6. Arithmetic Coding

The Huffman method is efficient and produces the best codes for the individual data symbols. However, it has been shown that the only case where it produces ideal variable-size codes (codes whose average size equals the entropy) is when the symbols have probabilities of occurrence that are negative powers of 2 (i.e., numbers such as 1/2, 1/4, or 1/8). This is because the Huffman method assigns a code with an integral number of bits to each symbol in the alphabet. Information theory shows that a symbol with probability 0.4 should ideally be assigned a 1.32- bit code, since $-\log_2 0.4 \approx 1.32$. The Huffman method, however, normally assigns such a symbol a code of 1 or 2 bits.

Arithmetic coding overcomes the problem of assigning integer codes to the individual symbols by assigning one (normally long) code to the entire input file. The method starts with a certain interval, it reads the input file symbol by symbol, and it uses the probability of each symbol to narrow the interval. Specifying a narrower interval requires more bits, so the number constructed by the algorithm grows continuously. To achieve compression, the algorithm is designed such that a high-probability symbol narrows the interval less than a low-probability symbol, with the result that high-probability symbols contribute fewer bits to the output.

An interval can be specified by its lower and upper limits or by one limit and width. We use the latter method to illustrate how an interval's specification becomes longer as the interval narrows. The interval [0, 1] can be specified by the two 1-bit numbers 0 and 1. The interval [0.1,0.512] can be specified by the longer numbers 0.1 and 0.412. The very narrow interval [0.12575,0.1257586] is specified by the long numbers 0.12575 and 0.0000086. The output of arithmetic coding is interpreted as a number in the range [0, 1). [The notation [a, b) means the range of real numbers from a to b, including a but not including b. The range is "closed" at a and "open" at b.]. Thus, the code 9746509 is be interpreted as 0.9746509, although the 0. part is not included in the output file.

Before we plunge into the details, here is a bit of history. The principle of arithmetic coding was first proposed by Peter Elias in the early 1960s and was first described in [Abramson 63]. Early practical implementations of this method were developed by [Rissanen 76], [Pasco 76], and [Rubin 79]. [Moffat et al. 98] and [Witten et al. 87] should especially be mentioned. They discuss both the principles and details of practical arithmetic coding and show examples.

The first step is to calculate, or at least to estimate, the frequencies of occurrence of each symbol. For best results, the exact frequencies are calculated by reading the entire input file in the first pass of a two-pass compression job. However, if

the program can get good estimates of the frequencies from a different source, the first pass may be omitted. The main steps of arithmetic coding can be summarized as follows:

1. Start by defining the "current interval" as [0, 1).
2. Repeat the following two steps for each symbol *S* in the input stream:

   2.1 Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities.
   2.2 Select the subinterval for *S* and define it as the new current interval.

3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval (i.e., any number inside the current interval).

For each symbol processed, the current interval gets smaller, so it takes more bits to express it, but the point is that the final output is a single number and does not consist of codes for the individual symbols. The average code size can be obtained by dividing the size of the output (in bits) by the size of the input (in symbols). Notice also that the probabilities used in step 2.1 may change all the time, since they may be supplied by an adaptive probability model.

This example shows the compression steps for the short string SWISS_MISS. Table 2.47 shows the information prepared in the first step (the statistical model of the data). The five symbols appearing in the input may be arranged in any order. For each symbol, its frequency is first counted, followed by its probability of occurrence (the frequency divided by the string size, 10). The range [0, 1) is then divided among the symbols, in any order, with each symbol getting a chunk, or a subrange, equal in size to its probability. Thus, S gets the subrange [0.5,1.0) (of size 0.5), whereas the subrange of I is of size 0.2 [0.2, 0.4).

Char	Freq	Prob.	Range	CumFreq
		Total CumFreq=		10
S	5	$5/10 = 0.5$	[0.5, 1.0)	5
W	1	$1/10 = 0.1$	[0.4, 0.5)	4
I	2	$2/10 = 0.2$	[0.2, 0.4)	2
M	1	$1/10 = 0.1$	[0.1, 0.2)	1
␣	1	$1/10 = 0.1$	[0.0, 0.1)	0

Table 2.47: Frequencies and Probabilities of Five Symbols.

