⋆ 选择 $v = 1.0$: 输出位 0 并
  ⋆进位传播

Algorithm 6

ˆ 算术解码器程序（N, M, c, d S）

1. 设置 { $b \leftarrow 0$; $l \leftarrow 1$;
   $v = \sum_{n=1}^{P} 2^{-n} d(n)$;
   $t \leftarrow P$; }
2. 对于 $k = 1$ 至 N，执行
   2.1. ˆs_k = 区间选择 (v, b, l, M,c);
   2.2. 如果 $b \geq 1$，则
      2.2.1. 设置 { $b \leftarrow b - 1$;
         $v \leftarrow v - 1$; }
   2.3. 如果 $l \leq 0.5$ 则
      2.3.1. 解码器重归一化 (v, b, l, t, d);
3. 返回。

⋆初始化区间
⋆ 读取 P 比特编码值
⋆ 初始化位计数器
⋆ 解码 N 数据符号
  ⋆ 解码符号并更新区间
  ⋆ 检查“溢出”
    ⋆ 移动区间基数
    ⋆ 移动编码值
  ⋆ 如果区间足够小
    ⋆ 则对区间进行重归一化

•

Algorithm 7

函数区间选择 (v, b, l, M, c)

1. 设置 { $s \leftarrow M - 1$;
   $x \leftarrow b + l \cdot c(M - 1)$;
   $y \leftarrow b + l$; }
2. 当 $x > v$ 时执行
   2.1. 设置 { $s \leftarrow s - 1$;
      $y \leftarrow x$;
      $x \leftarrow b + l \cdot c(s)$; }
3. 设置 { $b \leftarrow x$;
   $l \leftarrow y - b$; }
4. 返回 s。

⋆ 从最后一个符号开始搜索
⋆ 搜索区间的起点
⋆ 搜索区间的终点
⋆ 顺序搜索正确区间
  ⋆ 符号减一
  ⋆ 移动区间终点
  ⋆ 计算新的区间起点
⋆ 更新区间基值
⋆ 将区间长度更新为差值

•

排序符号和更高效的搜索将在第 2.2 节中说明 .

在算法 7 中，我们只使用与编码器所用运算完全相同的算术运算。这样，我们便能轻松确保编码器和解码器的近似值完全相同。可以采用若干简化方法来减少算术运算的次数（见附录 A）。

算法 8 中的重归一化与算法 4 类似，事实上，其中所有决策（比较操作）都应基于与编码器所用值完全相同的值。不过，它还需要以与区间基数相同的方式重新缩放编码值（比较式 (1.35) 和 (1.36)），并读取其最低有效位（值为 2^{-P}）。

Example 8

◁ 我们对示例 3 中的信源和数据序列应用了算法 1 至 8，结果如表 2.1 所示。第一列显示了导致区间发生变化的事件。如果对新符号 σ 进行编码，则将其显示为 $s = \sigma$，并显示

Algorithm 8

解码器重归一化过程（$v, b, l, t,$ d）

1。当 $l \leq 0.5$ 时
  1.1. 如果 $b \geq 0.5$
    则设置 $\{ b \leftarrow 2(b - 0.5);$
      $v \leftarrow 2(v - 0.5); \}$
    否则设置 $\{ b \leftarrow 2b;$
      $v \leftarrow 2v; \}$
  1.2. 设置 $\{ t \leftarrow t + 1;$
    $v \leftarrow v + 2^{-P} d(t);$
    $l \leftarrow 2l; \}$
2. 返回。

⋆重归一化循环
  ⋆ 移除最高有效位
    ⋆ 移位并缩放区间基数
    ⋆ 移位并缩放编码值
    ⋆ 缩放区间基数
    ⋆ 缩放编码值
  ⋆ 递增位计数器
  ⋆ 设置编码值的最低有效位
  ⋆ 缩放区间长度

算法 2 的结果。我们通过显示根据 (1.31) 用于重新缩放的值 δ，来指示重归一化（算法 4 ）期间区间的变化。

将这些结果与表 1.2 中的结果进行比较，我们可以看到，重归一化如何使所有数值保持在一个能够最大限度提高数值精度的范围内。事实上，表 2.1 中的结果是精确的，只需几个有效数字即可表示。表 2.1 还显示了解码器更新后的编码值。同样，这些结果是精确的，并且与表 1.2 中显示的结果一致。表 2.1 的第三列显示了比特缓冲区 d 的内容，以及每次区间重新缩放时添加到该缓冲区中的比特。请注意，进位传播发生了两次：一次是在 $s = 3$ 时，另一次是在算法 5 选择最终码值 $v = 1$ 时。 ▷

2.1.2 使用整数运算实现

尽管我们使用实数来描述算术编码的原理，但大多数实际实现只使用整数运算。自适应过程相当简单：只需假设 P 位整数包含实数的小数部分，并进行以下调整即可。（详情见附录 A。）

• 定义 $B = 2^P b$、$L = 2^P l$、$V = 2^P v$ 和 $C(s) = 2^P c(s)$。乘积可以使用 $2P$ 位计算，然后丢弃 P 个最低有效位。例如，更新区间长度时，我们计算 $L \leftarrow \lfloor L \cdot [C(s+1) - C(s)] \cdot 2^{-P} \rfloor$。长度值 $l = 1$ 无法用这种形式表示，但这并不是真正的问题。我们只需用 $L \leftarrow 2^P - 1$初始化缩放长度，并仅在 $l < 0.5$ （严格不等式）时应用重归一化。

• 进位条件 $b \geq 1$ 等价于 $B \geq 2^P$，这可能意味着整数溢出。可以通过访问中央处理器的进位标志来检测，也可以等效地检查 B 的值是否减小。

• 由于 $l > 0$，我们可以使用等于 $L' = 2^P l - 1$ 的缩放长度。这样一来，我们就能表示数值 $l = 1$，并且在 P 较小时获得一些额外的精度。另一方面

事件	缩放后的 **base** b	缩放后的 长度 l	比特缓冲区 **d**	缩放的 编码值 v	归一化 编码值 $(v-b)/l$
—	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	—	

表 2.1：将算术编码和解码（含重归一化）应用于示例 3 的信源和数据序列所得的结果。最终码值为 $\hat{v} = 0.1011111000100_2$ =0.74267578125。

，使用 $L' \leftarrow \left\lfloor (L'+1) \cdot [C(s+1) - C(s)] \cdot 2^{-P} - 1 \right\rfloor$ 更新长度还需要进行两次加法。

当使用 $2P$ 比特进行乘法运算时，我们可以确定避免长度下溢所允许的最小概率。由于重归一化保证 $L \geq 2^{P-1}$，因此若满足 (1.44)，则

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

这意味着，使用 P 位整数的这类实现所支持的最小概率为 $p(s) \geq 2^{1-P}$。

输入编码器 s_k	Data	二进制表示	十进制表示	8 位寄存器
—	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	—

表 2.2：使用 8 位精度进行算术运算时，示例 3 的算术编码与解码结果。

Example 9

◁ 表 2.2 展示了一个 8 位寄存器实现（$P = 8$）应用于示例 3 中所用数据序列和信源时的结果。表中以二进制表示法给出了区间 $\lfloor b_k, l_k \rangle$，以清楚表明，尽管我们使用 8 位算术运算，实际实现的仍是精确加法。按照式 (2.3) 和图 2.1 中的惯例，我们用粗体字母表示活动位，并用下划线表示待定比特。我们还展示了乘法的近似结果 $l_{k-1} \cdot c(s_k)$。最后一列显示了寄存器中包含活动位的二进制内容。我们用 $L' = 2^P l - 1$ 表示长度。结果也以十进制表示法给出，以便与表 1.2 中的精确结果进行比较。请注意，这些近似值仅在 7 个比特后就改变了编码值，但表示最终码值仍需要 13 个比特。 ▷

Algorithm 9

编码器重归一化过程（b, l, t, d）

1. while $l \le 1/D$ do
   1.1. 设置 $\{ t \leftarrow t+1; \}$
   $d(t) \leftarrow \lfloor D \cdot b \rfloor$;
   $b \leftarrow D \cdot b - d(t)$;
   $l \leftarrow D \cdot l$; $\}$
2. 返回。

⋆重归一化循环
- ⋆ 递增符号计数器
- ⋆ 从最高有效位输出符号
- ⋆ 更新区间基值
- ⋆ 缩放区间长度

•

2.1.3 高效输出

使用短寄存器的实现（如示例 9）需要尽快重新归一化区间，以避免精度和压缩效率下降。从表 2.1 可以看出，因此每当编码一个符号时，区间都可能需要多次重新缩放。虽然可以通过位移而非乘法来完成重新缩放，但这一过程仍然需要许多 CPU 周期（参见第 2.3 节）。

如果使用更长的寄存器（例如 16 比特或更多），则每次重新缩放时，将多个比特移至输出缓冲区 d，就能显著提高效率。这个过程相当于使用一个包含 D 个符号的编码器输出字母表，其中 D 是 2 的幂。例如，每次移动 1、2、4 或 8 比特的分组，分别对应包含 $D = 2$、4、16 和 256 个符号的输出字母表。较大的 D 也能提高进位传播和输出缓冲区 d 的使用效率。

将第 2.1.1 节中的算法修改为输出 D 个符号并不困难。重归一化是最重要的改动。式 (1.31) 定义的重缩放参数满足 $\delta \in \{0, 1/D, 2/D, \ldots, 1\}$ 和 $\gamma \equiv D$。同样，$\delta = 1$ 对应一个进位。算法 9 对算法 4 作出了所需的修改，而解码器重归一化的相应修改见算法 10。在附录 A 中，我们给出了输出 D 个符号的整数算术实现，其中进位传播见算法 25。

我们还需要修改定义归一化区间的方程 (2.1) 和 (2.2)，将其改为

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

and

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

Example 10

◁ 表 2.3 展示了当 $D = 16$ 时，算术编码的重归一化如何变化。与前面一样，信源和数据序列均来自示例 3。将这些结果与示例 8（表 2.1）的结果进行比较，可以看出区间重新缩放的次数大幅减少。 ▷

Algorithm 10

解码器重归一化过程（$v, b, l, t,$ d）

1. while $l \leq 1/D$ do
   1.1. set $\{\ t \leftarrow t+1;\ \}$
   $a \leftarrow \lfloor D \cdot b \rfloor;$
   $b \leftarrow D \cdot b - a;$
   $v \leftarrow D \cdot v - a + D^{-P} d(t);$
   $l \leftarrow D \cdot l;\ \}$
3. 返回。

