

# 5

## Statistical Methods

Statistical data compression methods employ variable-length codes, with the shorter codes assigned to symbols or groups of symbols that appear more often in the data (have a higher probability of occurrence). Designers and implementors of variable-length codes have to deal with the two problems of (1) assigning codes that can be decoded unambiguously and (2) assigning codes with the minimum average size. The first problem is discussed in detail in Chapters 2 through 4, while the second problem is solved in different ways by the methods described here.

This chapter is devoted to statistical compression algorithms, such as Shannon-Fano, Huffman, arithmetic coding, and PPM. It is recommended, however, that the reader start with the short presentation of information theory in the Appendix. This presentation covers the principles and important terms used by information theory, especially the terms redundancy and entropy. An understanding of these terms leads to a deeper understanding of statistical data compression, and also makes it possible to calculate how redundancy is reduced, or even eliminated, by the various methods.

### 5.1 Shannon-Fano Coding

Shannon-Fano coding, named after Claude Shannon and Robert Fano, was the first algorithm to construct a set of the best variable-length codes.

We start with a set of  $n$  symbols with known probabilities (or frequencies) of occurrence. The symbols are first arranged in descending order of their probabilities. The set of symbols is then divided into two subsets that have the same (or almost the same) probabilities. All symbols in one subset get assigned codes that start with a 0, while the codes of the symbols in the other subset start with a 1. Each subset is then recursively divided into two subsubsets of roughly equal probabilities, and the second bit of all the codes is determined in a similar way. When a subset contains just two symbols,

their codes are distinguished by adding one more bit to each. The process continues until no more subsets remain. Table 5.1 illustrates the Shannon-Fano algorithm for a seven-symbol alphabet. Notice that the symbols themselves are not shown, only their probabilities.

---

Robert M. Fano was Ford Professor of Engineering, in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology until his retirement. In 1963 he organized MIT's Project MAC (now the Computer Science and Artificial Intelligence Laboratory) and was its Director until September 1968. He also served as Associate Head of the Department of Electrical Engineering and Computer Science from 1971 to 1974.



Professor Fano chaired the Centennial Study Committee of the Department of Electrical Engineering and Computer Science whose report, "Lifelong Cooperative Education," was published in October, 1982.

Professor Fano was born in Torino, Italy, and did most of his undergraduate work at the School of Engineering of Torino before coming to the United States in 1939. He received the Bachelor of Science degree in 1941 and the Doctor of Science degree in 1947, both in Electrical Engineering from MIT. He has been a member of the MIT staff since 1941 and a member of its faculty since 1947.

During World War II, Professor Fano was on the staff of the MIT Radiation Laboratory, working on microwave components and filters. He was also group leader of the Radar Techniques Group of Lincoln Laboratory from 1950 to 1953. He has worked and published at various times in the fields of network theory, microwaves, electromagnetism, information theory, computers and engineering education. He is author of the book entitled *Transmission of Information*, and co-author of *Electromagnetic Fields, Energy and Forces* and *Electromagnetic Energy Transmission and Radiation*. He is also co-author of Volume 9 of the Radiation Laboratory Series.

---

The first step splits the set of seven symbols into two subsets, one with two symbols and a total probability of 0.45 and the other with the remaining five symbols and a total probability of 0.55. The two symbols in the first subset are assigned codes that start with 1, so their final codes are 11 and 10. The second subset is divided, in the second step, into two symbols (with total probability 0.3 and codes that start with 01) and three symbols (with total probability 0.25 and codes that start with 00). Step three divides the last three symbols into 1 (with probability 0.1 and code 001) and 2 (with total probability 0.15 and codes that start with 000).

The average size of this code is  $0.25 \times 2 + 0.20 \times 2 + 0.15 \times 3 + 0.15 \times 3 + 0.10 \times 3 + 0.10 \times 4 + 0.05 \times 4 = 2.7$  bits/symbol. This is a good result because the entropy (the smallest number of bits needed, on average, to represent each symbol) is

$$-(0.25 \log_2 0.25 + 0.20 \log_2 0.20 + 0.15 \log_2 0.15 + 0.15 \log_2 0.15 + 0.10 \log_2 0.10 + 0.10 \log_2 0.10 + 0.05 \log_2 0.05) \approx 2.67.$$

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.20	1	0			:10
3.	0.15	0		1	1	:011
4.	0.15	0		1	0	:010
5.	0.10	0		0	1	:001
6.	0.10	0		0	0	:0001
7.	0.05	0		0	0	:0000

Table 5.1: Shannon-Fano Example.

- ◇ **Exercise 5.1:** Repeat the calculation above but place the first split between the third and fourth symbols. Calculate the average size of the code and show that it is greater than 2.67 bits/symbol.

The code in the table in the answer to Exercise 5.1 has longer average size because the splits, in this case, were not as good as those of Table 5.1. This suggests that the Shannon-Fano method produces better code when the splits are better, i.e., when the two subsets in every split have very close total probabilities. Carrying this argument to its limit suggests that perfect splits yield the best code. Table 5.2 illustrates such a case. The two subsets in every split have identical total probabilities, yielding a code with the minimum average size (zero redundancy). Its average size is  $0.25 \times 2 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 + 0.125 \times 3 + 0.125 \times 3 = 2.5$  bits/symbols, which is identical to its entropy. This means that it is the theoretical minimum average size.

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.25	1	0			:10
3.	0.125	0	1	1		:011
4.	0.125	0	1	0		:010
5.	0.125	0	0	1		:001
6.	0.125	0	0	0		:000

Table 5.2: Shannon-Fano Balanced Example.

The conclusion is that this method produces the best results when the symbols have probabilities of occurrence that are (negative) powers of 2.

- ◇ **Exercise 5.2:** Compute the entropy of the codes of Table 5.2.

The Shannon-Fano method is easy to implement but the code it produces is generally not as good as that produced by the Huffman method (Section 5.2).

## 5.2 Huffman Coding

---

---

**David Huffman (1925–1999)**

Being originally from Ohio, it is no wonder that Huffman went to Ohio State University for his BS (in electrical engineering). What is unusual was his age (18) when he earned it in 1944. After serving in the United States Navy, he went back to Ohio State for an MS degree (1949) and then to MIT, for a PhD (1953, electrical engineering).



That same year, Huffman joined the faculty at MIT. In 1967, he made his only career move when he went to the University of California, Santa Cruz as the founding faculty member of the Computer Science Department. During his long tenure at UCSC, Huffman played a major role in the development of the department (he served as chair from 1970 to 1973) and he is known for his motto “my products are my students.” Even after his retirement, in 1994, he remained active in the department, teaching information theory and signal analysis courses.

Huffman made significant contributions in several areas, mostly information theory and coding, signal designs for radar and communications, and design procedures for asynchronous logical circuits. Of special interest is the well-known Huffman algorithm for constructing a set of optimal prefix codes for data with known frequencies of occurrence. At a certain point he became interested in the mathematical properties of “zero curvature” surfaces, and developed this interest into techniques for folding paper into unusual sculptured shapes (the so-called computational origami).

---

---

Huffman coding is a popular method for data compression. It serves as the basis for several popular programs run on various platforms. Some programs use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it produces the best code when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the codes from right to left). Since its development, in 1952, by D. Huffman, this method has been the subject of intensive research into data compression.

Since its development in 1952 by D. Huffman, this method has been the subject of intensive research in data compression. The long discussion in [Gilbert and Moore 59] proves that the Huffman code is a minimum-length code in the sense that no other encoding has a shorter average length. An algebraic approach to constructing the Huffman code is introduced in [Karp 61]. In [Gallager 74], Robert Gallager shows that the redundancy of Huffman coding is at most  $p_1 + 0.086$  where  $p_1$  is the probability of the most-common symbol in the alphabet. The redundancy is the difference between the average Huffman codeword length and the entropy. Given a large alphabet, such as the

set of letters, digits and punctuation marks used by a natural language, the largest symbol probability is typically around 15–20%, bringing the value of the quantity  $p_1 + 0.086$  to around 0.1. This means that Huffman codes are at most 0.1 bit longer (per symbol) than an ideal entropy encoder, such as arithmetic coding.

The Huffman algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols.

This process is best illustrated by an example. Given five symbols with probabilities as shown in Figure 5.3a, they are paired in the following order:

1.  $a_4$  is combined with  $a_5$  and both are replaced by the combined symbol  $a_{45}$ , whose probability is 0.2.
2. There are now four symbols left,  $a_1$ , with probability 0.4, and  $a_2$ ,  $a_3$ , and  $a_{45}$ , with probabilities 0.2 each. We arbitrarily select  $a_3$  and  $a_{45}$ , combine them, and replace them with the auxiliary symbol  $a_{345}$ , whose probability is 0.4.
3. Three symbols are now left,  $a_1$ ,  $a_2$ , and  $a_{345}$ , with probabilities 0.4, 0.2, and 0.4, respectively. We arbitrarily select  $a_2$  and  $a_{345}$ , combine them, and replace them with the auxiliary symbol  $a_{2345}$ , whose probability is 0.6.
4. Finally, we combine the two remaining symbols,  $a_1$  and  $a_{2345}$ , and replace them with  $a_{12345}$  with probability 1.

The tree is now complete. It is shown in Figure 5.3a “lying on its side” with its root on the right and its five leaves on the left. To assign the codes, we arbitrarily assign a bit of 1 to the top edge, and a bit of 0 to the bottom edge, of every pair of edges. This results in the codes 0, 10, 111, 1101, and 1100. The assignments of bits to the edges is arbitrary.

The average size of this code is  $0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$  bits/symbol, but even more importantly, the Huffman code is not unique. Some of the steps above were chosen arbitrarily, since there were more than two symbols with smallest probabilities. Figure 5.3b shows how the same five symbols can be combined differently to obtain a different Huffman code (11, 01, 00, 101, and 100). The average size of this code is  $0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2$  bits/symbol, the same as the previous code.

- ◇ **Exercise 5.3:** Given the eight symbols A, B, C, D, E, F, G, and H with probabilities  $1/30$ ,  $1/30$ ,  $1/30$ ,  $2/30$ ,  $3/30$ ,  $5/30$ ,  $5/30$ , and  $12/30$ , draw three different Huffman trees with heights 5 and 6 for these symbols and calculate the average code size for each tree.
- ◇ **Exercise 5.4:** Figure Ans.5d shows another Huffman tree, with height 4, for the eight symbols introduced in Exercise 5.3. Explain why this tree is wrong.