The symbols and frequencies in Table 2.47 are written on the output stream before any of the bits of the compressed code. This table will be the first thing

input by the decoder. The encoding process starts by defining two variables, Low and High, and setting them to 0 and 1, respectively. They define an interval [Low, High). As symbols are being input and processed, the values of Low and High are moved closer together, to narrow the interval.

After processing the first symbol S, Low and High are updated to 0.5 and 1, respectively. The resulting code for the entire input stream will be a number in this range ($0.5 \leq \text{Code} < 1.0$). The rest of the input stream will determine precisely where, in the interval [0.5, 1), the final code will lie. A good way to understand the process is to imagine that the new interval [0.5,1) is divided among the five symbols of our alphabet using the same proportions as for the original interval [0,1). The result is the five subintervals [0.5,0.55), [0.55,0.60), [0.60,0.70), [0.70,0.75), and [0.75,1.0). When the next symbol W is input, the third of those subintervals is selected and is again divided into five subsubintervals.

As more symbols are being input and processed, Low and High are being updated according to

*NewLow:=OldLow+Range*LowRange(X);*
*NewHigh:=OldLow+Range*HighRange(X);*

where *Range=OldHigh−OldLow* and *LowRange(X), HighRange(X)* indicate the low and high limits of the range of symbol X, respectively. In the example above, the second input symbol is W, so we update Low := 0.5+(1.0−0.5)×0.4 = 0.70, High := 0.5+ (1.0−0.5)×0.5 = 0.75. The new interval [0.70,0.75) covers the stretch [40%,50%) of the subrange of S. Table 2.48 shows all the steps involved in coding the string SWISS_MISS. The final code is the final value of Low, 0.71753375, of which only the eight digits 71753375 need be written on the output stream.

The decoder works in reverse. It starts by inputting the symbols and their ranges, and reconstructing Table 2.47. It then inputs the rest of the code. The first digit is 7, so the decoder immediately knows that the entire code is a number of the form 0.7 This number is inside the subrange [0.5,1) of S, so the first symbol is S. The decoder then eliminates the effect of symbol S from the code by subtracting the lower limit 0.5 of S and dividing by the width of the subrange of S (0.5). The result is 0.4350675, which tells the decoder that the next symbol is W (since the subrange of W is [0.4, 0.5)).

To eliminate the effect of symbol X from the code, the decoder performs the operation *Code:=(Code-LowRange(X))/Range*, where Range is the width of the subrange of X. Table 2.50 summarizes the steps for decoding our example string (notice that it has two rows per symbol).

Char.		The calculation of low and high
S	L	$0.0 + (1.0 - 0.0) \times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.70) \times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.70) \times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \times 1.0 = 0.72$
S	L	$0.715 + (0.72 - 0.715) \times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \times 1.0 = 0.72$
$\sqcup$	L	$0.7175 + (0.72 - 0.7175) \times 0.0 = 0.7175$
	H	$0.7175 + (0.72 - 0.7175) \times 0.1 = 0.71775$
M	L	$0.7175 + (0.71775 - 0.7175) \times 0.1 = 0.717525$
	H	$0.7175 + (0.71775 - 0.7175) \times 0.2 = 0.717550$
I	L	$0.717525 + (0.71755 - 0.717525) \times 0.2 = 0.717530$
	H	$0.717525 + (0.71755 - 0.717525) \times 0.4 = 0.717535$
S	L	$0.717530 + (0.717535 - 0.717530) \times 0.5 = 0.7175325$
	H	$0.717530 + (0.717535 - 0.717530) \times 1.0 = 0.717535$
S	L	$0.7175325 + (0.717535 - 0.7175325) \times 0.5 = 0.71753375$
	H	$0.7175325 + (0.717535 - 0.7175325) \times 1.0 = 0.717535$

Table 2.48: The Process of Arithmetic Encoding.

Char.	Code−low		Range
S	$0.71753375 - 0.5$	$= 0.21753375$	$/0.5 = 0.4350675$
W	$0.4350675 - 0.4$	$= 0.0350675$	$/0.1 = 0.350675$
I	$0.350675 - 0.2$	$= 0.150675$	$/0.2 = 0.753375$
S	$0.753375 - 0.5$	$= 0.253375$	$/0.5 = 0.50675$
S	$0.50675 - 0.5$	$= 0.00675$	$/0.5 = 0.0135$
$\sqcup$	$0.0135 - 0$	$= 0.0135$	$/0.1 = 0.135$
M	$0.135 - 0.1$	$= 0.035$	$/0.1 = 0.35$
I	$0.35 - 0.2$	$= 0.15$	$/0.2 = 0.75$
S	$0.75 - 0.5$	$= 0.25$	$/0.5 = 0.5$
S	$0.5 - 0.5$	$= 0$	$/0.5 = 0$

Table 2.50: The Process of Arithmetic Decoding.