⋆重归一化循环⋆ 递增符号计数器⋆ 最高有效位⋆ 更新区间基数⋆ 更新编码值⋆ 缩放区间长度

•

事件	缩放值 **base** b	缩放值 长度 l	比特缓冲区 **d**	缩放后的编码值 值 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	—

表 2.3：示例 3 的算术编码与解码结果，采用重归一化和一个包含 16 个符号的（十六进制）输出字母表。最终码值为 $\hat{}v = 0.\mathrm{BE20}_{16} = 0.1011\ 1110\ 0010\ 0000_2 = 0.74267578125$。

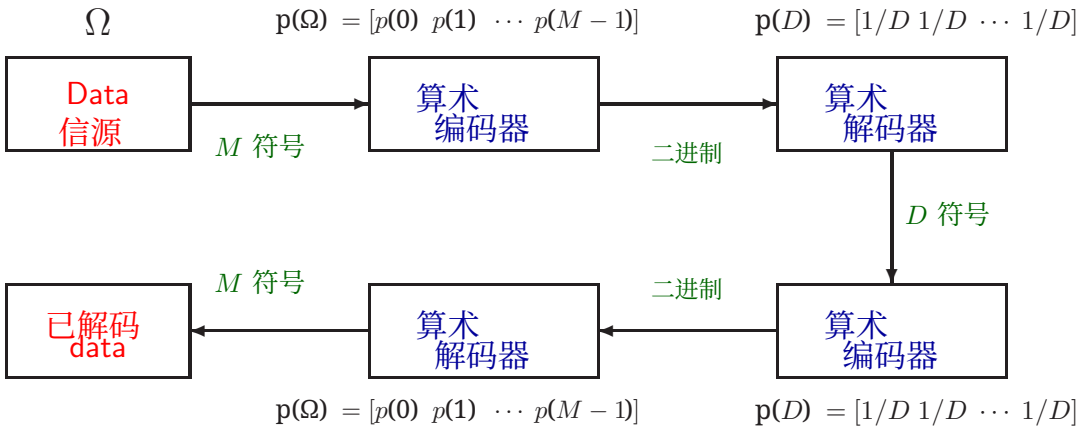

图 2.2：使用标准（二进制输出）算术编码器替代具有 D 符号（三进制、十进制等）输出的编码器时的配置。

算法 9 和 10 也可用于 D 不是 2 的幂的情况，但需要采用效率低下的基数为 D 的算术。问题在于，虽然可以近似计算与区间长度的乘法，但重新缩放必须精确进行。例如，如果 $D = 2$，那么可以通过位移精确计算乘以 2^{-P}；但要精确乘以 3^{-P}，则需要特殊函数 [39]。

更好的替代方案是采用一个小技巧，如图 2.2 所示。我们可以使用采用二进制算术的标准编码器/解码器来压缩数据序列。接下来，我们使用具有均匀概率分布 $\mathbf{p} = [1/D \quad 1/D \quad \cdots \quad 1/D]$ 的 D 符号字母表来“解码”二进制压缩数据。均匀分布不会改变码值的分布（因此不会影响压缩），但会将数据转换为所需的字母表。由于解码器执行的两个过程都完全可逆，因此解码器只需执行相应的逆过程。这样编码和解码所需的时间会增加一倍，但比使用基数为 D 的算术运算快得多。

2.1.4 注意进位

我们在第 2.1 节中已经看到，进位传播仅应用于待定比特集；该集合始终以一个 0 比特开头，后面可能跟着若干个 1 比特。待定比特集的示例如下：

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

显然，结构如此简单，我们无需保存待定比特就能知道它们是什么。只需用一个计数器记录待定比特的数量，并在重归一化过程中每当确定新的比特时递增计数器。每当发生进位，或重归一化产生新的待定 0 比特时，我们就可以输出（或“刷新”）这些比特。

请注意，重新缩放输出的比特并非都必须先处于未决状态，之后才能成为已确定状态。例如，如果我们有 $b < b + l \leq 0.5$，我们不仅知道下一个比特是 0，还知道它绝不可能因进位而改变，因此它已经确定。在缓冲区中实现进位传播时，可以忽略这些细节，但使用计数器时则不能。

采用 D 符号输出时，未决符号序列以符号 $\sigma \neq D-1$ 开始，后面可能跟着多个符号 $D-1$，如下所示。

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

在这种情况下，我们只能保留第一个符号 σ 以及一个用于记录未决符号数量的计数器。

需要注意的是，未决符号的数量可能无限增长，也就是说，我们可以构造出这样的分布和无限长的数据序列，使得比特永远无法确定。最终编码值选择（算法 5）会确定所有未决符号，因此我们可以定期重新启动编码器，以限制未决符号的数量。

处理进位传播有许多方法。最常见的方法是：

1. 将未决符号暂时保存到缓冲区中，然后在缓冲区内执行进位传播。这种简单的方法在处理字节时效率很高（$D = 256$），但在 D 较小时效率可能较低。只有在确定缓冲区足够大、能够容纳所有压缩符号时才能使用这种方法，因为所有符号都可能处于未决状态。

2. 使用计数器跟踪未决符号，并在符号确定后立即将其保存到缓冲区或文件中。根据计数器所用比特数的不同，可能会发生溢出（尾数使用 32、53 和 64 比特时，分别允许 $4 \cdot 10^9$、$9 \cdot 10^{15}$ 和 $2 \cdot 10^{19}$ 个未决符号）。在实际应用中，计数器溢出的可能性极低，尤其是在使用自适应编码器时。

3. 在缓冲区中使用进位传播，并采用以下形式的“比特填充” [15, 27]。待定比特一旦确定，就立即从缓冲区移出。当 1 比特的数量超过某个阈值（例如 16）时，便向压缩比特序列中添加一个人为的零，迫使待定比特确定下来。解码器可以通过比较所读连续 1 比特的数量来识别这种情况。当数量超过阈值时，解码器不会将下一个比特解释为数据，而是将其解释为进位信息。如果该比特为 1，则在解码器缓冲区中传播进位，解码过程继续正常进行。Q 编码器 [27]使用了这种技术。

2.1.5 其他重归一化方法

我们在第 1.6.4 节中展示了重新缩放区间具有很大的灵活性，并在第 2.1.1 节中介绍了一种基于特定形式重归一化的实现。其他选择会产生具有不同性质的重归一化区间，而不同的算术编码实现已以多种方式加以利用。下面我们展示几个示例。

- 我们选择了重归一化（2.6），由此产生的区间满足 $b \in [0, 1)$、$l \in (1/D, 1]$，以及 $b + l \in (1/D, 2)$。其主要优点是简化了进位检测和重归一化，尤其是在 $D > 2$ 时。请注意，它仅依据区间长度来确定何时进行重归一化。

• 何时进行重归一化的决定可以基于已确定的符号。例如，Rubin 提出的方法 [13] 保持区间满足 $b \in [0, 1)$，以及 $b + l \in (0, 1]$，当最高有效符号变为已确定状态时，即 $\lfloor Db \rfloor = \lfloor D(b + l) \rfloor$，便对区间进行重归一化；这意味着未决符号会保存在寄存器中。为避免区间长度最终收缩为零，编码器和解码器会在区间长度变得过小时重新缩放并移动区间，迫使未决符号变为已确定状态。

• Witten、Neal 和 Cleary [25, 51] 提出了一种颇受欢迎的算术编码实现。它不使用基数长度，而是用两个端点来表示区间，并以这样的方式保持其重归一化：$b + l \in (0, 1]$。每当比特已确定时都会进行重归一化，此外，当 $0.25 \leq b < b + l \leq 0.75$ 时也会进行。因此，它避免了 [13]，中出现的精度损失，并使用计数器来跟踪待定比特的数量。这种技术可以扩展到 D 符号输出，但并不像附录 A 中的方法那么简单。

• IBM 开发的二进制 Q 编码器 [27, 28]，将区间保持在 $l \in (0.75, 1.5]$。这样，我们通常会有 $l \approx 1$，并且可以用 $p \cdot l \approx p$ 近似乘法，以及 $(1 - p) \cdot l \approx l - p$。其变体包括 QM [36] 和 MQ 编码器 [62]。

2.2 自适应编码

由于典型的信息源往往相当复杂，要实现最优压缩，我们必须建立良好的数据源模型。有许多用于复杂信源建模的技术 [14, 25, 29, 44, 45, 46, 49]，它们会将源数据划分为不同类别，并假设每个类别中的源符号近似独立同分布，因此非常适合使用算术编码进行压缩。一般来说，我们并不知道每个类别中符号的概率。自适应编码是在编码过程中估计源符号概率的方法。本节将探讨如何将算术编码与动态概率估计高效结合。

2.2.1 计算符号分布的策略

计算分布的最高效技术取决于数据类型。处理完全未知的数据时，我们可能希望自适应过程完全自动进行。在其他情况下，我们可以利用对数据属性的一些了解，减少或消除自适应所需的工作量。下面将介绍几种最常见的分布估计策略的特点。

• 使用一种在编码和解码之前即可获得的恒定分布，通常通过收集大量典型样本中的统计数据来进行估计。这种方法可用于英语文本或天气数据等信源，但由于很少有信息源简单到可以用单一分布建模，因此它很少能取得最佳效果。此外，这种方法的灵活性非常有限（例如，英语文本的统计数据

并不适用于西班牙语文本）。另一方面，如果源模型非常详细，这种方法可能效果良好；事实上，在某些非常复杂的模型中，只有从海量数据中才能收集到有意义的统计数据，因此它是唯一可行的选择。

- 使用预定义分布并自适应地估计参数。例如，我们可以假设数据服从高斯分布，只估计每个符号的均值和方差。如果分布参数只允许取少数几个值，那么编码器和解码器就可以创建多个包含所有分布取值的向量，并根据共同的参数估计来使用它们。示例参见参考文献 [49] 。

- 使用两遍编码。第一遍收集信源的统计数据，第二遍利用收集到的统计数据对数据进行编码。解码时，必须在压缩数据开头包含向量 p 或 c 的缩放版本。例如，可以将一本书与其特有的符号统计数据一起归档（压缩）。通过在两遍之间共享处理过程，可以减少计算开销。例如，第一遍可以同时收集统计数据并将数据转换为游程。

- 使用基于先前已编码符号出现情况的分布，并在每编码一个符号后更新 c。我们可以从一个非常粗略的分布（例如均匀分布）开始；如果概率频繁变化，则可以定期重置估计值。下一节将介绍这种技术，它非常有效，也最为方便灵活。然而，持续更新累积分布会显著增加计算复杂度。另一种方法是在每编码一个符号后仅更新概率向量 p，而较少频繁地更新累积分布 c（第 2.2.5 节）。