It turns out that the arbitrary decisions made in constructing the Huffman tree affect the individual codes but not the average size of the code. Still, we have to answer the obvious question, which of the different Huffman codes for a given set of symbols is best? The answer, while not obvious, is simple: The best code is the one with the

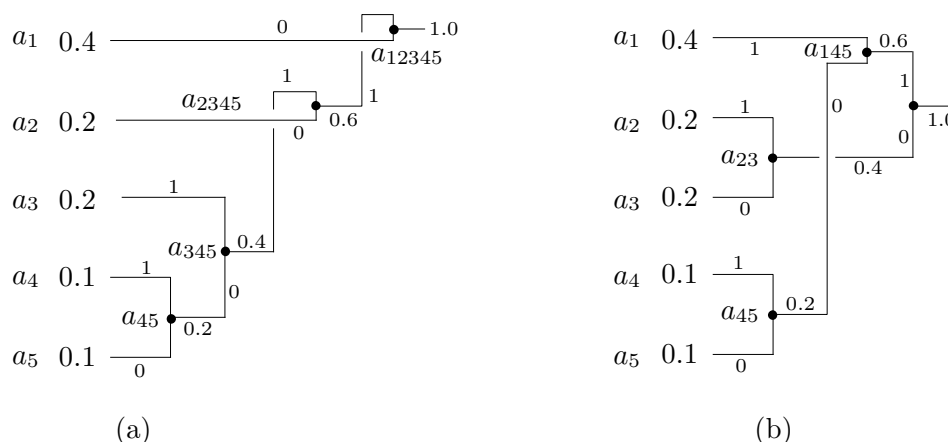


Figure 5.3: Huffman Codes.

smallest variance. The variance of a code measures how much the sizes of the individual codes deviate from the average size (see page 624 for the definition of variance). The variance of code 5.3a is

$$0.4(1 - 2.2)^2 + 0.2(2 - 2.2)^2 + 0.2(3 - 2.2)^2 + 0.1(4 - 2.2)^2 + 0.1(4 - 2.2)^2 = 1.36,$$

while the variance of code 5.3b is

$$0.4(2 - 2.2)^2 + 0.2(2 - 2.2)^2 + 0.2(2 - 2.2)^2 + 0.1(3 - 2.2)^2 + 0.1(3 - 2.2)^2 = 0.16.$$

Code 5.3b is therefore preferable (see below). A careful look at the two trees shows how to select the one we want. In the tree of Figure 5.3a, symbol  $a_{45}$  is combined with  $a_3$ , whereas in the tree of 5.3b it is combined with  $a_1$ . The rule is: When there are more than two smallest-probability nodes, select the ones that are lowest and highest in the tree and combine them. This will combine symbols of low probability with ones of high probability, thereby reducing the total variance of the code.

If the encoder simply writes the compressed stream on a file, the variance of the code makes no difference. A small-variance Huffman code is preferable only in cases where the encoder *transmits* the compressed stream, as it is being generated, over a communications line. In such a case, a code with large variance causes the encoder to generate bits at a rate that varies all the time. Since the bits have to be transmitted at a constant rate, the encoder has to use a buffer. Bits of the compressed stream are entered into the buffer as they are being generated and are moved out of it at a constant rate, to be transmitted. It is easy to see intuitively that a Huffman code with zero variance will enter bits into the buffer at a constant rate, so only a short buffer will be needed. The larger the code variance, the more variable is the rate at which bits enter the buffer, requiring the encoder to use a larger buffer.

The following claim is sometimes found in the literature:

It can be shown that the size of the Huffman code of a symbol  $a_i$  with probability  $P_i$  is always less than or equal to  $\lceil -\log_2 P_i \rceil$ .

Even though it is correct in many cases, this claim is not true in general. It seems to be a wrong corollary drawn by some authors from the Kraft-MacMillan inequality, Equation (2.3). The authors are indebted to Guy Blelloch for pointing this out and also for the example of Table 5.4.

- ◇ **Exercise 5.5:** Find an example where the size of the Huffman code of a symbol  $a_i$  is greater than  $\lceil -\log_2 P_i \rceil$ .

$P_i$	Code	$-\log_2 P_i$	$\lceil -\log_2 P_i \rceil$
.01	000	6.644	7
*.30	001	1.737	2
.34	01	1.556	2
.35	1	1.515	2

Table 5.4: A Huffman Code Example.

- ◇ **Exercise 5.6:** It seems that the size of a code must also depend on the number  $n$  of symbols (the size of the alphabet). A small alphabet requires just a few codes, so they can all be short; a large alphabet requires many codes, so some must be long. This being so, how can we say that the size of the code of symbol  $a_i$  depends just on its probability  $P_i$ ?

Figure 5.5 shows a Huffman code for the 26 letters.

As a self-exercise, the reader may calculate the average size, entropy, and variance of this code.

- ◇ **Exercise 5.7:** Discuss the Huffman codes for equal probabilities.

Exercise 5.7 shows that symbols with equal probabilities don't compress under the Huffman method. This is understandable, since strings of such symbols normally make random text, and random text does not compress. There may be special cases where strings of symbols with equal probabilities are not random and can be compressed. A good example is the string  $a_1 a_1 \dots a_1 a_2 a_2 \dots a_2 a_3 a_3 \dots$  in which each symbol appears in a long run. This string can be compressed with RLE but not with Huffman codes.

Notice that the Huffman method cannot be applied to a two-symbol alphabet. In such an alphabet, one symbol can be assigned the code 0 and the other code 1. The Huffman method cannot assign to any symbol a code shorter than one bit, so it cannot improve on this simple code. If the original data (the source) consists of individual bits, such as in the case of a bi-level (monochromatic) image, it is possible to combine several bits (perhaps four or eight) into a new symbol and pretend that the alphabet consists of these (16 or 256) symbols. The problem with this approach is that the original binary data may have certain statistical correlations between the bits, and some of these correlations would be lost when the bits are combined into symbols. When a typical bi-level image (a painting or a diagram) is digitized by scan lines, a pixel is more likely to be followed by an identical pixel than by the opposite one. We therefore have a file that can start with either a 0 or a 1 (each has 0.5 probability of being the first bit). A zero is more likely to be followed by another 0 and a 1 by another 1. Figure 5.6 is a finite-state

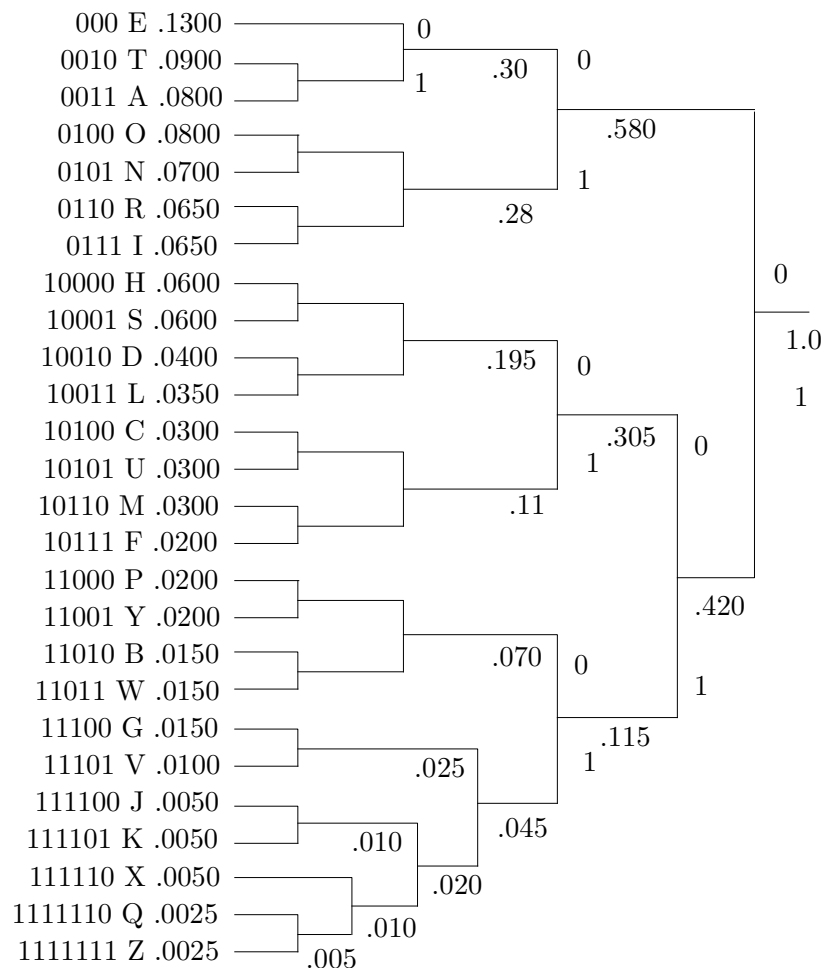


Figure 5.5: A Huffman Code for the 26-Letter Alphabet.

machine illustrating this situation. If these bits are combined into, say, groups of eight, the bits inside a group will still be correlated, but the groups themselves will not be correlated by the original pixel probabilities. If the input stream contains, e.g., the two adjacent groups 00011100 and 00001110, they will be encoded independently, ignoring the correlation between the last 0 of the first group and the first 0 of the next group. Selecting larger groups improves this situation but increases the number of groups, which implies more storage for the code table and longer time to calculate the table.

- ◇ **Exercise 5.8:** How does the number of groups increase when the group size increases from  $s$  bits to  $s + n$  bits?

A more complex approach to image compression by Huffman coding is to create several complete sets of Huffman codes. If the group size is, e.g., eight bits, then several sets of 256 codes are generated. When a symbol  $S$  is to be encoded, one of the sets is selected, and  $S$  is encoded using its code in that set. The choice of set depends on the



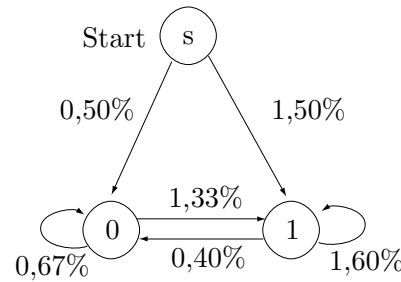


Figure 5.6: A Finite-State Machine.

symbol preceding S.

- ◇ **Exercise 5.9:** Imagine an image with 8-bit pixels where half the pixels have values 127 and the other half have values 128. Analyze the performance of RLE on the individual bitplanes of such an image, and compare it with what can be achieved with Huffman coding.

### 5.2.1 Dual Tree Coding

Dual tree coding, an idea due to G. H. Freeman ([Freeman 91] and [Freeman 93]), combines Tunstall and Huffman coding in an attempt to improve the latter's performance for a 2-symbol alphabet. The idea is to use the Tunstall algorithm to extend such an alphabet from 2 symbols to  $2^k$  strings of symbols, and select  $k$  such that the probabilities of the strings will be close to negative powers of 2. Once this is achieved, the strings are assigned Huffman codes and the input stream is compressed by replacing the strings with these codes. This approach is illustrated here by a simple example.