Example 4.3.1:

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

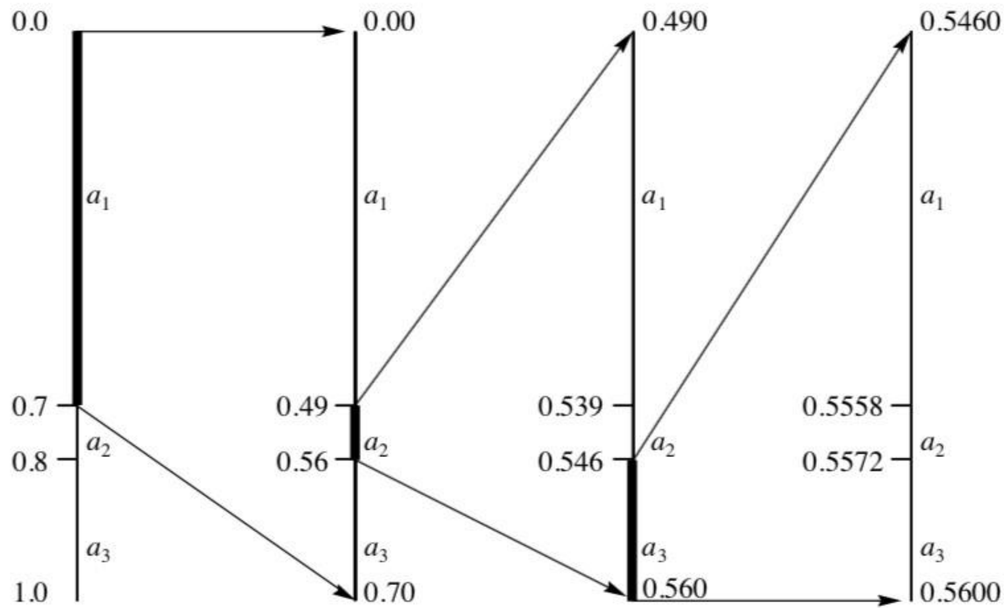

FIGURE 4.1 Restricting the interval containing the tag for the input sequence $\{a_1, a_2, a_3, \ldots\}$.

The partition in which the tag resides depends on the first symbol of the sequence being encoded. For example, if the first symbol is a_1, the tag lies in the interval $[0.0, 0.7)$; if the first symbol is a_2, the tag lies in the interval $[0.7, 0.8)$; and if the first symbol is a_3, the tag lies in the interval $[0.8, 1.0)$. Once the interval containing the tag has been determined, the rest of the unit interval is discarded, and this restricted interval is again divided in the same proportions as the original interval. Suppose the first symbol was a_1. The tag would be contained in the subinterval $[0.0, 0.7)$. This subinterval is then subdivided in exactly the same proportions as the original interval, yielding the subintervals $[0.0, 0.49)$, $[0.49, 0.56)$, and $[0.56, 0.7)$. The first partition as before corresponds to the symbol a_1, the second partition corresponds to the symbol a_2, and the third partition $[0.56, 0.7)$ corresponds to the symbol a_3. Suppose the second symbol in the sequence is a_2. The tag value is then restricted to lie in the interval $[0.49, 0.56)$. We now partition this interval in the same proportion as the original interval to obtain the subintervals $[0.49, 0.539)$ corresponding to the symbol a_1, $[0.539, 0.546)$ corresponding to the symbol a_2, and $[0.546, 0.56)$ corresponding to the symbol a_3. If the third symbol is a_3, the tag will be restricted to the interval $[0.546, 0.56)$, which can then be subdivided further. This process is described graphically in Figure 4.1.