2.2.2 累积分布的直接更新

编码/解码 k 个符号后，编码器/解码器可以估计符号的概率：

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

其中，$\tilde{P}(m) > 0$ 是符号 m 被编码/解码的次数加一（加上˜以避免概率为零）。符号出现计数器初始化为 $\tilde{P}(m) = 1$，并在符号编码后递增。我们将出现次数的累积和定义为

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

而累积分布为

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

Algorithm 11

~过程区间更新（s, b, l, M, C）

~1. 设置 $\gamma = l/C(M)$ ⋆ 计算除法结果
2. 如果 $s = M - 1$ ⋆ 最后一个符号的特殊情况
则设置 $y \leftarrow b + l$; ⋆ 区间末端
~否则设置 $y \leftarrow b + \gamma \cdot C(s+1)$; ⋆ 下一个子区间的基值
~3. 设置 $\{ b \leftarrow b + \gamma \cdot C(s)$; ⋆ 更新区间基值
$l \leftarrow y - b$; $\}$ ⋆ 将区间长度更新为差值
4. 返回。

Algorithm 12

~函数区间选择（v, b, l, M, C）

1. 设置 $\{ s \leftarrow M - 1$; ⋆ 从最后一个符号开始搜索
~$\gamma = l/C(M)$ ⋆ 计算除法结果
~$x \leftarrow b + \gamma \cdot C(M-1)$; ⋆ 搜索区间的起点
$y \leftarrow b + l$; $\}$ ⋆ 搜索区间的终点
2. 当 $x > v$ 时 ⋆ 顺序搜索正确区间
2.1. 设置 $\{ s \leftarrow s - 1$; ⋆ 符号减一
$y \leftarrow x$; ⋆ 移动区间终点
~$x \leftarrow b + \gamma \cdot C(s)$; $\}$ ⋆ 计算新的区间起点
3. 设置 $\{ b \leftarrow x$; ⋆ 更新区间基值
$l \leftarrow y - b$; $\}$ ⋆ 将区间长度更新为差值
4. 返回 s。

只需对第 2.1.1 节中的算法稍作修改，就足以加入动态更新累积分布的能力。首先，我们可以使用 (2.10) 将所有乘以 $c(s)$ 的运算改为如下形式。

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

同样，对 附录 A 中整数运算版本的修改可以采用以下形式 .

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

然而，由于除法的代价远高于加法和乘法，~因此最好只计算一次 $\gamma = l/C(M)$，并按算法 11 [50, 51] 所示进行区间更新。算法 7 的相应修改见算法 12。

在算法 7 和 12 中，我们可能需要计算几次乘法，前提是它们比一次除法更快（有关更高效的搜索方法，请参见算法 21）。如果事实并非如此，算法 13 将说明如何用额外的一次除法替代这些乘法。

Algorithm 13

˜函数区间选择 (v, b, l, M, C)

1. 设置 { $s \leftarrow M-1$;
   ˜$\gamma = l/C(M)$
   $W \leftarrow (v-b)/\gamma$; }
   ˜2. 当 $C(s) > W$ 时执行
   2.1. 设置 $s \leftarrow s-1$;
   ˜3. 设置 { $b \leftarrow b+\gamma \cdot C(s)$;
   ˜ ˜$l \leftarrow \gamma \cdot [C(s+1) - C(s)]$; }
4. 返回 s。

⋆ 从最后一个符号开始搜索
⋆ 计算除法结果
⋆ 按 C 缩放的编码值（M）
⋆ 查找正确区间
⋆ 符号减一
⋆ 更新区间基值
⋆ 更新区间长度

•

Algorithm 14

˜过程 更新分布 (s, M, C)

1. 对于 $m = s+1$ 到 M 执行
   ˜ ˜1.1 设置 $C(s) \leftarrow C(s) + 1$;
2. 返回。

⋆ 对于所有大于 s 的符号
⋆ 递增累积分布

•

经过上述修改，通过添加以下一行，使算法 1 和算法 6 具有自适应性：

˜2.4. 更新分布 (s_k, M, C);

更新累积分布的过程如算法 14 所示。请注意，在该算法中，最多需要进行 M 次加法。类似地，在算法 13 的步骤 2 中，我们最多需要执行 M 次比较操作和减法。由于这两种情况下的操作次数都会随着符号编号的增加而减少，因此按概率递增的顺序对符号进行排序是有益的。参考文献 [25] 给出了一种实现方式，可以在保持分布有序的同时更新分布。

2.2.3 二进制算术编码

二进制算术编码器仅适用于二进制源字母表（$M = 2$）。这是一类重要的编码器，因为它有助于解决动态更新累积分布（算法 14）所带来的许多复杂度问题。当 $M = 2$ 累积分布向量仅为 c $= [0\ p(0)\ 1]$, 时，编码和更新累积分布就会简单得多。

不过，需要注意的是，这里存在性能权衡。虽然二进制算术编码大大简化了对每个二进制数据符号的编码，但其最终的信息吞吐量（实际信息比特数）不可能超过每个编码符号 1 比特，这通常意味着每几个中央处理器时钟周期 [63] 才能处理 1 比特。因此，与过去相比，它们不再那么适合进行快速编码，但在编码二进制源时，仍然没有理由不利用其特殊属性。

算法 15 、16 和 17，分别给出了二进制编码（区间 -

Algorithm 15

二进制区间更新过程（$s, b, l, C(1)\ C(2)$）

1. 设定 $x \leftarrow l \cdot \tilde{C}(1)/\tilde{C}(2)$;
2. 如果 $s = 0$
   则设置 $l \leftarrow x$;
   否则设置 { $b \leftarrow b + x$;
   $l \leftarrow l - x$; }
3. 返回。

⋆ 区间划分点
⋆ 如果符号为零
  ⋆ 更新区间长度,
  ⋆ 移动区间基数并
  ⋆ 更新区间长度

•

Algorithm 16

二进制区间选择函数（$v,\ b,\ l,\ C(1)\ C(2)$）

1. 设置 $x \leftarrow l \cdot C(1)/C(2)$;
2. 如果 $b + x > v$
   然后设置 { $s \leftarrow 0$;
   $l \leftarrow x$; }
   否则设置 { $s \leftarrow 1$;
   $b \leftarrow b + x$;
   $l \leftarrow l - x$; }
3. 返回 s。

⋆ 区间划分点
⋆ 查找正确区间
  ⋆ 符号为 0：区间基数不变
  ⋆ 更新区间长度
  ⋆ 符号为 1:
  ⋆ 移动区间基数并
  ⋆ 更新区间长度

•

值更新）、解码（区间选择和更新）以及分布更新。请注意，我们不使用完整向量 C，而只使用 $C(1) = P(0)$ 和 $C(2) = P(0) + P(1)$。二进制算术编码无需更改重归一化过程。

Example 11

◁ 二进制算术编码具有普遍适用性，因为正如任何数都可以用比特表示一样，任何字母表中的数据符号都可以编码为二进制符号序列。图 2.3 展示了如何将来自 6 符号源的数据编码过程分解为一系列二元决策，并用二叉搜索树 [30]表示。叶节点对应数据源符号，中间节点则对应其下方所示的决策。

带下划线的数字用于表示中间节点，其值对应于比较时使用的数字（它们是二叉搜索树 [30] 的“键”）。例如，节点 $\underline{m}$ 对应于测试“$s < m$”? 符号从树的根节点开始编码，直到遇到叶节点为止。例如，如果我们要编码符号 $s = 2$，就从编码节点 3 所指示的信息“$s < 3$? ”开始；接着转到节点 1，编码信息“$s < 1$? ”，最后移动到节点 2，编码“$s < 2$? ”。在每个节点，信息都使用一组不同的概率进行编码，这些概率在图 2.3 中显示于树节点下方。这些基于符号出现次数的概率通过算法 17进行更新。包含 M 个符号的字母表需要 $M - 1$ 个概率

Algorithm 17

~过程 更新二元分布（s，C）

~ ~1. 若 $s = 0$ 则设置 $C(1) \leftarrow C(1) + 1$;

~ ~2. 令 $C(2) \leftarrow C(2) + 1$;

3. 返回。

⋆ 若 $s = 0$ 则递增 0 符号计数器

⋆ 递增符号计数器

•

中间节点的估计值。解码器按照相同的顺序，使用同一组概率进行解码。（注意，这等价于采用第 1.6.3 节中解释并在图 1.5 中展示的方案。）

采用这种方案不会造成压缩损失。例如，在编码符号 $s = 2$ 时，我们可以将符号概率计算为条件概率 [22] 的乘积。

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

这意味着

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

式 (2.14) 左侧是在 6 符号模型中直接编码符号 $s = 2$ 所需的比特数，这等于通过依次编码二叉搜索树中的二进制符号来编码同一符号所使用的比特数（见图 2.3）。我们不必担心计算条件概率的明确值，因为当我们为每个节点使用不同的自适应二元模型时，就会自动得到适当条件概率的估计值。此性质适用于任意数据字母表的二叉树分解。 ▷

例如，可以使用二分算法 [17, 19, 39]自动生成如图 2.3 所示的二叉搜索树。算法 18 和 19 展示了用于编码、解码和整体概率估计更新的此类实现。请注意，它们使用算法 15 和 16 的¯二进制编码和解码函数，并且仅使用一个维度为 $M - 1$ 的向量 C¯来存储树分支出现计数器。每个分量 $C(m) 0 < m < M$记录在树节点 m上出现“$s < m$?”的次数，而 C (0)记录已编码符号的总数。该向量初始化为相应节点左分支可到达的叶节点数量。

二进制条件概率估计在编码和¯解码过程中直接从 C 计算得出。例如，在图 2.3 的示例中，我们有

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

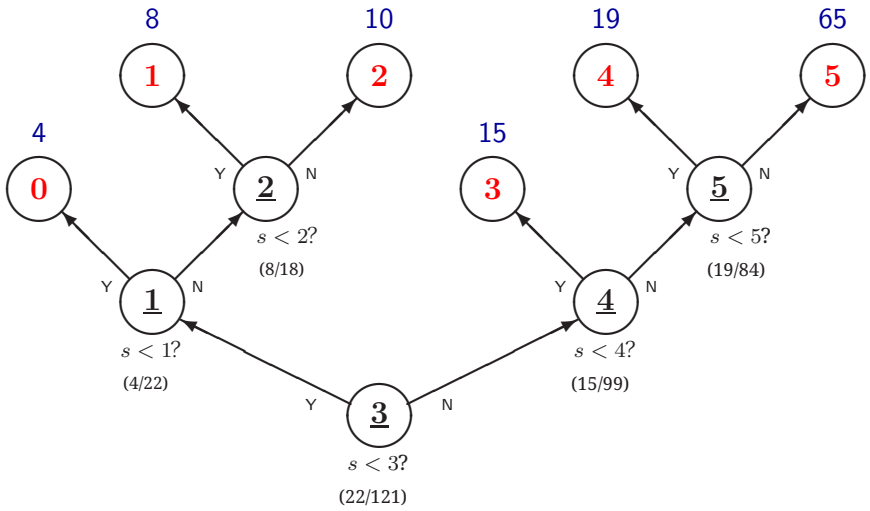

图 2.3：使用二进制编码器对来自 6 符号字母表的数据进行编码所需的顺序决策示例（二叉搜索树）。叶节点表示数据符号，其上方的数字表示其出现次数，$P(s)$。每个问号所示的二进制信息均使用括号中的概率估计进行编码。