Given a binary source that emits two symbols  $a$  and  $b$  with probabilities 0.15 and 0.85, respectively, we try to compress it in four different ways as follows:

1. We apply the Huffman algorithm directly to the two symbols. This simply assigns the two 1-bit codes 0 and 1 to  $a$  and  $b$ , so there is no compression.
2. We combine the two symbols to obtain the four 2-symbol strings  $aa$ ,  $ab$ ,  $ba$ , and  $bb$ , with probabilities 0.0225, 0.1275, 0.1275, and 0.7225, respectively. The four strings are assigned Huffman codes as shown in Figure 5.7a, and it is obvious that the average code length is  $0.0225 \times 3 + 0.1275 \times 3 + 0.1275 \times 2 + 0.7225 \times 1 = 1.4275$  bits. On average, each 2-symbol string is compressed to 1.4275 bits, yielding a compression ratio of  $1.4275/2 \approx 0.714$ .
3. We apply Tunstall's algorithm to obtain the four strings  $bbb$ ,  $bba$ ,  $ba$ , and  $a$  with probabilities 0.614, 0.1084, 0.1275, and 0.15, respectively. The resulting parse tree is shown in Figure 5.7b. Tunstall's method compresses these strings by replacing each with a 2-bit code. Given a string of 257 bits with these probabilities, we expect the strings  $bbb$ ,  $bba$ ,  $ba$ , and  $a$  to occur 61, 11, 13, and 15 times, respectively, for a total of 100 strings. Thus, Tunstall's method compresses the 257 input bits to  $2 \times 100 = 200$  bits, for a compression ratio of  $200/257 \approx 0.778$ .
4. We now change the probabilities of the four strings above to negative powers of 2, because these are the best values for the Huffman method. Strings  $bbb$ ,  $bba$ ,

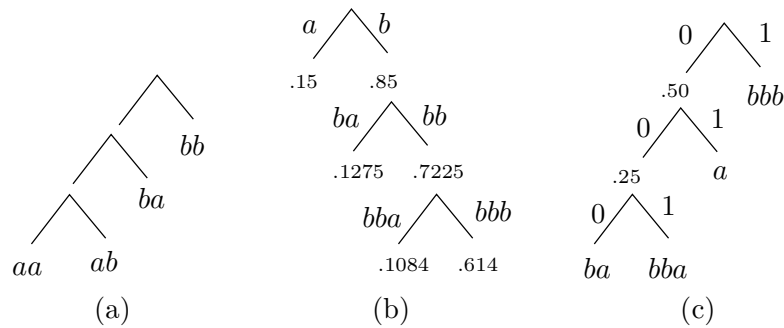


Figure 5.7: Dual Tree Coding.

$ba$ , and  $a$  are thus assigned the probabilities 0.5, 0.125, 0.125, and 0.25, respectively. The resulting Huffman code tree is shown in Figure 5.7c and it is easy to see that the 61, 11, 13, and 15 occurrences of these strings will be compressed to a total of  $61 \times 1 + 11 \times 3 + 13 \times 3 + 15 \times 2 = 163$  bits, resulting in a compression ratio of  $163/257 \approx 0.634$ , much better.

To summarize, applying the Huffman method to a 2-symbol alphabet produces no compression. Combining the individual symbols in strings as in 2 above or applying the Tunstall method as in 3, produce moderate compression. In contrast, combining the strings produced by Tunstall with the codes generated by the Huffman method, results in much better performance. The dual tree method starts by constructing the Tunstall parse tree and then using its leaf nodes to construct a Huffman code tree. The only (still unsolved) problem is determining the best value of  $k$ . In our example, we iterated the Tunstall algorithm until we had  $2^2 = 4$  strings, but iterating more times may have resulted in strings whose probabilities are closer to negative powers of 2.

### 5.2.2 Huffman Decoding

Before starting the compression of a data stream, the compressor (encoder) has to determine the codes. It does that based on the probabilities (or frequencies of occurrence) of the symbols. The probabilities or frequencies have to be written, as side information, on the compressed stream, so that any Huffman decompressor (decoder) will be able to decompress the stream. This is easy, since the frequencies are integers and the probabilities can be written as scaled integers. It normally adds just a few hundred bytes to the compressed stream. It is also possible to write the variable-length codes themselves on the stream, but this may be awkward, because the codes have different sizes. It is also possible to write the Huffman tree on the stream, but this may require more space than just the frequencies.

In any case, the decoder must know what is at the start of the stream, read it, and construct the Huffman tree for the alphabet. Only then can it read and decode the rest of the stream. The algorithm for decoding is simple. Start at the root and read the first bit off the compressed stream. If it is zero, follow the bottom edge of the tree; if it is one, follow the top edge. Read the next bit and move another edge toward the leaves of the tree. When the decoder gets to a leaf, it finds the original, uncompressed code of the symbol (normally its ASCII code), and that code is emitted by the decoder. The

process starts again at the root with the next bit.

This process is illustrated for the five-symbol alphabet of Figure 5.8. The four-symbol input string  $a_4a_2a_5a_1$  is encoded into 1001100111. The decoder starts at the root, reads the first bit 1, and goes up. The second bit 0 sends it down, as does the third bit. This brings the decoder to leaf  $a_4$ , which it emits. It again returns to the root, reads 110, moves up, up, and down, to reach leaf  $a_2$ , and so on.

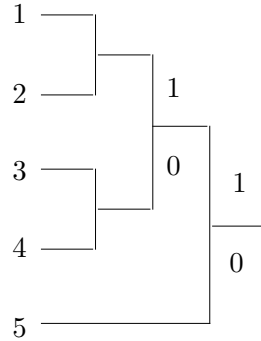


Figure 5.8: Huffman Codes for Equal Probabilities.

### 5.2.3 Fast Huffman Decoding

Decoding a Huffman-compressed file by sliding down the code tree for each symbol is conceptually simple, but slow. The compressed file has to be read bit by bit and the decoder has to advance a node in the code tree for each bit. The method of this section, originally conceived by [Choueka et al. 85] but later reinvented by others, uses preset partial-decoding tables. These tables depend on the particular Huffman code used, but not on the data to be decoded. The compressed file is read in chunks of  $k$  bits each (where  $k$  is normally 8 or 16 but can have other values) and the current chunk is used as a pointer to a table. The table entry that is selected in this way can decode several symbols and it also points the decoder to the table to be used for the next chunk.

As an example, consider the Huffman code of Figure 5.3a, where the five codewords are 0, 10, 111, 1101, and 1100. The string of symbols  $a_1a_1a_2a_4a_3a_1a_5\dots$  is compressed by this code to the string 0|0|10|1101|111|0|1100.... We select  $k = 3$  and read this string in 3-bit chunks 001|011|011|110|110|0.... Examining the first chunk, it is easy to see that it should be decoded into  $a_1a_1$  followed by the single bit 1 which is the prefix of another codeword. The first chunk is 001 =  $1_{10}$ , so we set entry 1 of the first table (table 0) to the pair  $(a_1a_1, 1)$ . When chunk 001 is used as a pointer to table 0, it points to entry 1, which immediately provides the decoder with the two decoded symbols  $a_1a_1$  and also directs it to use table 1 for the next chunk. Table 1 is used when a partially-decoded chunk ends with the single-bit prefix 1. The next chunk is 011 =  $3_{10}$ , so entry 3 of table 1 corresponds to the encoded bits 1|011. Again, it is easy to see that these should be decoded to  $a_2$  and there is the prefix 11 left over. Thus, entry 3 of table 1 should be  $(a_2, 2)$ . It provides the decoder with the single symbol  $a_2$  and also directs it to use table 2 next (the table that corresponds to prefix 11). The next chunk is again 011 =  $3_{10}$ , so entry 3 of table 2 corresponds to the encoded bits 11|011. It is again obvious that these

```

i ← 0; output ← null;
repeat
  j ← input next chunk;
  (s, i) ← Tablei[j];
  append s to output;
until end-of-input

```

Figure 5.9: Fast Huffman Decoding.

should be decoded to  $a_4$  with a prefix of 1 left over. This process continues until the end of the encoded input. Figure 5.9 is the simple decoding algorithm in pseudocode.

Table 5.10 lists the four tables required to decode this code. It is easy to see that they correspond to the prefixes  $\Lambda$  (null), 1, 11, and 110. A quick glance at Figure 5.3a shows that these correspond to the root and the four interior nodes of the Huffman code tree. Thus, each partial-decoding table corresponds to one of the four prefixes of this code. The number  $m$  of partial-decoding tables therefore equals the number of interior nodes (plus the root) which is one less than the number  $N$  of symbols of the alphabet.

$T_0 = \Lambda$	$T_1 = 1$	$T_2 = 11$	$T_3 = 110$
000 $a_1 a_1 a_1$ 0	1 000 $a_2 a_1 a_1$ 0	11 000 $a_5 a_1$ 0	110 000 $a_5 a_1 a_1$ 0
001 $a_1 a_1$ 1	1 001 $a_2 a_1$ 1	11 001 $a_5$ 1	110 001 $a_5 a_1$ 1
010 $a_1 a_2$ 0	1 010 $a_2 a_2$ 0	11 010 $a_4 a_1$ 0	110 010 $a_5 a_2$ 0
011 $a_1$ 2	1 011 $a_2$ 2	11 011 $a_4$ 1	110 011 $a_5$ 2
100 $a_2 a_1$ 0	1 100 $a_5$ 0	11 100 $a_3 a_1 a_1$ 0	110 100 $a_4 a_1 a_1$ 0
101 $a_2$ 1	1 101 $a_4$ 0	11 101 $a_3 a_1$ 1	110 101 $a_4 a_1$ 1
110 — 3	1 110 $a_3 a_1$ 0	11 110 $a_3 a_2$ 0	110 110 $a_4 a_2$ 0
111 $a_3$ 0	1 111 $a_3$ 1	11 111 $a_3$ 2	110 111 $a_4$ 2

Table 5.10: Partial-Decoding Tables for a Huffman Code.

Notice that some chunks (such as entry 110 of table 0) simply send the decoder to another table and do not provide any decoded symbols. Also, there is a tradeoff between chunk size (and thus table size) and decoding speed. Large chunks speed up decoding, but require large tables. A large alphabet (such as the 128 ASCII characters or the 256 8-bit bytes) also requires a large set of tables. The problem with large tables is that the decoder has to set up the tables after it has read the Huffman codes from the compressed stream and before decoding can start, and this process may preempt any gains in decoding speed provided by the tables.

To set up the first table (table 0, which corresponds to the null prefix  $\Lambda$ ), the decoder generates the  $2^k$  bit patterns 0 through  $2^k - 1$  (the first column of Table 5.10) and employs the decoding method of Section 5.2.2 to decode each pattern. This yields the second column of Table 5.10. Any remainders left are prefixes and are converted by the decoder to table numbers. They become the third column of the table. If no remainder is left, the third column is set to 0 (use table 0 for the next chunk). Each of the other partial-decoding tables is set in a similar way. Once the decoder decides that

table 1 corresponds to prefix  $p$ , it generates the  $2^k$  patterns  $p|00\dots 0$  through  $p|11\dots 1$  that become the first column of that table. It then decodes that column to generate the remaining two columns.

This method was conceived in 1985, when storage costs were considerably higher than today (early 2007). This prompted the developers of the method to find ways to cut down the number of partial-decoding tables, but these techniques are less important today and are not described here.

Truth is stranger than fiction, but this is because fiction is obliged to stick to probability; truth is not.