7. Dictionary Coding

All the aforementioned statistical compression methods use a statistical model of the data, which is why the quality of compression they achieve depends on how good that model is. Dictionary- based compression methods do not use a statistical model, nor do they use variable-size codes. Instead they select strings of symbols and encode each string as a token using a dictionary. The dictionary holds strings of symbols, and it may be static or dynamic (adaptive). The former is permanent, sometimes allowing the addition of strings but no deletions, whereas the latter holds strings previously found in the input stream, allowing for additions and deletions of strings as new input is being read.

Given a string of n symbols, a dictionary-based compressor can, in principle, compress it down to nH bits where H is the entropy of the string. Thus, dictionary-based compressors are entropy encoders, but only if the input file is very large. For most files in practical applications, dictionary-based compressors produce results that are good enough to make this type of encoder very popular. Such encoders are also general purpose, performing on images and audio data as well as they perform on text.

The simplest example of a static dictionary is a dictionary of the English language used to compress English text. Imagine a dictionary containing perhaps half a million words (without their definitions). A word (a string of symbols terminated by a space or a punctuation mark) is read from the input stream and the dictionary is searched. If a match is found, an index to the dictionary is written into the output stream. Otherwise, the uncompressed word itself is written.

Adaptive dictionary-based method is preferable. Such a method can start with an empty dictionary or with a small, default dictionary, add words to it as they are found in the input stream, and delete old words because a big dictionary slows down the search. Such a method consists of a loop where each iteration starts by reading the input stream and breaking it up (parsing it) into words or phrases. It then should search the dictionary for each word and, if a match is found, write a token on the output stream. Otherwise, the uncompressed word should be written and also added to the dictionary. The last step in each iteration checks whether an old word should be deleted from the dictionary.

Having one’s name attached to a scientific discovery, technique, or phenomenon is considered a special honour in science. Having one’s name associated with an entire field of science is even more so. This is what happened to Jacob Ziv and Abraham Lempel. In the 1970s these two researchers developed the first methods, LZ77 and LZ78, for dictionary-based compression. Their ideas have been a source of inspiration to many researchers, who generalized, improved, and

combined them with RLE and statistical methods to form many commonly used lossless compression methods for text, images, and audio.

In general, compression methods based on strings of symbols can be more efficient than methods that compress individual symbols. In principle, better compression is possible if the symbols of the alphabet have very different probabilities of occurrence. We use a simple example to show that the probabilities of strings of symbols vary more than the probabilities of the individual symbols constituting the strings.

We start with a 2-symbol alphabet a_1 and a_2, with probabilities $P_1 = 0.8$ and $P_2 = 0.2$, respectively. The average probability is 0.5, and we can get an idea of the variance (how much the individual probabilities deviate from the average) by calculating the sum of absolute differences $|0.8 - 0.5| + |0.2 - 0.5| = 0.6$. Any variable-size code would assign 1-bit codes to the two symbols, so the average size of the code is one bit per symbol.

We now generate all the strings of two symbols. There are four of them, shown in Table 3.1a, together with their probabilities and a set of Huffman codes. The average probability is 0.25, so a sum of absolute differences similar to the one above yields

$$|0.64 - 0.25| + |0.16 - 0.25| + |0.16 - 0.25| + |0.04 - 0.25| = 0.78.$$

The average size of the Huffman code is $1\times0.64+2\times0.16+3\times0.16+3\times0.04 = 1.56$ bits per string, which is 0.78 bits per symbol.

In the next step we similarly create all eight strings of three symbols. They are shown in Table 3.1b, together with their probabilities and a set of Huffman codes. The average probability is 0.125, so a sum of absolute differences similar to the ones above yields

$$|0.512 - 0.125| + 3|0.128 - 0.125| + 3|0.032 - 0.125| + |0.008 - 0.125| = 0.792.$$

The average size of the Huffman code in this case is $1\times0.512+3\times3\times0.128+3\times5\times 0.032 + 5 \times 0.008 = 2.184$ bits per string, which equals 0.728 bits per symbol. As we keep generating longer and longer strings, the probabilities of the strings differ more and more from their average, and the average code size gets better (Table 3.1c). This is why a compression method that compresses strings, rather than individual symbols, can, in principle, yield better results.