为了对节点 3 处的决策进行编码，我们使用概率估计 22/121，该估计由计数器 $C(0) = 121$ 和 $C(3) = 22$定义。如果转到节点 1，那么下一个概率估计为 $C(1)/C(3) = 4/22$。另一方面，如果转到节点 4，则需要使用概率估计 $C(4)/[C(0) - C(3)] = 15/99$，这也可以直接根据 C 的组成部分计算得出。详情请参见算法 18 和 19。

由于算法 18 和 19 使用二分搜索，因此对于所有数据符号，调用二进制编码和解码函数的次数介于 $\lfloor \log_2 M \rfloor$ 和 $\lceil \log_2 M \rceil$ 之间。因此，通过使用二叉树搜索和二进制算术编码，我们大大降低了更新概率估计所需的最坏情况复杂度。

Example 12

◁ 图 2.4 展示了另一个可用于对示例 10 中的 6 符号数据进行编码的二叉搜索树示例。可以使用不同的搜索算法来创建不同的树。例如，我们可以使用由以下测试组成的顺序搜索：“$s < 5$?”、“$s < 4$?”、“$s < 3$?”、.....、“$s < 1$?”

通过二分搜索构建的树能够最大限度地减少待编码所需的二进制符号数量，但不能减少平均数量。图 2.4 中的树经过设计，使出现频率最高的符号只需经过最少次数的测试即可到达。表 2.4 显示了对信源中的符号进行编码所需的平均符号数，该信源的

Algorithm 18

过程区间更新 (s, b, l, M, C)

1. 设置 { u ← 0; n ← M; ⋆ 初始化二分搜索界限 k ← C(0); ⋆ 第一个除数 = 符号计数器 C(0) ← C(0) + 1; } ⋆ 递增符号计数器 2. 当 n − u > 1 时执行 ⋆ 二分搜索循环 2.1. 设置 m ← ⌊(u + n)/2⌋; ⋆ 计算中点 2.2. 如果 s < m ⋆ 符号小于中点，则设置 { n ← m; ⋆ 然后更新上限二进制区间更新 (0, b, l, C(m) k); ⋆ 编码符号 0 k ← C(m); ⋆ 设置下一个除数 C(m) ← C(m) + 1; } ⋆ 递增 0 符号计数器 否则设置 { u ← m; ⋆ 否则更新下限二进制区间更新 (1, b, l, C(m) k); ⋆ 编码符号 1 k ← k − C(m); } ⋆ 设置下一个除数 3

· 返回 ·

Algorithm 19

函数区间选择 (v, b, l, M, C)

1. 设置 { s ← 0; n ← M; ⋆ 初始化二分搜索边界 k ← C(0); ⋆ 第一个除数 = 符号计数器 C(0) ← C(0) + 1; } ⋆ 递增符号计数器 2. 当 n − s > 1 时执行 ⋆ 二分搜索循环 2.1. 设置 m ← ⌊(s + n)/2⌋; ⋆ 计算中点 2.2. 如果二进制区间选择 (v, b, l, C(m) k) = 0 ⋆ 如果符号小于中点，则设置 { n ← m; ⋆ 然后更新上限 k ← C(m); ⋆ 设置下一个除数 C(m) ← C(m) + 1; } ⋆ 递增 0 符号计数器；否则设置 { s ← m; ⋆ 否则更新下限 k ← k − C(m); } ⋆ 设置下一个除数 3. 返回 s。

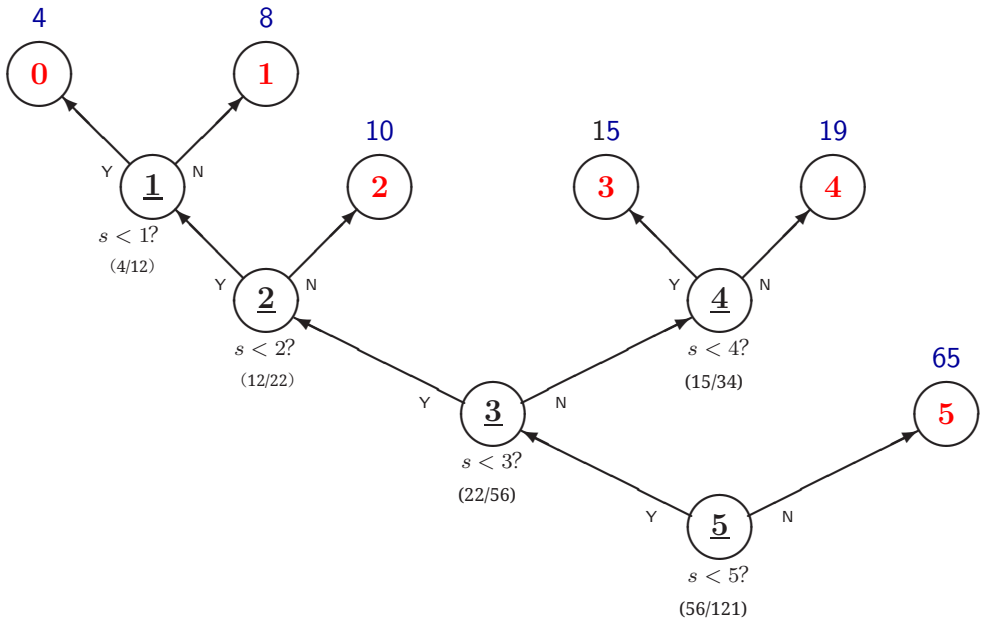

图 2.4：另一个用于对 6 符号源进行数据编码的二叉搜索树示例。符号出现次数与图 2.3 所示相同。该树经过专门设计，以尽量减少编码的二进制符号数。

例 11（将概率估计视为实际概率），由不同搜索方法生成的不同树。

由于符号已按概率从低到高排序，从概率最高的符号开始进行顺序搜索所定义的树，性能相当不错。图 2.4 所示的树使编码二进制符号的平均数量最少。下面我们将说明它是如何设计的。 ▷

前缀编码 [4, 5, 21, 32, 55, 56] 是使用上述决策树对信息进行编码的过程。我们上面展示的编码过程完全相同，只是在每个节点调用一个二进制算术编码/解码函数。因此，如果以编码的二进制符号数量来衡量复杂度，我们就可以利用所有关于前缀编码的已知结论，来分析二进制算术编码/解码的计算复杂度。

由于前缀编码的最优树是使用霍夫曼算法构建的 [2]，这些树对于二进制算术编码/解码也同样是最优的 [23]。严格来说，如果数据符号没有按照概率排序，最优霍夫曼树就不符合二叉搜索树的要求，也就是说，“键”没有正确排序，因此我们无法通过简单的比较（例如“$s < m?$ ”）来定义节点。解决这一问题的方法是存储从根节点到叶节点的路径，即霍夫曼码字。

Data 符号 s	概率 估计 $p(s)$	编码的二进制符号数					
		顺序搜索		二分搜索		最优搜索	
		$N(s)$	$p(s)N(s)$	$N(s)$	$p(s)N(s)$	$N(s)$	$p(s)N(s)$
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

表 2.4：对示例 10 的数据源应用由不同类型的二分搜索构建的树进行编码时所用的二进制符号数量。二分法和最优搜索对应的树分别见图 2.3 和图 2.4。

2.2.4 基于树的累积分布更新

在本节中，我们将展示如何利用二叉搜索树（第 2.2.3 节），高效地结合累积分布的计算、更新、编码和解码，而无需使用二进制算术编码器。我们介绍了与 Moffat [31] 和 Fenwick [42]提出的方法类似的技术。首先，我们通过一个示例说明如何根据使用二叉搜索树时收集的统计数据 C，计算累积分布向量 $\tilde{\mathbf{C}}$。

Example 13

◁ 我们来看看图 2.3 所示的二叉搜索树。假设我们一直使用算法 18 和 19 来计算树中符号的出现次数 $\bar{\mathbf{C}}$，现在想根据 $\mathbf{C} = [121\ 4\ 8\ 22\ 15\ 19]$ 计算累积分布 $\tilde{\mathbf{C}}$。

从树结构可以看出，除根节点外，每个节点上的计数器都记录了沿左分支向下找到的所有符号的出现次数，即

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

其中 $\tilde{P}(s)$ 是符号 s 出现的次数。由这些方程，我们

可以计算出累积分布向量：

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

我们可以对图 2.4 的树执行相同的操作，并得到不同的方程组。在这种情况下，计数器为

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

它们与累积分布的关系为

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

▷

从任何决策树得到的方程都是线性无关的，因此 $\tilde{C}$ 总是可以根据计数器 $\bar{C}$ 计算出累积分布 $\tilde{C}$。接下来我们将展示如何利用树结构，高效地计算编码符号 s、$C(s)$ 和 $C(s+1)$ 所需的 $\bar{C}$ 的分量。

为了计算 $C(s)$，从树的根节点向上移动到代表符号 s 的叶节点的过程中，对于每个 n 条件未满足（即沿其右侧分支向上移动）的节点，我们只需将 $\bar{C}(n)$ 的值相加。例如，要使用图 2.3 的树计算 $C(2)$，我们从节点 3 出发，向左移动到节点 1，再向右移动到节点 2，然后向右移动到叶节点 2。由于我们沿着节点 1 和节点 2 的右分支移动，因此可以得出 $\bar{C}(1)$、$\bar{C}(2)$，$C(2) = \bar{C}(1) + \bar{C}(2) = 12$。

其值：我们只需将到达叶节点 s 之前经过的最后一个左侧分支处的 $\bar{C}$ 累加和加到 $C(s)$ 和 $C(n)$ 上。例如，在计算 $C(2)$ 时，最后一个左侧分支位于根节点 3，因此 $C(3)$ 等于累加和（零）加上 $\bar{C}(3) = 22$。

Algorithm 20

过程区间更新 (s, b, l, M, C)