—Anonymous

### 5.2.4 Average Code Size

Figure 5.13a shows a set of five symbols with their probabilities and a typical Huffman tree. Symbol A appears 55% of the time and is assigned a 1-bit code, so it contributes  $0.55 \cdot 1$  bits to the average code size. Symbol E appears only 2% of the time and is assigned a 4-bit Huffman code, so it contributes  $0.02 \cdot 4 = 0.08$  bits to the code size. The average code size is therefore calculated to be

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7 \text{ bits per symbol.}$$

Surprisingly, the same result is obtained by adding the values of the four internal nodes of the Huffman code-tree  $0.05 + 0.2 + 0.45 + 1 = 1.7$ . This provides a way to calculate the average code size of a set of Huffman codes without any multiplications. Simply add the values of all the internal nodes of the tree. Table 5.11 illustrates why this works.

<i>.05</i>	=	.02+.03
<i>.20</i>	=	<i>.05</i> + .15 = .02+ .03+ .15
<i>.45</i>	=	<i>.20</i> + .25 = .02+ .03+ .15+ .25
<i>1.0</i>	=	<i>.45</i> + .55 = .02+ .03+ .15+ .25+ .55

Table 5.11: Composition of Nodes.

<i>0.05</i>	=		=	0.02 + 0.03 + ...
<i>a<sub>1</sub></i>	=	<i>0.05</i> + ...	=	0.02 + 0.03 + ...
<i>a<sub>2</sub></i>	=	<i>a<sub>1</sub></i> + ...	=	0.02 + 0.03 + ...
$\vdots$	=			
<i>a<sub>d-2</sub></i>	=	<i>a<sub>d-3</sub></i> + ...	=	0.02 + 0.03 + ...
<i>1.0</i>	=	<i>a<sub>d-2</sub></i> + ...	=	0.02 + 0.03 + ...

Table 5.12: Composition of Nodes.

(Internal nodes are shown in italics in this table.) The left column consists of the values of all the internal nodes. The right columns show how each internal node is the sum of some of the leaf nodes. Summing the values in the left column yields 1.7, and summing the other columns shows that this 1.7 is the sum of the four values 0.02, the four values 0.03, the three values 0.15, the two values 0.25, and the single value 0.55.

This argument can be extended to the general case. It is easy to show that, in a Huffman-like tree (a tree where each node is the sum of its children), the weighted sum

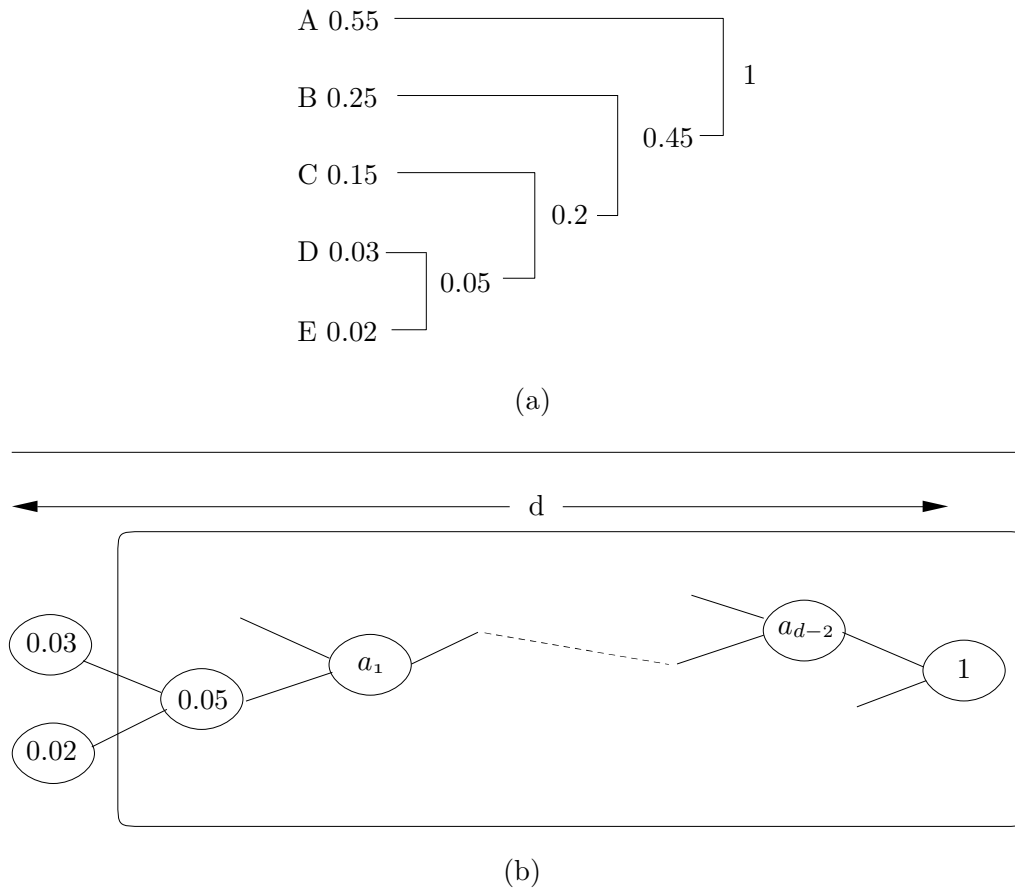


Figure 5.13: Huffman Code-Trees.

of the leaves, where the weights are the distances of the leaves from the root, equals the sum of the internal nodes. (This property has been communicated to us by John M. Motil.)

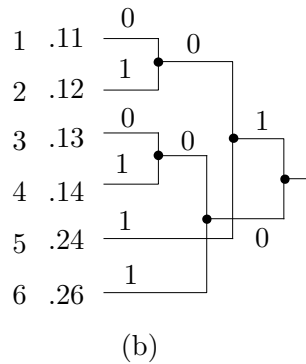
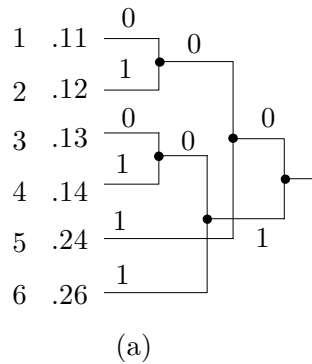
Figure 5.13b shows such a tree, where we assume that the two leaves 0.02 and 0.03 have  $d$ -bit Huffman codes. Inside the tree, these leaves become the children of internal node 0.05, which, in turn, is connected to the root by means of the  $d - 2$  internal nodes  $a_1$  through  $a_{d-2}$ . Table 5.12 has  $d$  rows and shows that the two values 0.02 and 0.03 are included in the various internal nodes exactly  $d$  times. Adding the values of all the internal nodes produces a sum that includes the contributions  $0.02 \cdot d + 0.03 \cdot d$  from the two leaves. Since these leaves are arbitrary, it is clear that this sum includes similar contributions from all the other leaves, so this sum is the average code size. Since this sum also equals the sum of the left column, which is the sum of the internal nodes, it is clear that the sum of the internal nodes equals the average code size.

Notice that this proof does not assume that the tree is binary. The property illustrated here exists for any tree where a node contains the sum of its children.

### 5.2.5 Number of Codes

Since the Huffman code is not unique, the natural question is: How many different codes are there? Figure 5.14a shows a Huffman code-tree for six symbols, from which we can answer this question in two different ways.

Answer 1. The tree of 5.14a has five interior nodes, and in general, a Huffman code-tree for  $n$  symbols has  $n - 1$  interior nodes. Each interior node has two edges coming out of it, labeled 0 and 1. Swapping the two labels produces a different Huffman code-tree, so the total number of different Huffman code-trees is  $2^{n-1}$  (in our example,  $2^5$  or 32). The tree of Figure 5.14b, for example, shows the result of swapping the labels of the two edges of the root. Table 5.15a,b lists the codes generated by the two trees.



000	100	000
001	101	001
100	000	010
101	001	011
01	11	10
11	01	11

(a) (b) (c)

Figure 5.14: Two Huffman Code-Trees.

Table 5.15.

Answer 2. The six codes of Table 5.15a can be divided into the four classes  $00x$ ,  $10y$ ,  $01$ , and  $11$ , where  $x$  and  $y$  are 1-bit each. It is possible to create different Huffman codes by changing the first two bits of each class. Since there are four classes, this is the same as creating all the permutations of four objects, something that can be done in  $4! = 24$  ways. In each of the 24 permutations it is also possible to change the values of  $x$  and  $y$  in four different ways (since they are bits) so the total number of different Huffman codes in our six-symbol example is  $24 \times 4 = 96$ .

The two answers are different because they count different things. Answer 1 counts the number of different Huffman code-trees, while answer 2 counts the number of different Huffman codes. It turns out that our example can generate 32 different code-trees but only 94 different codes instead of 96. This shows that there are Huffman codes that cannot be generated by the Huffman method! Table 5.15c shows such an example. A look at the trees of Figure 5.14 should convince the reader that the codes of symbols 5 and 6 must start with different bits, but in the code of Table 5.15c they both start with 1. This code is therefore impossible to generate by any relabeling of the nodes of the trees of Figure 5.14.

### 5.2.6 Ternary Huffman Codes

The Huffman code is not unique. Moreover, it does not have to be binary! The Huffman method can easily be applied to codes based on other number systems. Figure 5.16a

shows a Huffman code tree for five symbols with probabilities 0.15, 0.15, 0.2, 0.25, and 0.25. The average code size is

$$2 \times 0.25 + 3 \times 0.15 + 3 \times 0.15 + 2 \times 0.20 + 2 \times 0.25 = 2.3 \text{ bits/symbol.}$$

Figure 5.16b shows a ternary Huffman code tree for the same five symbols. The tree is constructed by selecting, at each step, three symbols with the smallest probabilities and merging them into one parent symbol, with the combined probability. The average code size of this tree is

$$2 \times 0.15 + 2 \times 0.15 + 2 \times 0.20 + 1 \times 0.25 + 1 \times 0.25 = 1.5 \text{ trits/symbol.}$$

Notice that the ternary codes use the digits 0, 1, and 2.

- ◇ **Exercise 5.10:** Given seven symbols with probabilities .02, .03, .04, .04, .12, .26, and .49, we construct binary and ternary Huffman code-trees for them and calculate the average code size in each case.

### 5.2.7 Height Of A Huffman Tree

The height of the code-tree generated by the Huffman algorithm may sometimes be important because the height is also the length of the longest code in the tree. The Deflate method (Section 6.25), for example, limits the lengths of certain Huffman codes to just three bits.

It is easy to see that the shortest Huffman tree is created when the symbols have equal probabilities. If the symbols are denoted by A, B, C, and so on, then the algorithm combines pairs of symbols, such as A and B, C and D, in the lowest level, and the rest of the tree consists of interior nodes as shown in Figure 5.17a. The tree is balanced or close to balanced and its height is  $\lceil \log_2 n \rceil$ . In the special case where the number of symbols  $n$  is a power of 2, the height is exactly  $\log_2 n$ . In order to generate the tallest tree, we need to assign probabilities to the symbols such that each step in the Huffman method will increase the height of the tree by 1. Recall that each step in the Huffman algorithm combines two symbols. Thus, the tallest tree is obtained when the first step combines two of the  $n$  symbols and each subsequent step combines the result of its predecessor with one of the remaining symbols (Figure 5.17b). The height of the complete tree is therefore  $n - 1$ , and it is referred to as a lopsided or unbalanced tree.