String	Probability	Code
a_1a_1	$0.8 \times 0.8 = 0.64$	0
a_1a_2	$0.8 \times 0.2 = 0.16$	11
a_2a_1	$0.2 \times 0.8 = 0.16$	100
a_2a_2	$0.2 \times 0.2 = 0.04$	101

(a)

Str. size	Variance of prob.	Avg. size of code
1	0.6	1
2	0.78	0.78
3	0.792	0.728

(c)

String	Probability	Code
$a_1a_1a_1$	$0.8 \times 0.8 \times 0.8 = 0.512$	0
$a_1a_1a_2$	$0.8 \times 0.8 \times 0.2 = 0.128$	100
$a_1a_2a_1$	$0.8 \times 0.2 \times 0.8 = 0.128$	101
$a_1a_2a_2$	$0.8 \times 0.2 \times 0.2 = 0.032$	11100
$a_2a_1a_1$	$0.2 \times 0.8 \times 0.8 = 0.128$	110
$a_2a_1a_2$	$0.2 \times 0.8 \times 0.2 = 0.032$	11101
$a_2a_2a_1$	$0.2 \times 0.2 \times 0.8 = 0.032$	11110
$a_2a_2a_2$	$0.2 \times 0.2 \times 0.2 = 0.008$	11111

(b)

Table 3.1: Probabilities and Huffman Codes for a Two-Symbol Alphabet.

7.1 Static Dictionary

Choosing a static dictionary technique is most appropriate when considerable prior knowledge about the source is available. This technique is especially suitable for use in specific applications. For example, if the task were to compress the student records at a university, a static dictionary approach may be the best. This is because we know ahead of time that certain words such as “Name” and “Student ID” are going to appear in almost all of the records.

One of the more common forms of static dictionary coding is digram coding. In this form of coding, the dictionary consists of all letters of the source alphabet followed by as many pairs of letters, called digrams, as can be accommodated by the dictionary. For example, suppose we were to construct a dictionary of size 256 for digram coding of all printable ASCII characters. The first 95 entries of the dictionary would be the 95 printable ASCII characters. The remaining 161 entries would be the most frequently used pairs of characters.

The digram encoder reads a two-character input and searches the dictionary to see if this input exists in the dictionary. If it does, the corresponding index is encoded and transmitted. If it does not, the first character of the pair is encoded. The second character in the pair then becomes the first character of the next digram. The encoder reads another character to complete the digram, and the search procedure is repeated.

Example: Suppose we have a source with a five-letter alphabet A = {a, b, c, d, r}. Based on knowledge about the source, we build the dictionary shown in Table 5.1. Suppose we wish to encode the sequence: ***abracadabra***

TABLE 5.1 A sample dictionary.

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

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

A list of the 30 most frequently occurring pairs of characters in an earlier version of this chapter is shown in Table 5.2. For comparison, the 30 most frequently occurring pairs of characters in a set of C programs is shown in Table 5.3. In these tables, b/ corresponds to a space and *nl* corresponds to a new line. Notice how different the two tables are. It is easy to see that a dictionary designed for compressing LATEX documents would not work very well when compressing C programs. However, generally we want techniques that will be able to compress a variety of source outputs. If we wanted to compress computer files, we do not want to change techniques based on the content of the file. Rather, we would like the technique to adapt to the characteristics of the source output. We discuss adaptive-dictionary-based techniques in the next section.

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

Pair	Count	Pair	Count
$e\not b$	1128	ar	314
$\not b t$	838	at	313
$\not b\not b$	823	$\not b w$	309
th	817	te	296
he	712	$\not b s$	295
in	512	$d\not b$	272
$s\not b$	494	$\not b o$	266
er	433	io	257
$\not b a$	425	co	256
$t\not b$	401	re	247
en	392	$\not b\$$	246
on	385	$r\not b$	239
$n\not b$	353	di	230
ti	322	ic	229
$\not b i$	317	ct	226

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

Pair	Count	Pair	Count
$\not b\not b$	5728	st	442
$nl\not b$	1471	le	440
$;nl$	1133	ut	440
in	985	$f($	416
nt	739	ar	381
$=\not b$	687	or	374
$\not b i$	662	$r\not b$	373
$t\not b$	615	en	371
$\not b =$	612	er	358
$);$	558	ri	357
$,\not b$	554	at	352
$nlnl$	506	pr	351
$\not b f$	505	te	349
$e\not b$	500	an	348
$\not b *$	444	lo	347