1. 设置 { $u \leftarrow 0$; $n \leftarrow M$;
   $E \leftarrow 0$; $F \leftarrow C(0)$;
   $\gamma \leftarrow l/C(0)$; }
2. 当 $n - u > 1$ 时执行
   2.1. 设 $m \leftarrow \lfloor (u+n)/2 \rfloor$;
   2.2. 如果 $s < m$
   那么设置 { $n \leftarrow m$;
   $F \leftarrow E + C(m)$;
   $C(m) \leftarrow C(m) + 1$; }
   否则设置 { $u \leftarrow m$;
   $E \leftarrow E + C(m)$; }
3. 如果 $s = M - 1$
   然后设置 $y \leftarrow b + l$;
   否则设置 $y \leftarrow b + \gamma \cdot F$;
4. 设置 { $b \leftarrow b + \gamma \cdot E$;
   $l \leftarrow y - b$;
   $C(0) \leftarrow C(0) + 1$; }
5. 返回。

- ⋆ 初始化二分搜索界限
- ⋆ 初始化累积和界限
- ⋆ 计算除法结果
- ⋆ 二分搜索循环
  - ⋆ 计算中点
  - ⋆ 如果符号小于中间值
    - ⋆ 然后更新二分上界
    - ⋆ 设置累积和的上界
    - ⋆ 递增节点出现计数器
    - ⋆ 否则更新二分下界
    - ⋆ 递增累积和的下界
- ⋆ 根据符号设置区间末端
  - ⋆ 精确乘法
  - ⋆ 下一个子区间的基值
- ⋆ 更新区间基值
- ⋆ 更新区间长度之差
- ⋆ 递增符号计数器

算法 20 和 21 展示了如何结合上述所有技术，同时计算并更新累积分布，并将计算所得的值用于编码和解码。它们使用由二分搜索定义的树，但很容易将其修改为其他树结构。算法 20 的步骤 2 执行后，我们已计算并更新了 $E = C(s)$ 和 $F = C(s+1)$，所需加法次数与 $\log_2(M)$ 成正比。

2.2.5 累积分布的周期性更新

我们在第 2.2.2 节中看到，自适应编码会显著增加算术编码的计算复杂度，因为更新累积分布需要付出较大代价。在第 2.2.3 节和第 2.2.4 节中，我们介绍了基于树的更新技术，可大幅降低这种复杂度。然而，自适应仍可能占编码总计算复杂度的相当大一部分；如果坚持假设概率模型必须在每个符号编码/解码后立即更新，那么我们实际上也无能为力，即估计值采用公式 (2.10) 所示的形式，并通过因子 $C(M)$ 进行一次除法。

然而，通过准确且快速的信源建模，我们可以避免概率变化过快，从而无需逐个符号地刷新估计值。例如，图像编码器可能会根据待编码部分的分类使用不同的概率，这些分类可以是“恒定”“平滑”“边缘”“高频模式”等。借助这些技术，我们可以假设信源状态可能快速变化，但每种状态对应的信源属性（符号概率）变化缓慢。

Algorithm 21

¯函数区间选择（v、 b、 l、 M、C）

1. 设置 { $s \leftarrow 0$; $n \leftarrow M$;
   $x \leftarrow b$; $y \leftarrow b + l$;
   $E \leftarrow 0$;
   $\gamma \leftarrow l/C(0)$; }
2. while $n - s > 1$ do
   2.1. 集合 { $m \leftarrow \lfloor (s+n)/2 \rfloor$;
   $z \leftarrow b + \gamma \cdot [E + C(m)]$; }
   2.2. 如果 $z > v$
   则设定 { $n \leftarrow m$;
   $y \leftarrow z$;
   $C(m) \leftarrow C(m) + 1$; }
   否则设置 { $s \leftarrow m$;
   $x \leftarrow z$;
   $E \leftarrow E + C(m)$; }
3. 设置 { $b \leftarrow x$;
   $l \leftarrow y - x$;
   $C(0) \leftarrow C(0) + 1$; }
4. 返回 s。

- ⋆ 初始化二分搜索符号范围
- ⋆ 设置搜索区间边界
- ⋆ 初始化累积和
- ⋆ 计算除法结果
- ⋆ 二分搜索循环
  - ⋆ 计算中点
  - ⋆ 中点处的值
  - ⋆ 如果符号小于中点
    - ⋆ 然后更新二分上界
    - ⋆ 新区间端点
    - ⋆ 增加节点出现计数器
    - ⋆ 否则更新二分下界
    - ⋆ 新区间基数
    - ⋆ 增加累积和的下限
- ⋆ 更新区间基值
- ⋆ 更新区间长度之差
- ⋆ 递增符号计数器

在这些假设下，除非数据符号的数量非常庞大（数千或数百万 [51]），否则效率显著更高的一种自适应算术编码形式是：每处理一个符号后，仅更新带有符号出现计数器的向量 (P)˜，并定期或在发生某些特殊事件后更新分布估计 (c)。例如，Q 编码器仅在重归一化期间更新概率估计 [27]。对于周期性更新，我们将 R 定义为两次更新 c 之间编码的符号数量。通常，更新周期R 是 M 的小倍数（例如 $R = 4M$），但为了在尽量减少计算量的同时提高准确度，我们可以先频繁更新，然后逐渐降低更新频率。

这种方法的一个直接结果是，编码时我们可以使用第 2.1.1 节和附录 A 中更简单的过程，并完全避免公式 (2.10) 至 (2.12) 中的除法。此外，如果周期 R 足够长，那么在更新 c 时进行许多复杂操作是可行的，从而提高编码 / 解码速度。例如，在第 2.3.2 节中，我们将说明如何通过按概率对符号排序、找出最优解码序列（霍夫曼树），或计算用于快速查表解码的数据，来加快解码速度。

2.3 复杂度分析

高计算复杂度一直是算术编码得以普及的一大障碍。事实上，在算术编码发明后的许多年里，它都被视为一种数学上的奇闻，因为加法使其速度变慢，乘法使其成本高昂，而除法则使其难以实际应用。在本节中，我们将分析

算术编码的复杂度，并说明使算术运算变得快速的技术进步如何改变复杂度分析。

本节中的许多结论都基于为分析算术编码复杂度而设计的实验测试，相关内容详见参考文献 [63]。另一项较早的复杂度实验评估见参考文献 [43]。

我们必须注意到，不同编码操作的相对计算复杂度已经发生了巨大变化。如今，算术运算的效率高得多，这不仅是因为中央处理器时钟频率大幅提高。过去，某些简单操作（比较操作、位移、查表）与算术运算（尤其是除法）所需时钟周期的比值相当大。如今，这一比值小得多 [58, 59, 60, 61], ，因此先前关于降低复杂度的假设已不再成立。例如，现在一组比较操作、位移和查表所花的时间，远远超过一次乘法。

算术编码的速度需要根据编码器输出或解码器接收的数据（真实信息）比特速率来衡量，也就是信息吞吐量。例如，二进制编码器可以非常简单、快速，但其吞吐量仅限于每个中央处理器时钟周期的一部分比特。另一方面，采用较大字母表的算术编码器处理信息的效率显著更高，每个中央处理器时钟周期轻松即可达到数个比特的信息吞吐量 [63]。

本节使用 [30], 的“大-O”符号表示，其中 $O(\cdot)$ 表示计算复杂度的渐近上界。

影响复杂度的主要因素包括

- 区间重归一化与压缩数据输入和输出。
- 符号搜索。
- 统计估计（仅适用于自适应模型）。
- 算术运算。

在接下来的章节中，我们将对这些方面逐一进行较为详细的分析。不过，出于以下三个原因，我们不讨论专用硬件实现（专用集成电路）：（a）这显然超出了本文的范围；（b）我们的模型也适用于某些专用硬件，例如数字信号处理器芯片；（c）许多针对通用中央处理器的优化技术也适用于高效硬件。

2.3.1 区间重归一化与压缩数据的输入和输出

在整数算术实现中，区间重归一化只需通过位移即可计算。然而，当 $D = 2$ 时，重归一化会相当频繁地发生，消耗许多时钟周期。采用 $D = 2^r$ 形式的较大输出字母表，可将重归一化的频率降低 r 倍（见示例 10），因此可能大幅减少重归一化所需的时钟周期数。浮点实现可能需要在重归一化时进行乘法运算，但如果 D 足够大，这些运算对总体复杂度的影响就不大。

在重归一化期间进行的数据输入和输出，也会显著影响编码和解码速度。由于计算机和通信系统以 8 比特（字节）为组进行处理，逐比特处理需要额外的时钟周期来正确对齐数据。因此，如果重归一化产生的比特能够轻松按字节对齐（例如， $D = 2^4$、 $D = 2^8$ 或 $D = 2^{16}$），速度就会显著提升。研究表明， $D = 256$ 这种情况尤其高效 [52, 54]。

2.3.2 符号搜索

算术解码器的计算复杂度可能比编码器高出许多倍，因为解码器需要找出当前编码值位于哪个子区间内，也就是说，需要求解

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

or

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

从数学上讲，这些方法是等价的，但我们用 (2.16) 表示先计算除法 $(v-b)/l$ 的情况（例如，算法 13）。

这是一个线搜索问题 [17, 30], ，我们需要尽可能减少测试点的数量，以及确定这些点所需的计算量。编码器和解码器之间可能存在的复杂度差异会随着数据符号数量 M 的增加而扩大。从算法 15 和 16 可以看出，当 $M = 2$ 时，编码器和解码器的复杂度实际上相同。

下面介绍符号搜索的五种主要方案。

(a) 已排序符号的顺序搜索

这种搜索方法用于算法 7 和 12，在最坏情况下需要搜索 $M-1$ 个区间才能找到解码符号。我们可以尝试根据符号的概率对其排序，以提高平均性能。假设符号按概率递增顺序排列，则平均测试次数为

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

其中， $\bar{s}(\Omega)$ 是数据源 Ω 输出的平均符号编号（排序后）。因此，只有当符号分布极度偏斜时，顺序搜索才可能高效（$\bar{s}(\Omega) \approx M-1$）。

(b) 二分搜索

采用这种搜索方法（见算法 19、21 和 28）时，测试次数有如下上界：

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

与数据符号的概率无关。请注意，编码器在与二进制编码（第 2.2.3 节）结合使用时，或使用树来更新概率估计（第 2.2.4 节）时，也可能需要实现二分搜索。

(c) 最优树形搜索
我们将在第 2.2.4 节展示，如何将对 M 符号字母表中的数据进行编码和解码的过程分解为一系列二元决策；在示例12中，我们将展示它与前缀编码以及使用霍夫曼算法设计的最优决策树之间的相似之处。解码过程中的区间搜索也由二元决策定义，而最优决策集合也使用霍夫曼算法计算得出 [2, 23]。因此，我们可以得出结论，此类方案中的平均测试次数受限于使用霍夫曼编码对信源进行编码时所用比特数的平均值，其表达式为 [9]