It is easy to see what symbol probabilities result in such a tree. Denote the two smallest probabilities by  $a$  and  $b$ . They are combined in the first step to form a node whose probability is  $a + b$ . The second step will combine this node with an original symbol if one of the symbols has probability  $a + b$  (or smaller) and all the remaining symbols have greater probabilities. Thus, after the second step, the root of the tree has probability  $a + b + (a + b)$  and the third step will combine this root with one of the remaining symbols if its probability is  $a + b + (a + b)$  and the probabilities of the remaining  $n - 4$  symbols are greater. It does not take much to realize that the symbols have to have probabilities  $p_1 = a$ ,  $p_2 = b$ ,  $p_3 = a + b = p_1 + p_2$ ,  $p_4 = b + (a + b) = p_2 + p_3$ ,  $p_5 = (a + b) + (a + 2b) = p_3 + p_4$ ,  $p_6 = (a + 2b) + (2a + 3b) = p_4 + p_5$ , and so on (Figure 5.17c). These probabilities form a Fibonacci sequence whose first two elements



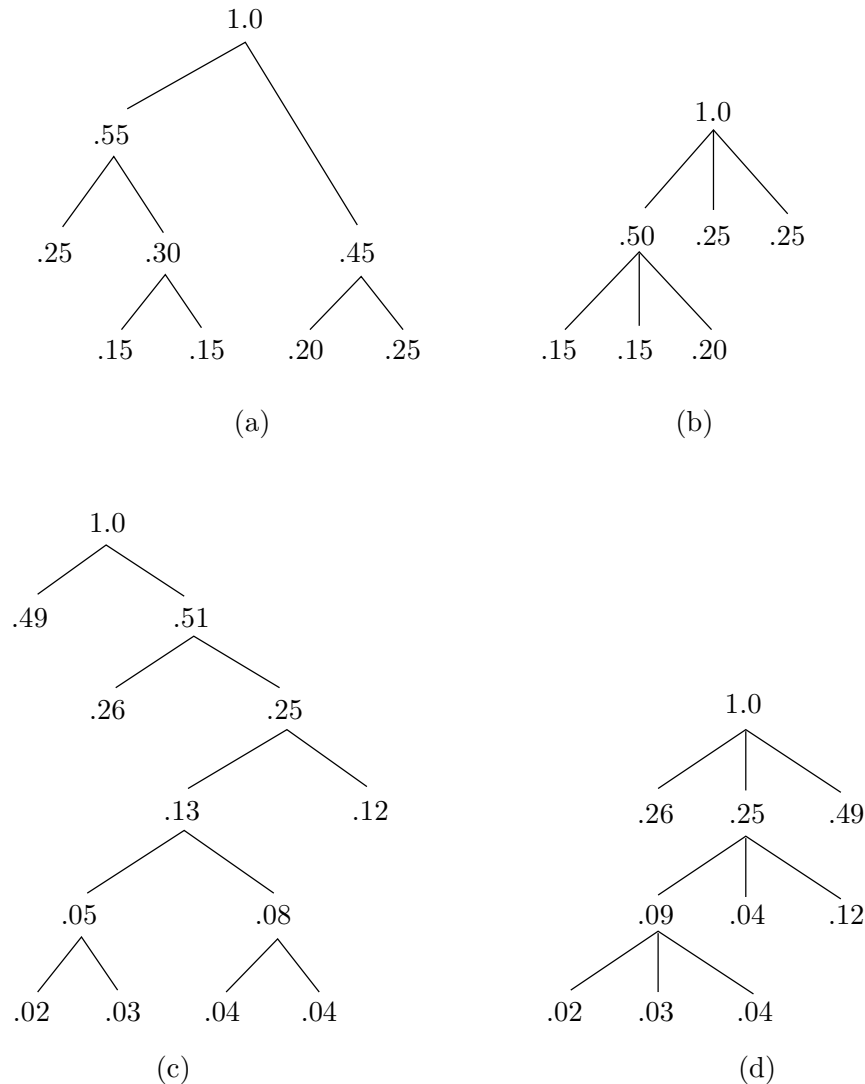


Figure 5.16: Binary and Ternary Huffman Code-Trees.

are  $a$  and  $b$ . As an example, we select  $a = 5$  and  $b = 2$  and generate the 5-number Fibonacci sequence 5, 2, 7, 9, and 16. These five numbers add up to 39, so dividing them by 39 produces the five probabilities  $5/39$ ,  $2/39$ ,  $7/39$ ,  $9/39$ , and  $15/39$ . The Huffman tree generated by them has a maximal height (which is 4).

In principle, symbols in a set can have any probabilities, but in practice, the probabilities of symbols in an input file are computed by counting the number of occurrences of each symbol. Imagine a text file where only the nine symbols A through I appear. In order for such a file to produce the tallest Huffman tree, where the codes will have lengths from 1 to 8 bits, the frequencies of occurrence of the nine symbols have to form a Fibonacci sequence of probabilities. This happens when the frequencies of the symbols are 1, 1, 2, 3, 5, 8, 13, 21, and 34 (or integer multiples of these). The sum of these

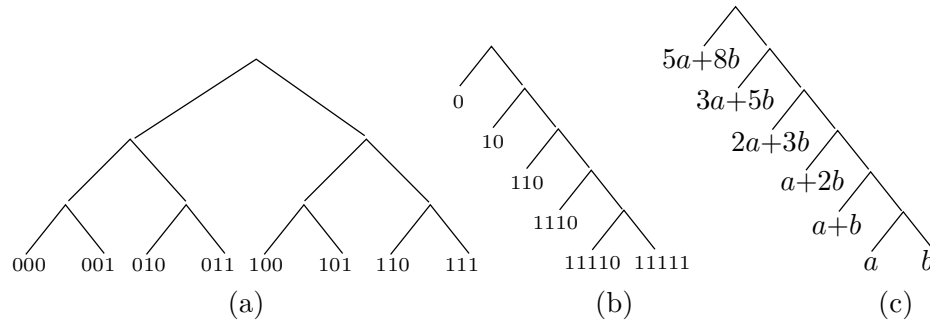


Figure 5.17: Shortest and Tallest Huffman Trees.

frequencies is 88, so our file has to be at least that long in order for a symbol to have 8-bit Huffman codes. Similarly, if we want to limit the sizes of the Huffman codes of a set of  $n$  symbols to 16 bits, we need to count frequencies of at least 4180 symbols. To limit the code sizes to 32 bits, the minimum data size is 9,227,464 symbols.

If a set of symbols happens to have the Fibonacci probabilities and therefore results in a maximal-height Huffman tree with codes that are too long, the tree can be reshaped (and the maximum code length shortened) by slightly modifying the symbol probabilities, so they are not much different from the original, but do not form a Fibonacci sequence.

### 5.2.8 Canonical Huffman Codes

The code of Table 5.15c has a simple interpretation. It assigns the first four symbols the 3-bit codes 0, 1, 2, 3, and the last two symbols the 2-bit codes 2 and 3. This is an example of a *canonical Huffman code*. The word “canonical” means that this particular code has been selected from among the several (or even many) possible Huffman codes because its properties make it easy and fast to use.

Table 5.18 shows a slightly bigger example of a canonical Huffman code. Imagine a set of 16 symbols (whose probabilities are irrelevant and are not shown) such that four symbols are assigned 3-bit codes, five symbols are assigned 5-bit codes, and the remaining seven symbols are assigned 6-bit codes. Table 5.18a shows a set of possible Huffman codes, while Table 5.18b shows a set of canonical Huffman codes. It is easy to see that the seven 6-bit canonical codes are simply the 6-bit integers 0 through 6. The five codes are the 5-bit integers 4 through 8, and the four codes are the 3-bit integers 3 through 6. We first show how these codes are generated and then how they are used.

The top row (length) of Table 5.19 lists the possible code lengths, from 1 to 6 bits. The second row (numl) lists the number of codes of each length, and the bottom row (first) lists the first code in each group. This is why the three groups of codes start with values 3, 4, and 0. To obtain the top two rows we need to compute the lengths of all the Huffman codes for the given alphabet (see below). The third row is computed by setting `first[6]:=0`; and iterating

```
for l:=5 downto 1 do first[l]:=(first[l+1]+numl[l+1])/2;
```

This guarantees that all the 3-bit prefixes of codes longer than three bits will be less than `first[3]` (which is 3), all the 5-bit prefixes of codes longer than five bits will be less than `first[5]` (which is 4), and so on.

1: 000 011	9: 10100 01000
2: 001 100	10: 101010 000000
3: 010 101	11: 101011 000001
4: 011 110	12: 101100 000010
5: 10000 00100	13: 101101 000011
6: 10001 00101	14: 101110 000100
7: 10010 00110	15: 101111 000101
8: 10011 00111	16: 110000 000110
(a) (b)	(a) (b)

Table 5.18.

length:	1	2	3	4	5	6
numl:	0	0	4	0	5	7
first:	2	4	3	5	4	0

Table 5.19.

Now for the use of these unusual codes. Canonical Huffman codes are useful in cases where the alphabet is large and where fast decoding is mandatory. Because of the way the codes are constructed, it is easy for the decoder to identify the length of a code by reading and examining input bits one by one. Once the length is known, the symbol can be found in one step. The pseudocode listed here shows the rules for decoding:

```

l:=1; input v;
while v<first[l]
append next input bit to v; l:=l+1;
endwhile

```

As an example, suppose that the next code is 00110. As bits are input and appended to  $v$ , it goes through the values 0, 00=0, 001=1, 0011=3, 00110=6, while  $l$  is incremented from 1 to 5. All steps except the last satisfy  $v < \text{first}[l]$ , so the last step determines the value of  $l$  (the code length) as 5. The symbol itself is found by subtracting  $v - \text{first}[5] = 6 - 4 = 2$ , so it is the third symbol (numbering starts at 0) in group  $l = 5$  (symbol 7 of the 16 symbols).

It has been mentioned that canonical Huffman codes are useful in cases where the alphabet is large and fast decoding is important. A practical example is a collection of documents archived and compressed by a *word-based* adaptive Huffman coder (Section 11.6.1). In an archive a slow encoder is acceptable, but the decoder should be fast. When the individual symbols are words, the alphabet may be huge, making it impractical, or even impossible, to construct the Huffman code-tree. However, even with a huge alphabet, the number of different code lengths is small, rarely exceeding 20 bits (just the number of 20-bit codes is about a million). If canonical Huffman codes are used, and the maximum code length is  $L$ , then the code length  $l$  of a symbol is found by the decoder in at most  $L$  steps, and the symbol itself is identified in one more step.

He uses statistics as a drunken man uses lampposts—for support rather than illumination.

—Andrew Lang, *Treasury of Humorous Quotations*

The last point to be discussed is the encoder. In order to construct the canonical Huffman code, the encoder needs to know the length of the Huffman code of every symbol. The main problem is the large size of the alphabet, which may make it impractical or even impossible to build the entire Huffman code-tree in memory. The algorithm

described here (see [Hirschberg and Lelewer 90] and [Sieminski 88]) solves this problem. It calculates the code sizes for an alphabet of  $n$  symbols using just one array of size  $2n$ . One half of this array is used as a *heap*, so we start with a short description of this useful data structure.

A *binary tree* is a tree where every node has at most two children (i.e., it may have 0, 1, or 2 children). A *complete binary tree* is a binary tree where every node except the leaves has exactly two children. A *balanced binary tree* is a complete binary tree where some of the bottom-right nodes may be missing (see also page 276 for another application of those trees). A *heap* is a balanced binary tree where every leaf contains a data item and the items are ordered such that every path from a leaf to the root traverses nodes that are in sorted order, either nondecreasing (a max-heap) or nonincreasing (a min-heap). Figure 5.20 shows examples of min-heaps.

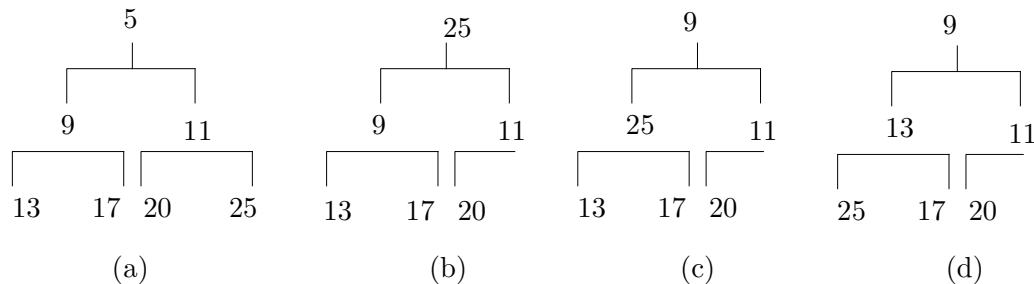


Figure 5.20: Min-Heaps.

A common operation on a heap is to remove the root and rearrange the remaining nodes to get back a heap. This is called *sifting* the heap. The four parts of Figure 5.20 show how a heap is sifted after the root (with data item 5) has been removed. Sifting starts by moving the bottom-right node to become the new root. This guarantees that the heap will remain a balanced binary tree. The root is then compared with its children and may have to be swapped with one of them in order to preserve the ordering of a heap. Several more swaps may be necessary to completely restore heap ordering. It is easy to see that the maximum number of swaps equals the height of the tree, which is  $\lceil \log_2 n \rceil$ .

The reason a heap must always remain balanced is that this makes it possible to store it in memory without using any pointers. The heap is said to be “housed” in an array. To house a heap in an array, the root is placed in the first array location (with index 1), the two children of the node at array location  $i$  are placed at locations  $2i$  and  $2i + 1$ , and the parent of the node at array location  $j$  is placed at location  $\lfloor j/2 \rfloor$ . Thus the heap of Figure 5.20a is housed in an array by placing the nodes 5, 9, 11, 13, 17, 20, and 25 in the first seven locations of the array.

The algorithm uses a single array  $A$  of size  $2n$ . The frequencies of occurrence of the  $n$  symbols are placed in the top half of  $A$  (locations  $n + 1$  through  $2n$ ), and the bottom half of  $A$  (locations 1 through  $n$ ) becomes a min-heap whose data items are pointers to the frequencies in the top half (Figure 5.21a). The algorithm then goes into a loop where in each iteration the heap is used to identify the two smallest frequencies and replace them with their sum. The sum is stored in the last heap position  $A[h]$ , and the heap

shrinks by one position (Figure 5.21b). The loop repeats until the heap is reduced to just one pointer (Figure 5.21c).

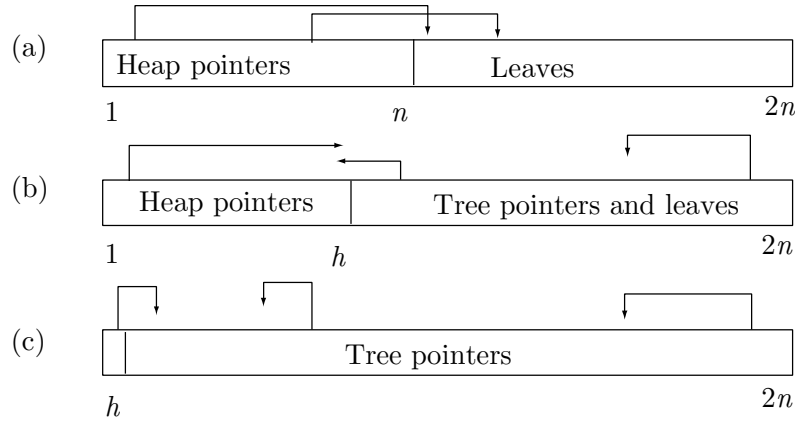


Figure 5.21: Huffman Heaps and Leaves in an Array.

We now illustrate this part of the algorithm using seven frequencies. The table below shows how the frequencies and the heap are initially housed in an array of size 14. Pointers are shown in *italics*, and the heap is delimited by square brackets.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[14	12	13	10	11	9	8]	25	20	13	17	9	11	5

The first iteration selects the smallest frequency (5), removes the root of the heap (pointer 14), and leaves  $A[7]$  empty.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[12	10	13	8	11	9]		25	20	13	17	9	11	5

The heap is sifted, and its new root (12) points to the second smallest frequency (9) in  $A[12]$ . The sum  $5 + 9$  is stored in the empty location 7, and the three array locations  $A[1]$ ,  $A[12]$ , and  $A[14]$  are set to point to that location.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[7	10	13	8	11	9]	5+9	25	20	13	17	7	11	7

The heap is now sifted.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[13	10	7	8	11	9]	14	25	20	13	17	7	11	7

The new root is 13, implying that the smallest frequency (11) is stored at  $A[13]$ . The root is removed, and the heap shrinks to just five positions, leaving location 6 empty.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[10	11	7	8	9]		14	25	20	13	17	7	11	7

The heap is now sifted. The new root is 10, showing that the second smallest frequency, 13, is stored at  $A[10]$ . The sum  $11 + 13$  is stored at the empty location 6, and the three locations  $A[1]$ ,  $A[13]$ , and  $A[10]$  are set to point to 6.

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>		<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
[6	11	7	8	9]	11+13	14	25	20	6	17	7	6	7	

- ◇ **Exercise 5.11:** Continue this loop.
- ◇ **Exercise 5.12:** Complete this loop.
- ◇ **Exercise 5.13:** Find the lengths of all the other codes.

Considine's Law. Whenever one word or letter can change the entire meaning of a sentence, the probability of an error being made will be in direct proportion to the embarrassment it will cause.

—Bob Considine

### 5.2.9 Is Huffman Coding Dead?

The advantages of arithmetic coding are well known to users of compression algorithms. Arithmetic coding can compress data to its entropy, its adaptive version works well if fed the correct probabilities, and its performance does not depend on the size of the alphabet. On the other hand, arithmetic coding is slower than Huffman coding, its compression potential is not always utilized to its maximum, its adaptive version is very sensitive to the symbol probabilities and in extreme cases may even expand the data. Finally, arithmetic coding is not robust; a single error may propagate indefinitely and may result in wrong decoding of a large quantity of compressed data. (Some users may complain that they don't understand arithmetic coding and have no idea how to implement it, but this doesn't seem a serious concern, because implementations of this method are available for all major computing platforms.) A detailed comparison and analysis of both methods is presented in [Bookstein and Klein 93], with the conclusion that arithmetic coding has the upper hand only in rare situations.

In [Gallager 74], Robert Gallager shows that the redundancy of Huffman coding is at most  $p_1 + 0.086$  where  $p_1$  is the probability of the most-common symbol in the alphabet. The redundancy is the difference between the average Huffman codeword length and the entropy. Since arithmetic coding can compress data to its entropy, the quantity  $p_1 + 0.086$  indicates by how much arithmetic coding outperforms Huffman coding. Given a two-symbol alphabet, the more probable symbol appears with probability 0.5 or more, but given a large alphabet, such as the set of letters, digits and punctuation marks used by a language, the largest symbol probability is typically around 15–20%, bringing the value of the quantity  $p_1 + 0.086$  to around 0.1. This means that Huffman codes are at most 0.1 bit longer (per symbol) than arithmetic coding. For some (perhaps even many) applications, such a small difference may be insignificant, but those applications for which this difference is significant may be important.

Bookstein and Klein examine the two extreme cases of large and small alphabets. Given a text file in a certain language, it is often compressed in blocks. This limits the propagation of errors and also provides several entry points into the file. The authors examine the probabilities of characters of several large alphabets (each consisting of the letters and punctuation marks of a natural language), and list the average codeword length for Huffman and arithmetic coding (the latter is the size of the compressed file divided by the number of characters in the original file). The surprising conclusion is that the Huffman codewords are longer than the arithmetic codewords by less than one percent. Also, arithmetic coding performs better than Huffman coding only in large blocks of text. The minimum block size where arithmetic coding is preferable turns out to be between 269 and 457 characters. Thus, for shorter blocks, Huffman coding outperforms arithmetic coding.

The other extreme case is a binary alphabet where one symbol has probability  $e$  and the other has probability  $1 - e$ . If  $e = 0.5$ , no method will compress the data. If the probabilities are skewed, Huffman coding does a bad job. The Huffman codes of the two symbols are 0 and 1 independent of the symbols' probabilities. Each code is 1-bit long, and there is no compression. Arithmetic coding, on the other hand, compresses such data to its entropy, which is  $-[e \log_2 e + (1 - e) \log_2 (1 - e)]$ . This expression tends to 0 for both small  $e$  (close to 0) and for large  $e$  (close to 1). However, there is a simple way to improve the performance of Huffman coding in this case. Simply group several bits into a word. If we group the bits in 4-bit words, we end up with an alphabet of 16 symbols, where the probabilities are less skewed and the Huffman codes do a better job, especially because of the Gallager bound.

Another difference between Huffman and arithmetic coding is the case of wrong probabilities. This is especially important when a compression algorithm employs a mathematical model to estimate the probabilities of occurrence of individual symbols. The authors show that, under reasonable assumptions, arithmetic coding is affected by wrong probabilities more than Huffman coding.

Speed is also an important consideration in many applications. Huffman encoding is fast. Given a symbol to encode, the symbol is used as a pointer to a code table, the Huffman code is read from the table, and is appended to the codes-so-far. Huffman decoding is slower because the decoder has to start at the root of the Huffman code tree and slide down, guided by the bits of the current codeword, until it reaches a leaf node, where it finds the symbol. Arithmetic coding, on the other hand, requires multiplications and divisions, and is therefore slower. (Notice, however, that certain versions of arithmetic coding, most notably the Q-coder, MQ-coder, and QM-coder, have been developed specifically to avoid slow operations and are not slow.)