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

实现最优二叉搜索树需要额外的存储空间，用于存放通常表示霍夫曼编码所需的数据；还需要按照概率对符号重新排序（如图 2.4 所示的示例），或者采用累积分布的修订定义。

对于自适应编码，问题在于最优树由符号概率决定，而编码和解码开始时这些概率尚未知。可以通过定期重新构建树并更新分布来解决这个问题（第 2.2.5 节）。

(d) 排序符号上的二分搜索
我们可以将二分法的简洁性与一种考虑符号概率的技术结合起来。将数据符号按概率递增顺序排列后，我们可以查找满足 $c(m) \approx 0.5$ 的符号 m，并将其作为二分时划分区间的第一个符号（而不是中点）。如果分布近乎均匀，那么该符号应接近中点，性能与标准二分搜索相似。另一方面，如果分布高度偏斜，那么 $m \approx M-1$，这意味着先测试概率最高的符号，从而减少平均测试次数。

我们可以扩展这一技巧，不仅找到 $c(m)$ 接近 0.25 和 0.75 时对应的符号，还可以找到接近 0.125、0.375、0.625 和 0.825 等值时对应的符号。图2.5展示了一个累积分布示例，以及为初始化二分搜索而查找符号的过程。二叉搜索树的相应前几层如图 2.6 所示。解码器不必使用该树的所有层；它可以只使用第一层（并存储一个符号），也可以只使用前两层（并存储三个符号），还可以使用所需数量的层。图 2.6 中的完整二叉搜索树，是根据图中所示的测试应用标准二分搜索而定义的。如果编码器使用由累积分布定义的二叉搜索树的前 $V > 1$ 层，则平均测试次数接近最优值 (2.19)，最坏情况下的测试次数也不会比 (2.18) 大太多。

(e) 从查表开始的二分搜索
使用 (2.16) 的区间搜索方法需要进行一次除法，但其优点在于，这相当于在搜索之前将当前区间重新缩放为单位长度。这意味着，对于分数 $(v-b)/l$ 的量化版本，例如

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

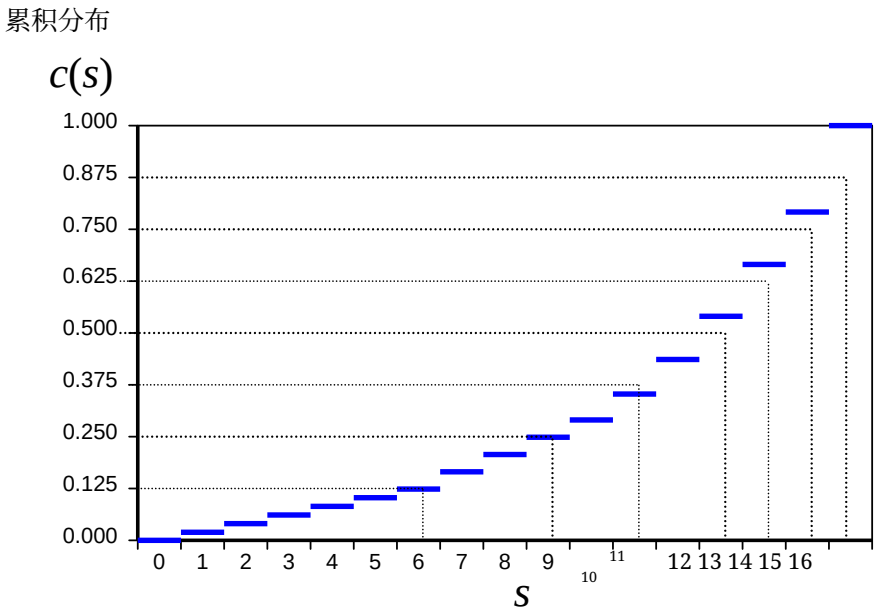

图2.5：用于确定准最优点以初始化已排序符号上的二分搜索的技术。

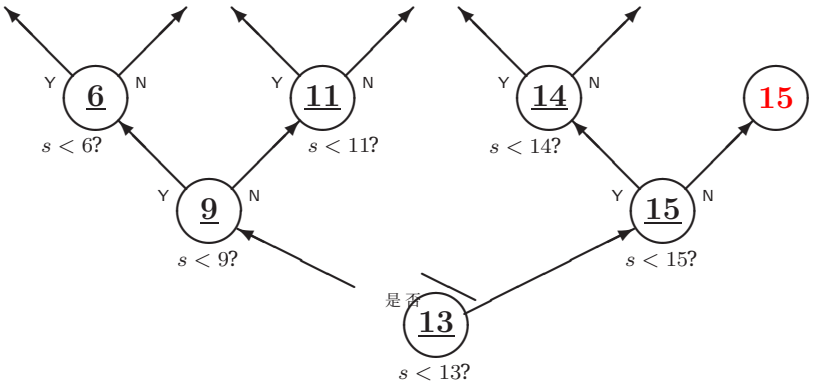

图 2.6：与图2.5的准最优二分搜索初始化相对应的决策过程。

$E(v,b,l)$	$(v-b)/l$	$\hat{}s_{\min}(E)$	$\hat{}s_{\max}(E)$
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

表 2.5：将 $(v-b)/l$ 量化为 8 个整数值，以便通过查表快速解码。解码符号（表项）的可能最小值和最大值使用 $\mathbf{c} = [0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1]$ 计算得出。

我们可以为二分搜索确定更好的初始值，并将其存储在含有K_t 个元素的表格中，从而显著加快解码速度。

表项为

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

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

注意，$\hat{}s_{\max}(E) <= \hat{}s_{\min}(E+1)$，因此一张表就足以实现正确解码。

Example 14

◁我们再次考虑示例 3 中的信源，其中 $\mathbf{c} = [0 \;\; 0.2 \;\; 0.7 \;\; 0.9 \;\; 1]$。表 2.5 展示了 $E(v,b,l)$ 的每个值如何对应于 $(v-b)/l$ 的一个可能值区间。通过分析 $\mathbf{c}$ 和这些区间，我们可以确定可能的解码符号范围，它们对应于 $\hat{}s_{\min}(E)$ 和 $\hat{}s_{\max}(E)$。例如，如果解码时有 $E(v,b,l) = 0$，那么我们知道 $(v-b)/l < 0.125$，因此 $\hat{}s(v) = 0$。另一方面，如果 $E(v,b,l) = 5$，那么 $0.625 \le (v-b)/l < 0.75$，这意味着我们只能确定 $1 \le \hat{}s(v) \le 2$，还需要再进行一次测试才能完成解码。▷

请注意，原始二分搜索所需的测试次数并不取决于概率；但当我们使用 $\hat{}s_{\min}(E)$ 和 $\hat{}s_{\max}(E)$ 初始化搜索时，初始区间取决于概率，因此查找概率最高的符号所需的测试次数更少。例如，我们可以从表 2.5 中看到，大多数表项都对应概率最高的符号。

查表解码的主要限制是，它仅适用于静态模型或定期更新的模型（第 2.2.5 节）。每次更新模型时计算表格的开销可能很小，因为查表仅用于初始化二分搜索，即使是较小的表格（例如包含 16 个条目）也能显著提高效率。

2.3.3 累积分布估计

自适应算术编码需要额外的工作，在对每个符号编码后更新累积分布估计值，或者定期更新。第 2.2 节涵盖了自适应编码器如何高效更新累积分布 c 的所有重要方面。我们分析了两种主要策略，具体取决于累积分布更新是与符号搜索同时进行，还是独立进行（第 2.3.2 节）。

（a）组合更新、编码和解码

在这种情况下，最高效的实现采用二叉搜索树，并将更新应用于向量 C，而不是 C。二进制编码（第 2.2.3 节）和基于树的更新（第 2.2.4 节）的渐近复杂度相同，具体取决于所用的树。此外，通过比较算法 18 和 19，以及算法 20 和 21，可以看出编码器和解码器的计算复杂度非常相近。

采用二分搜索可将最坏情况下的工作量降至最低，使每个编码符号所需的操作次数与 $\log_2(M)$ 成正比（公式 (2.18) ）。优化平均性能的霍夫曼树，所需的平均操作次数与 $_N_o$ (Ω) 成正比，其定义见公式 (2.19)。

（b）独立更新

当信源 Ω 的熵较低时，某些符号的出现频率应远高于其他符号；如果按概率递增的顺序排列符号，直接更新 C 可能会更高效。Witten 等人 [25, 51] 的实现采用了这一策略。然而，当所有符号的概率˜都相等时，排序和更新 C 所需的工作量平均与 $M/2$成正比，最坏情况下则与 $M-1$成正比。

˜如第 2.2.5 节所述，通过周期性更新 C，我们可以以合理的最坏情况复杂度重新计算并排序符˜号。我们假设更新周期为 R 个编码˜符号。一种选择是不对数据符号排序，并使用 (2.10) 更新 C；每次更新需要 $O(M)$ 次操作，每个编码符号的平均复杂度为 $O(M/R)$ 次操作。解码时可以使用二分法进行符号搜索。例如，如果选择 $R = M$，那么每次更新需要 $O(1)$ 次操作，这是相当合理的。

在每次更新期间按照概率对符号进行排序，在最坏情况下，每个符号需要执行 $O(M \log(M)/R)$ 次操作。由于符号通常已经按照之前的更新过程排好序，因此插入排序 [30] 通常平均只需执行 $O(M/R)$ 次操作。符号排好序后，我们可以使用第 2.3.2 节(d)中更高效的符号搜索。选择 $R = M$ 后，每个符号需要执行 $O(log(M))$ 次操作。

通过周期性更新，我们既可以对符号进行排序，也可以找到最优搜索策略（霍夫曼树），其复杂度合理：每个符号需要执行 $O(M \log(M)/R)$ 次操作 [30]。然而，尽管寻找霍夫曼编码的渐近复杂度与单纯排序相同，但它需要付出多得多的工作量。

表 2.6 总结了以上结果。请注意，对于最优搜索的复杂度分析，我们可以使用 $O(H(\Omega) + 1)$，而无需使用式 (2.19) 给出的更复杂且更紧的界。表 2.6 最后几列表明，需要使用形式如式 (2.10) 的除法。每个符号都改变累积分布，就需要使用不同的