Often, a data compression application requires a certain amount of robustness against transmission errors. Neither Huffman nor arithmetic coding is robust, but it is known from long experience that Huffman codes tend to synchronize themselves fairly quickly following an error, in contrast to arithmetic coding, where an error may propagate to the end of the compressed file. It is also possible to construct resynchronizing Huffman codes, as shown in Section 4.4.

The conclusion is that Huffman coding, being fast, simple, and effective, is preferable to arithmetic coding for most applications. Arithmetic coding is the method of choice only in cases where the alphabet has skewed probabilities that cannot be redefined.

## 5.3 Adaptive Huffman Coding

The Huffman method assumes that the frequencies of occurrence of all the symbols of the alphabet are known to the compressor. In practice, the frequencies are seldom, if ever, known in advance. One approach to this problem is for the compressor to read the original data twice. The first time, it just calculates the frequencies. The second time, it compresses the data. Between the two passes, the compressor constructs the Huffman tree. Such a method is called *semiadaptive* (page 10) and is normally too slow to be practical. The method that is used in practice is called *adaptive* (or *dynamic*) Huffman coding. This method is the basis of the UNIX `compact` program. (See also Section 11.6.1 for a word-based version of adaptive Huffman coding.) The method was originally developed by [Faller 73] and [Gallager 78] with substantial improvements by [Knuth 85].

The main idea is for the compressor and the decompressor to start with an empty Huffman tree and to modify it as symbols are being read and processed (in the case of the compressor, the word “processed” means compressed; in the case of the decompressor, it means decompressed). The compressor and decompressor should modify the tree in the same way, so at any point in the process they should use the same codes, although those codes may change from step to step. We say that the compressor and decompressor are *synchronized* or that they work in *lockstep* (although they don’t necessarily work together; compression and decompression normally take place at different times). The term *mirroring* is perhaps a better choice. The decoder mirrors the operations of the encoder.

Initially, the compressor starts with an empty Huffman tree. No symbols have been assigned codes yet. The first symbol being input is simply written on the output stream in its uncompressed form. The symbol is then added to the tree and a code assigned to it. The next time this symbol is encountered, its current code is written on the stream, and its frequency incremented by one. Since this modifies the tree, it (the tree) is examined to see whether it is still a Huffman tree (best codes). If not, it is rearranged, which results in changing the codes (Section 5.3.2).

The decompressor mirrors the same steps. When it reads the uncompressed form of a symbol, it adds it to the tree and assigns it a code. When it reads a compressed (variable-length) code, it scans the current tree to determine what symbol the code belongs to, and it increments the symbol’s frequency and rearranges the tree in the same way as the compressor.

The only subtle point is that the decompressor needs to know whether the item it has just input is an uncompressed symbol (normally, an 8-bit ASCII code, but see Section 5.3.1) or a variable-length code. To remove any ambiguity, each uncompressed symbol is preceded by a special, variable-length *escape code*. When the decompressor reads this code, it knows that the next 8 bits are the ASCII code of a symbol that appears in the compressed stream for the first time.

The trouble is that the escape code should not be any of the variable-length codes used for the symbols. These codes, however, are being modified every time the tree is rearranged, which is why the escape code should also be modified. A natural way to do this is to add an empty leaf to the tree, a leaf with a zero frequency of occurrence, that’s always assigned to the 0-branch of the tree. Since the leaf is in the tree, it



gets a variable-length code assigned. This code is the escape code preceding every uncompressed symbol. As the tree is being rearranged, the position of the empty leaf—and thus its code—change, but this escape code is always used to identify uncompressed symbols in the compressed stream. Figure 5.22 shows how the escape code moves and changes as the tree grows.

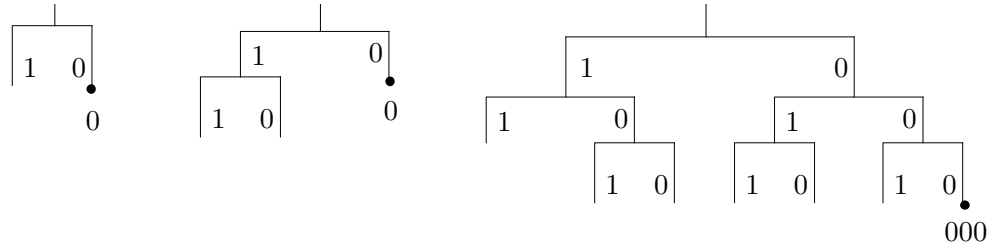


Figure 5.22: The Escape Code.

This method was used to compress/decompress data in the V.32 protocol for 14,400-baud modems.

Escape is not his plan. I must face him. Alone.

—David Prowse as Lord Darth Vader in *Star Wars* (1977)

### 5.3.1 Uncompressed Codes

If the symbols being compressed are ASCII characters, they may simply be assigned their ASCII codes as uncompressed codes. In the general case where there may be any symbols, the phased-in codes of Section 2.9 may be used.

Given  $n$  data symbols, where  $n = 2^m$  (implying that  $m = \log_2 n$ ), we can assign them  $m$ -bit codewords. However, if  $2^{m-1} < n < 2^m$ , then  $\log_2 n$  is not an integer. If we assign fixed-length codes to the symbols, each codeword would be  $\lceil \log_2 n \rceil$  bits long, but not all the codewords would be used. The case  $n = 1,000$  is a good example. In this case, each fixed-length codeword is  $\lceil \log_2 1,000 \rceil = 10$  bits long, but only 1,000 out of the 1,024 possible codewords are used.

The idea of phased-in codes is to try to assign two sets of codes to the  $n$  symbols, where the codewords of one set are  $m - 1$  bits long and may have several prefixes and the codewords of the other set are  $m$  bits long and have different prefixes. The average length of such a code is between  $m - 1$  and  $m$  bits and is shorter when there are more short codewords. See Section 2.9 for more details and examples.

### 5.3.2 Modifying the Tree

The main idea is to check the tree each time a symbol is input. If the tree is no longer a Huffman tree, it should be updated. A glance at Figure 5.23a shows what it means for a binary tree to be a Huffman tree. The tree in the figure contains five symbols:  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $E$ . It is shown with the symbols and their frequencies (in parentheses) after 16 symbols have been input and processed. The property that makes it a Huffman tree is that if we scan it level by level, scanning each level from left to right, and going from the bottom (the leaves) to the top (the root), the frequencies will be in sorted,

nondescending order. Thus, the bottom left node ( $A$ ) has the lowest frequency, and the top right node (the root) has the highest frequency. This is called the *sibling property*.

◇ **Exercise 5.14:** Why is this the criterion for a tree to be a Huffman tree?

Here is a summary of the operations needed to update the tree. The loop starts at the current node (the one corresponding to the symbol just input). This node is a leaf that we denote by  $X$ , with frequency of occurrence  $F$ . Each iteration of the loop involves three steps as follows:

1. Compare  $X$  to its successors in the tree (from left to right and bottom to top). If the immediate successor has frequency  $F + 1$  or greater, the nodes are still in sorted order and there is no need to change anything. Otherwise, some successors of  $X$  have identical frequencies of  $F$  or smaller. In this case,  $X$  should be swapped with the last node in this group (except that  $X$  should not be swapped with its parent).
2. Increment the frequency of  $X$  from  $F$  to  $F + 1$ . Increment the frequencies of all its parents.
3. If  $X$  is the root, the loop stops; otherwise, the loop repeats with the parent of node  $X$ .

Figure 5.23b shows the tree after the frequency of node  $A$  has been incremented from 1 to 2. It is easy to follow the three rules above to see how incrementing the frequency of  $A$  results in incrementing the frequencies of all its parents. No swaps are needed in this simple case because the frequency of  $A$  hasn't exceeded the frequency of its immediate successor  $B$ . Figure 5.23c shows what happens when  $A$ 's frequency has been incremented again, from 2 to 3. The three nodes following  $A$ , namely,  $B$ ,  $C$ , and  $D$ , have frequencies of 2, so  $A$  is swapped with the last of them,  $D$ . The frequencies of the new parents of  $A$  are then incremented, and each is compared with its successor, but no more swaps are needed.

Figure 5.23d shows the tree after the frequency of  $A$  has been incremented to 4. Once we decide that  $A$  is the current node, its frequency (which is still 3) is compared to that of its successor (4), and the decision is not to swap.  $A$ 's frequency is incremented, followed by incrementing the frequencies of its parents.

In Figure 5.23e,  $A$  is again the current node. Its frequency (4) equals that of its successor, so they should be swapped. This is shown in Figure 5.23f, where  $A$ 's frequency is 5. The next loop iteration examines the parent of  $A$ , with frequency 10. It should be swapped with its successor  $E$  (with frequency 9), which leads to the final tree of Figure 5.23g.

### 5.3.3 Counter Overflow

The frequency counts are accumulated in the Huffman tree in fixed-length fields, and such fields may overflow. A 16-bit unsigned field can accommodate counts of up to  $2^{16} - 1 = 65,535$ . A simple solution to the counter overflow problem is to watch the count field of the root each time it is incremented, and when it reaches its maximum value, to *rescale* all the frequency counts by dividing them by 2 (integer division). In practice, this is done by dividing the count fields of the leaves, then updating the counts of the interior nodes. Each interior node gets the sum of the counts of its children. The problem is that the counts are integers, and integer division reduces precision. This may change a Huffman tree to one that does not satisfy the sibling property.