搜索与更新方法	符号编码	解码器区间搜索	分布更新	除法
定长编码 最优搜索	$O(1)$	$O(H(\Omega)+1)$	NA	NA
顺序搜索 对已排序符号	$O(1)$	$O(M)$	$O(M)$	$O(1)$
组合更新和 编码，二分树	$O(\log(M))$			$O(1)$
组合更新和 编码，最优树	$O(H(\Omega)+1)$			$O(1)$
周期性更新， 二分搜索	$O(1)$	$O(\log(M))$	$O(M/R)$	O(1/R)
周期性更新， 最优搜索	$O(1)$	$O(H(\Omega)+1)$	$O(M$ 对数$(M)/R)$	O(1/R)

表 2.6：固定和不同自适应算术编码技术中，符号搜索和自适应（累积分布更新）的计算复杂度。通常，$R = 4M$ 可在不降低性能的情况下将复杂度降至最低。

除数来处理每个编码符号。另一方面，周期性更新使我们能够只计算一次除数的倒数，并将其用于编码多个（R）符号。

2.3.4 算术运算

目前，加法和乘法的速度与位移、比较等其他运算相差不大。另一方面，除法的延迟（执行该运算所需的时钟周期数）要长得多，而且无法与其他运算流水化执行（例如通过专用电路实现乘加） [58, 59, 60, 61]。

首先，我们来考虑定长编码器以及采用周期性更新的自适应编码器所需的运算。形式（1.9）和（1.40）中的算术编码递归分别需要进行两次乘法和一次、三次加法。因此，除了乘法速度非常慢的处理器外，编码所需的运算量相当合理。解码则更复杂，因为需要找到正确的解码区间。若使用（1.16）进行区间搜索，则还需要一次除法以及若干次比较（第 2.3.2 节）。仅使用乘法的版本（1.38）需要进行若干次乘法和比较，但如果符号搜索的效率足够高，其速度就会快于（1.16）；而且在乘法采用近似计算时，也无需定义除法。

由于式 (2.11) 和 (2.12) 中的除法，自适应编码可能会慢很多。每次区间更新最多可能需要两次除法。在算法 11、12 和 13 中，我们将说明如何避免其中一次除法。此外，更新累积分布可能需要大量加法。请注意，仅对累积分布进行周期性更新，可以显著降低这种自适应开销，因为并非每个编码符号都需要进行所有这些

除法和加法。例如，每次更新可能需要进行一次˜除法，用于计算 $1/C(M)$，以及若干次与 M 成正比的乘法和加法。如果每隔 M 个编码符号进行一次更新，那么平均每个编码符号所需的除法次数与 $1/M$ 成正比，而平均乘法和加法次数则为恒定值。

2.4 延伸阅读

我们介绍了这种编码方法，它如今通常被称为算术编码，但读者应当注意，其他类型的算术编码也曾被提出过 [6, 7]。一些国际标准中定义了算术编码的标准实现 [33, 36, 38, 62]。本文未涵盖的几种降低算术编码复杂度的技术，可参见引用文献 [11, 18, 23, 25, 27, 28, 34, 37, 42, 51,52]。我们没有提到这样一个事实：算术编码码流中的错误通常会导致灾难性的错误传播，因此，抗差错算术编码 [57] 在某些应用中非常重要。

附录 A

整数算术实现

以下算法展示了为适用于整数运算以及具有 D 个符号的输出字母表，对第 2.1.1 节中的算法所需进行的自适应修改。通常， D 是较小的 2 的幂，例如 2、4、16 或 256。如第 2.1.2 节所述，我们假设所有数字都表示为整数，但此处定义 $B = D^P b$、 $L = D^P l$ 和 $C(s) = D^P c(s)$。此外，我们假设乘法使用 $2P$ 位数进行计算，结果截断为 P 位数。重归一化（2.6）将 $L > D^{P-1}$ 设为，因此允许的最小概率定义为

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

即， $p(s) \geq D^{1-P}$。

用于整数运算的算法 22 与算法1几乎完全相同，只有 L 的初始化和重归一化判定不同。更新区间基数的算术运算可能导致整数寄存器溢出。为了明确这可以接受，我们像算法 23 中那样，将结果定义为模 D^P。乘法的结果再乘以 D^{-P} 并截断，即丢弃最低有效位。此处通过基数值的减小来检测溢出。因此，我们在算法 23 中将进位传播与区间更新一并实现。

请注意，在算法 24 的重归一化过程中，我们假设 D 是 2 的幂，且所有乘法和除法实际上都通过位移实现。算法 25 所示的、具有 D 符号输出的进位传播，与算法 3 非常相似。

在算法 5 中，代码值选择旨在尽量减少比特数，并假设解码器会在缓冲区 d 中填充足够多的零。只需在压缩数据末尾添加适当的额外符号，即可避免这一不便。算法 26 展示了所需的修改。它将区间基数稍微移位，并重置区间长度，因此调用编码器重归一化过程时，会将最后两个适当的输出符号添加到缓冲区 d。这样，正确解码就不依赖于解码器在最后一个压缩数据符号之后读取的符号值（取决于寄存器大小 P）。

使用整数运算时，解码器无需同时更新区间基数和编码值（参见算法 8）。由于解码始终基于差值 $v-b$，我们定义 $V = D^P(v-b)$，并且在解码时只使用 V和 L。唯一需要的其他改动是，我们必须从 V 中减去原本会加到 B上的数值。

ALGORITHM 22

函数算术编码器（N, S, M, C, d)

1. 设置 { $B \leftarrow 0$; $L \leftarrow D^P - 1$;
   $t \leftarrow 0$; }
2. 对于 $k = 1$ 到 N，执行
   2.1. 区间更新（s_k, M, B, L,C）；
   2.2. if $L < D^{P-1}$ then
   编码器重归一化（B, L, t, d）；
3. 代码值选择（B, L, t, d);
4. 返回 t。

⋆初始化区间
⋆ 和符号计数器
⋆ 编码 N 个数据符号
  ⋆ 根据符号更新区间
  ⋆ 如果区间足够小
    ⋆ 则对区间进行重归一化
⋆ 选择最终码值
⋆ 返回编码符号数量

ALGORITHM 23

过程区间更新 (s, M, B, L, C)

1. 如果 $s = M - 1$;
   然后将 $Y \leftarrow L$设为;
   否则将 $Y \leftarrow \left\lfloor [L \cdot C$设为$(s+1)] \cdot D^{-P} \right\rfloor$;
2. 将 { $A \leftarrow B$设为;
   $X \leftarrow \left\lfloor [L \cdot C(s)] \cdot D^{-P} \right\rfloor$;
   $B \leftarrow (B + X)$ 模 D^P;
   $L \leftarrow Y - X$; }
3. 如果 $A > B$，则传播进位（t,d）；
4. 返回。

⋆ 如果 s 是最后一个符号，则将第一个乘积设为
  ⋆ 设为当前区间长度
  ⋆ 或者设为计算出的值
⋆ 保存当前基数
⋆ 计算第二个乘积
⋆ 更新区间基值
⋆ 更新区间长度
⋆ 溢出时传播进位

ALGORITHM 24

编码器重归一化过程（B, L, t, d）

1. 当 $L < D^{P-1}$ 时执行
   1.1. set { $t \leftarrow t + 1$;
   $d(t) \leftarrow \left\lfloor B \cdot D^{1-P} \right\rfloor$;
   $L \leftarrow (D \cdot L)$ 模 D^P;
   $B \leftarrow (D \cdot B)$ 模 D^P; }
2. 返回。

⋆重归一化循环
  ⋆ 递增符号计数器
  ⋆ 输出符号
  ⋆ 更新区间长度
  ⋆ 更新区间基值

ALGORITHM 25

过程 进位传播 (t, d)

1. 设置 $n \leftarrow t$;
2. 当 $d(n) = D - 1$ 时执行循环
   2.1. 集合 { $d(n) \leftarrow 0$;
   $n \leftarrow n - 1$; }
3. 设置 $d(n) \leftarrow d(n) + 1$;
4. 返回。

⋆ 将指针初始化为最后一个未处理符号
⋆ 当进位传播时
  ⋆ 将未处理符号设为零
  ⋆ 移至前一个符号
⋆ 将第一个未处理符号递增

ALGORITHM 26

过程：代码值选择（B, L, t, d）

1. 设置 { $A \leftarrow B$;
   $B \leftarrow \left(B + D^{P-1}/2\right)$ 取模 D^P;
   $L \leftarrow D^{P-2} - 1$; }
2. 如果 $A > B$，则执行进位传播（t,d）；
3. 编码器重归一化（B, L, t, d）
4. 返回。

⋆ 保存当前基数
⋆ 增加区间基数
⋆ 设置新的区间长度
⋆ 如果发生溢出，则传播进位
⋆ 输出两个符号

ALGORITHM 27

过程：算术解码器（N, M, C, d）

1. 设置 { $L \leftarrow D^P - 1$;
   $V = \sum_{n=1}^{P} D^{P-n} d(n)$;
   $t \leftarrow P$; }
2. 对于 $k = 1$ 到 N，执行
   2.1. 设置 $s_k =$ 区间选择 (V, L, M, C);
   2.2. 如果 $L < D^{P-1}$ 则
   解码器重归一化（V, L, t, d）；
4. 返回。

⋆ 初始化区间长度
⋆ 读取编码值的 P 位
⋆ 初始化符号计数器
⋆ 解码循环
  ⋆ 解码符号并更新区间
  ⋆ 如果区间足够小
    ⋆重新归一化区间

ALGORITHM 28

函数区间选择（V, L, M, C）

1. 设置 $\{ s \leftarrow 0;\ n \leftarrow M;$
$X \leftarrow 0;\ Y \leftarrow L;$
2. 当 $n - s > 1$ 时执行
2.1. 设置 $\{ m \leftarrow \lfloor (s+n)/2 \rfloor;$
$Z \leftarrow \left\lfloor [L \cdot C(m)] \cdot D^{-P} \right\rfloor;\ \}$
2.2. 如果 $Z > V$
然后设置 $\{ n \leftarrow m;$
$Y \leftarrow Z;\ \}$否则设置 $\{ s \leftarrow m;$
$X \leftarrow Z;\ \}$3. 设置 $\{ V \leftarrow V - X;$
$L \leftarrow Y - X;\ \}$
4. 返回 s。

⋆ 初始化二分搜索界限
⋆ 初始化二分搜索区间
⋆ 二分搜索循环
⋆ 计算中点
⋆ 计算中点处的值
⋆ 如果新值大于编码值
⋆ 然后更新上限
⋆ 否则更新下限
⋆ 更新编码值
⋆ 将区间长度更新为差值

•

ALGORITHM 29

过程 解码器重归一化 (V, L, t, 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. 返回。

⋆重归一化循环⋆ 递增符号计数器⋆ 更新编码值⋆ 更新区间长度

•

在算法 7 中，我们在解码期间使用顺序搜索进行区间选择；在最坏情况下，需要测试 $M-1$ 个区间。在算法 28 中，我们使用二分法 [17, 19, 39] 求解 (1.16)，最多需要测试 $\lceil \log_2(M) \rceil$ 个区间（见第 2.2.3 节和第 2.3.2 节）。最后，算法 29 展示了解码器重归一化过程。

参考文献

[1] C.E. Shannon, 《通信的数学理论》, 《贝尔系统技术杂志》, 第 27 卷, 第 379–423 页, 1948 年 7 月。[2] D.A. Huffman, 《一种构造最小冗余码的方法》, IRE会议录, 第 40 卷, 第 1098–1101 页, 1952 年。[3] S.W. Golomb, 《游程编码》, IEEE信息论汇刊, 第 12 卷, 第 4 期, 第 399–401 页, 1966 年。[4] R.G. Gallager, 《信息论与可靠通信》, John Wiley & Sons, 纽约, 1968 年。[5] F. Jelinek, 《概率信息论》, McGraw Hill, 纽约州纽约, 1968 年。[6] R. Pasco, 《用于快速数据压缩的源编码算法》, 博士学位论文, 电气工程系, 斯坦福大学, 斯坦福, 马萨诸塞州, 1976 年。[7] J.J. Rissanen, 《广义Kraft不等式与算术编码》, 《IBM研究与开发杂志》, 第 20 卷, 第 3 期, 第 198–203 页, 1976 年 5 月。[8] J. Ziv 和 A. Lempel, 《一种用于序列数据压缩的通用算法》, IEEE信息论汇刊, 第 23 卷, 第 3 期, 第 337–343 页, 1977 年。[9] R.G. Gallagher, 《Huffman主题变体》, IEEE信息论汇刊, 第 24 卷, 第 6 期, 第 668–674 页, 1978 年 11 月。[10] J. Ziv 和 A. Lempel, 《通过变速率编码压缩单个序列》, IEEE信息论汇刊, 第 24 卷, 第 5 期, 第 530–536 页, 1978 年。[11] G.G. Langdon 和 J.J. Rissanen, 《算术字符串编码的方法与手段》, 美国专利 4,122,440, 1978 年 10 月。[12] G.N.N. Martin, 《区间编码：一种从数字化消息中去除冗余的算法》, 视频与数据记录会议, 英国南安普敦, 1979 年 7 月。[13] F. Rubin, 《使用定精度寄存器的算术流编码》, IEEE信息论汇刊, 第 25 卷, 第 6 期, 第 672–675 页, 1979 年 11 月。[14] J.J. Rissanen 和 G.G. Langdon, 《通用建模与编码》, IEEE信息论汇刊, 第 27 卷, 第 12–23 页, 1981 年 1 月。[15] G.G. Langdon 和 J.J. Rissanen, 《使用算术编码压缩黑白图像》, IEEE通信汇刊, 第 29 卷, 第 858–867 页, 1981 年 6 月。

[16] G.G. Langdon 和 J.J. Rissanen，《采用较少操作的算术编码方法与手段》，美国专利4,286,256，1981年8月。[17] P. Gill、M.H. Wright 和 W. Murray，《实用优化》，学术出版社，纽约，1982年。[18] G.G. Langdon 和 J.J. Rissanen，“一种简单的通用二进制信源编码”，IEEE信息论汇刊，第28卷，第800–803页，1982年9月。[19] J.R. Rice，《数值方法、软件与分析》，McGraw Hill，1983年。[20] N.S. Jayant 和 P. Noll，《波形数字编码》，普伦蒂斯-霍尔公司，恩格尔伍德克利夫斯，新泽西州，1984年。[21] R.J. McEliece，《信息与编码理论》，剑桥大学出版社，纽约州纽约，1984年。[22] A. Papoulis，《概率、随机变量与随机过程》，McGraw-Hill 出版公司，1984年。[23] G.G. Langdon，“算术编码导论”，《IBM研究与开发杂志》，第28卷，第2期，第135–149页，1984年3月。[24] D.E. Knuth，“动态霍夫曼编码”，《算法杂志》，第6卷，第163–180页，1985年6月。[25] I.H. Witten、R.M. Neal 和 J.G. Cleary，“用于数据压缩的算术编码”，《ACM 通讯》，第30卷，第6期，第520–540页，1987年6月。[26] J.S. Vitter，“动态霍夫曼码的设计与分析”，《ACM杂志》，第34卷，第825–845页，1987年10月。[27] W.B. Pennebaker、J.L. Mitchell、G.G. Langdon Jr.、R.B. Arps，“Q 编码器自适应二进制算术编码器基本原理概述”，《IBM研究与开发杂志》，第32卷，第6期，第717–726页，1988年11月。[28] W.B. Pennebaker 和 J.L. Mitchell，“Q 编码器的概率估计”，《IBM研究与开发杂志》，第32卷，第6期，第737–752页，1988年11月。[29] T.C. Bell、I.H. Witten 和 J. Cleary，《文本压缩》，普伦蒂斯-霍尔，1990年2月。[30] T.H. Cormen、C.E. Leiserson 和 R.L. Rivest，《算法导论》，麻省理工学院出版社，剑桥，马萨诸塞州，1990年。[31] A. Moffat，“线性时间自适应算术编码”，IEEE信息论汇刊，第36卷，第401–406页，1990年3月。[32] T.M. Cover 和 J.A. Thomas，《信息论基础》，John Wiley & Sons，纽约，1991年。[33]国际电信联盟（ITU），《连续色调静止图像的数字压缩与编码：要求与指南》，ITU-T建议书T.81，日内瓦，瑞士，1992年。

[34] P.G. Howard 和 J.S. Vitter，“算术编码的实际实现”，载于《图像与文本压缩》，J.A. Storer 编，克吕维尔学术出版社，波士顿，马萨诸塞州，1992年。[35] A. Gersho 和 R.M. Gray，《矢量量化与信号压缩》，克吕维尔学术出版社，波士顿，马萨诸塞州，1992年。[36] W.B. Pennebaker 和 J.L. Mitchell，《JPEG：静态图像数据压缩标准》，Von Nostrand Reinhold，纽约，1992年。[37] P.G. Howard 和 J.S. Vitter，“数据压缩的算术编码分析”，《信息处理与管理》，第28卷，第6期，第749–764页，1992年。[38] 国际电信联盟（ITU），《渐进式双层图像压缩》，ITU-T建议书T.82，日内瓦，瑞士，1993年。[39] W.H. Press、S.A. Teukolsky、W.T. Vetterling 和 B.P. Flannery，《C语言数值算法：科学计算艺术》，第2版，剑桥大学出版社，1993年。[40] J.M. Shapiro，“利用小波系数零树进行嵌入式图像编码”，IEEE信号处理汇刊，第41卷，第3445–3462页，1993年12月。[41] M. Burrows 和 D.J. Wheeler，《块排序无损数据压缩算法》，数字系统研究中心技术报告124，数字设备公司，加利福尼亚州帕洛阿尔托，1994年。[42] P.M. Fenwick，“一种用于累积频率表的新数据结构”，《软件实践与经验》，第24卷，第327–336页，1994年3月。[43] A. Moffat、N. Sharman、I.H. Witten 和 T.C. Bell，“多符号字母表编码方法的实证评估”，《信息处理与管理》，1994年。[44] A. Said 和 W.A. Pearlman，“通过字母表划分实现低复杂度波形编码”，IEEE信息论国际研讨会论文集，惠斯勒，加拿大，第373页，1995年9月。[45] M.J. Weinberger、J.J. Rissanen 和 R.B. Arps，“通用上下文建模在灰度图像无损压缩中的应用”，IEEE图像处理汇刊，第5卷，第575–586页，1996年4月。[46] X. Wu 和 N. Memon，“CALIC——一种基于上下文的自适应无损图像编解码器”，IEEE声学、语音与信号处理国际会议论文集，第4卷，第1890–1893页，1996年5月。[47] A. Said 和 W.A. Pearlman，“一种基于分层树集划分的新型快速高效图像编解码器”，IEEE电路与系统视频技术汇刊，第6卷，第243–250页，1996年6月[48] V. Bhaskaran 和 K. Konstantinides，《图像与视频压缩标准：算法与架构》，克吕维尔学术出版社，1997年。[49] S. LoPresto、K. Ramchandran 和 M. T. Orchard，“基于小波系数混合模型和快速估计量化框架的图像编码”，数据压缩会议，雪鸟，犹他州，第221–230页，1997年3月。[50] A. Moffat，“对论文《数据压缩算术编码的新颖设计》的评述”，《IEE计算机与数字技术论文集》，第144卷，第394–396页，1997年11月。

[51] A. Moffat、R.M. Neal 和 I.H. Witten，“重新审视算术编码”，ACM 信息系统汇刊，第16卷，第3期，第256–294页，1998年。[52] M. Schindler，“一种用于算术编码的快速再归一化方法”，IEEE 数据压缩会议论文集，1998年。[53] I.H. Witten、A. Moffat 和 T.C. Bell，《海量数据管理：文档与图像的压缩和索引》，第2版，摩根考夫曼出版社，1999年。[54] P-C. Wu，“一种算术编码的按字节归一化方法”，《软件：实践与经验》，第29卷，第299–309页，1999年4月。[55] K. Sayood，《数据压缩导论》，第2版，摩根考夫曼出版社，2000年。[56] D. Salomon，《数据压缩》，第2版，施普林格出版社，纽约州纽约，2000年。[57] I. Sodagar、B-B. Chai 和 J. Wus，“一种利用算术编码进行图像压缩的新型错误弹性技术”，国际声学、语音与信号处理会议论文集，第4卷，第2127–2130页，2000年6月。[58] 德州仪器公司，《TMS320C6000 中央处理器与指令集参考指南》，文献编号：SPRU189F，达拉斯，德克萨斯州，2000年。[59] 国际商业机器公司：《PowerPC 750CX/CXe RISC 微处理器用户手册》（初版），霍普韦尔章克申，纽约州，2001年。[60] 英特尔公司，《英特尔奔腾4处理器优化》，参考手册 248966，圣克拉拉，加利福尼亚州，2001年。[61] 太阳微系统公司，《UltraSPARC III 技术亮点》，加利福尼亚州帕洛阿尔托，2001年。[62] D.S. Taubman 和 M.W. Marcellin，《JPEG 2000：图像压缩基础、标准与实践》，克吕维尔学术出版社，波士顿，马萨诸塞州，2002年。[63] A. Said，《算术编码计算复杂度的比较分析》，惠普实验室报告，HPL–2004–75，加利福尼亚州帕洛阿尔托，2004年4月。