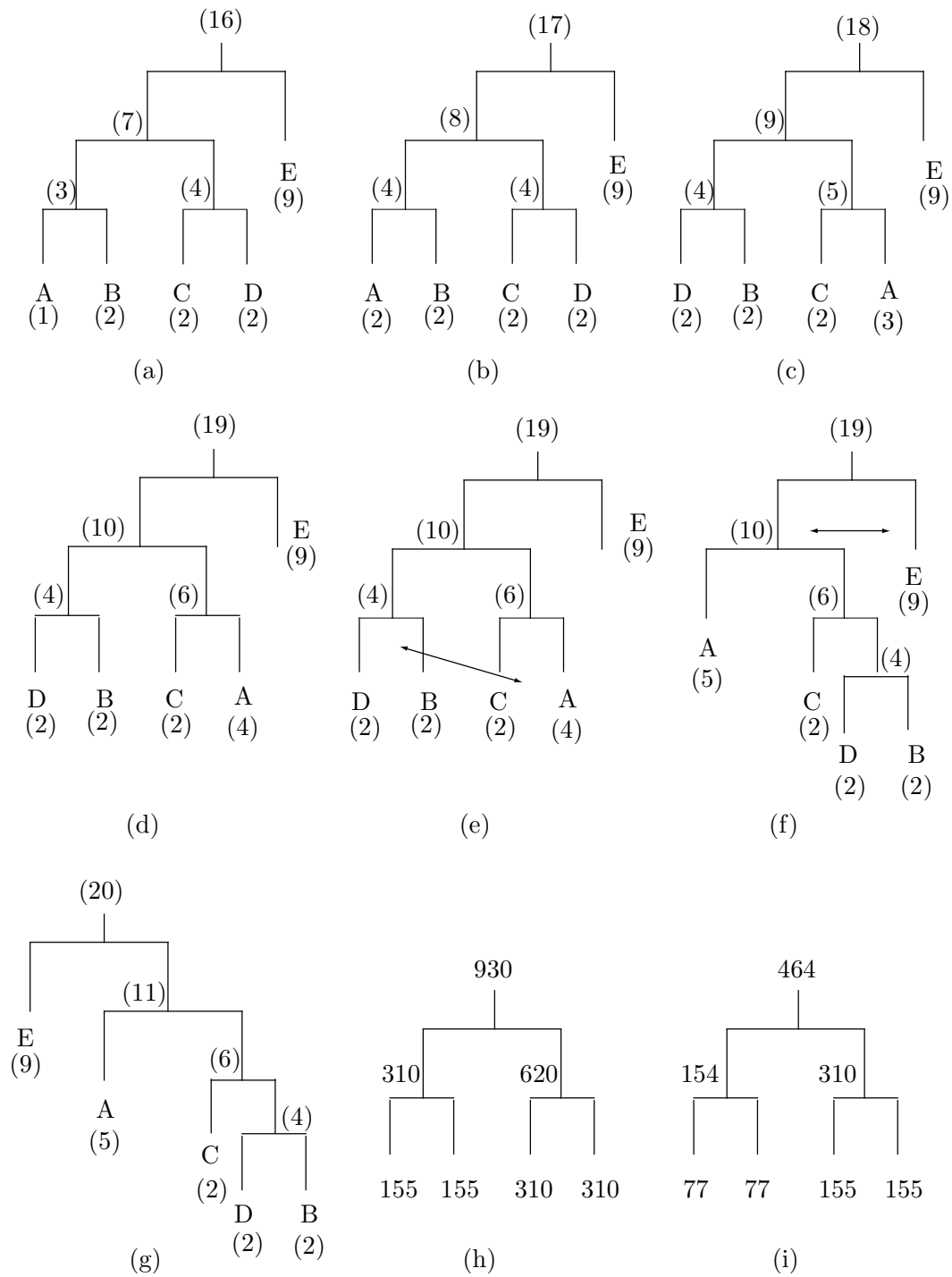


Figure 5.23: Updating the Huffman Tree.

A simple example is shown in Figure 5.23h. After the counts of the leaves are halved, the three interior nodes are updated as shown in Figure 5.23i. The latter tree, however, is no longer a Huffman tree, since the counts are no longer in sorted order. The solution is to rebuild the tree each time the counts are rescaled, which does not happen very often. A Huffman data compression program intended for general use should therefore have large count fields that would not overflow very often. A 4-byte count field overflows at  $2^{32} - 1 \approx 4.3 \times 10^9$ .

It should be noted that after rescaling the counts, the new symbols being read and compressed have more effect on the counts than the old symbols (those counted before the rescaling). This turns out to be fortuitous since it is known from experience that the probability of appearance of a symbol depends more on the symbols immediately preceding it than on symbols that appeared in the distant past.

### 5.3.4 Code Overflow

An even more serious problem is code overflow. This may happen when many symbols are added to the tree, and it becomes tall. The codes themselves are not stored in the tree, since they change all the time, and the compressor has to figure out the code of a symbol  $X$  each time  $X$  is input. Here are the details of this process:

1. The encoder has to locate symbol  $X$  in the tree. The tree has to be implemented as an array of structures, each a node, and the array is searched linearly.
2. If  $X$  is not found, the escape code is emitted, followed by the uncompressed code of  $X$ .  $X$  is then added to the tree.
3. If  $X$  is found, the compressor moves from node  $X$  back to the root, building the code bit by bit as it goes along. Each time it goes from a left child to a parent, a “1” is appended to the code. Going from a right child to a parent appends a “0” bit to the code (or vice versa, but this should be consistent because it is mirrored by the decoder). Those bits have to be accumulated someplace, since they have to be emitted in the *reverse order* in which they are created. When the tree gets taller, the codes get longer. If they are accumulated in a 16-bit integer, then codes longer than 16 bits would cause a malfunction.

One solution to the code overflow problem is to accumulate the bits of a code in a linked list, where new nodes can be created, limited in number only by the amount of available memory. This is general but slow. Another solution is to accumulate the codes in a large integer variable (perhaps 50 bits wide) and document a maximum code size of 50 bits as one of the limitations of the program.

Fortunately, this problem does not affect the decoding process. The decoder reads the compressed code bit by bit and uses each bit to move one step left or right down the tree until it reaches a leaf node. If the leaf is the escape code, the decoder reads the uncompressed code of the symbol off the compressed stream (and adds the symbol to the tree). Otherwise, the uncompressed code is found in the leaf node.

- ◇ **Exercise 5.15:** Given the 11-symbol string `sir_sid_is`, apply the adaptive Huffman method to it. For each symbol input, show the output, the tree after the symbol has been added to it, the tree after being rearranged (if necessary), and the list of nodes traversed left to right and bottom up.

### 5.3.5 A Variant

This variant of the adaptive Huffman method is simpler but less efficient. The idea is to calculate a set of  $n$  variable-length codes based on equal probabilities, to assign those codes to the  $n$  symbols at random, and to change the assignments “on the fly,” as symbols are being read and compressed. The method is not efficient because the codes are not based on the actual probabilities of the symbols in the input stream. However, it is simpler to implement and also faster than the adaptive method described earlier, because it has to swap rows in a table, rather than update a tree, when updating the frequencies of the symbols.

Name	Count	Code	Name	Count	Code	Name	Count	Code	Name	Count	Code
$a_1$	0	0	$a_2$	1	0	$a_2$	1	0	$a_4$	2	0
$a_2$	0	10	$a_1$	0	10	$a_4$	1	10	$a_2$	1	10
$a_3$	0	110	$a_3$	0	110	$a_3$	0	110	$a_3$	0	110
$a_4$	0	111	$a_4$	0	111	$a_1$	0	111	$a_1$	0	111
(a)			(b)			(c)			(d)		

Figure 5.24: Four Steps in a Huffman Variant.

The main data structure is an  $n \times 3$  table where the three columns store the names of the  $n$  symbols, their frequencies of occurrence so far, and their codes. The table is always kept sorted by the second column. When the frequency counts in the second column change, rows are swapped, but only columns 1 and 2 are moved. The codes in column 3 never change. Figure 5.24 shows an example of four symbols and the behavior of the method when the string  $a_2, a_4, a_4$  is compressed.

Figure 5.24a shows the initial state. After the first symbol  $a_2$  is read, its count is incremented, and since it is now the largest count, rows 1 and 2 are swapped (Figure 5.24b). After the second symbol  $a_4$  is read, its count is incremented and rows 2 and 4 are swapped (Figure 5.24c). Finally, after reading the last symbol  $a_4$ , its count is the largest, so rows 1 and 2 are swapped (Figure 5.24d).

The only point that can cause a problem with this method is overflow of the count fields. If such a field is  $k$  bits wide, its maximum value is  $2^k - 1$ , so it will overflow when incremented for the  $2^k$ th time. This may happen if the size of the input stream is not known in advance, which is very common. Fortunately, we do not really need to know the counts, we just need them in sorted order, which makes it easy to solve this problem.

One solution is to count the input symbols and, after  $2^k - 1$  symbols are input and compressed, to (integer) divide all the count fields by 2 (or shift them one position to the right, if this is easier).

Another, similar, solution is to check each count field every time it is incremented, and if it has reached its maximum value (if it consists of all ones), to integer divide all the count fields by 2, as mentioned earlier. This approach requires fewer divisions but more complex tests.

Naturally, whatever solution is adopted should be used by both the compressor and decompressor.

### 5.3.6 Vitter's Method

An improvement of the original algorithm, due to [Vitter 87], which also includes extensive analysis is based on the following key ideas:

1. A different scheme should be used to number the nodes in the dynamic Huffman tree. It is called *implicit numbering*, and it numbers the nodes from the bottom up and in each level from left to right.
2. The Huffman tree should be updated in such a way that the following will always be satisfied. For each weight  $w$ , all leaves of weight  $w$  precede (in the sense of implicit numbering) all the internal nodes of the same weight. This is an *invariant*.

These ideas result in the following benefits:

1. In the original algorithm, it is possible that a rearrangement of the tree would move a node down one level. In the improved version, this does not happen.
2. Each time the Huffman tree is updated in the original algorithm, some nodes may be moved up. In the improved version, at most one node has to be moved up.
3. The Huffman tree in the improved version minimizes the sum of distances from the root to the leaves and also has the minimum height.

A special data structure, called a *floating tree*, is proposed to make it easy to maintain the required invariant. It can be shown that this version performs much better than the original algorithm. Specifically, if a two-pass Huffman method compresses an input file of  $n$  symbols to  $S$  bits, then the original adaptive Huffman algorithm can compress it to at most  $2S + n$  bits, whereas the improved version can compress it down to  $S + n$  bits—a significant difference! Notice that these results do not depend on the size of the alphabet, only on the size  $n$  of the data being compressed and on its nature (which determines  $S$ ).

I think you're begging the question," said Haydock, "and I can see looming ahead one of those terrible exercises in probability where six men have white hats and six men have black hats and you have to work it out by mathematics how likely it is that the hats will get mixed up and in what proportion. If you start thinking about things like that, you would go round the bend. Let me assure you of that!

—Agatha Christie, *The Mirror Crack'd*

## 5.4 MNP5

Microcom, Inc., a maker of modems, has developed a protocol (called MNP, for Microcom Networking Protocol) for use in its modems. Among other things, the MNP protocol specifies how to unpack bytes into individual bits before they are sent by the modem, how to transmit bits serially in the synchronous and asynchronous modes, and what modulation techniques to use. Each specification is called a *class*, and classes 5 and 7 specify methods for data compression. These methods (especially MNP5) have become very popular and were used by many modems in the 1980s and 1990s.

The MNP5 method is a two-stage process that starts with run-length encoding, followed by adaptive frequency encoding.

The first stage is described on page 32 and is repeated here. When three or more identical consecutive bytes are found in the source stream, the compressor emits three copies of the byte onto its output stream, followed by a repetition count. When the decompressor reads three identical consecutive bytes, it knows that the next byte is a repetition count (which may be zero, indicating just three repetitions). A downside of the method is that a run of three characters in the input stream results in four characters written to the output stream (expansion). A run of four characters results in no compression. Only runs longer than four characters get compressed. Another, slight, problem is that the maximum count is artificially limited to 250 instead of to 255.

The second stage operates on the bytes in the partially compressed stream generated by the first stage. Stage 2 is similar to the method of Section 5.3.5. It starts with a table of  $256 \times 2$  entries, where each entry corresponds to one of the 256 possible 8-bit bytes 00000000 to 11111111. The first column, the frequency counts, is initialized to all zeros. Column 2 is initialized to variable-length codes, called *tokens*, that vary from a short “000|0” to a long “111|11111110”. Column 2 with the tokens is shown in Table 5.25 (which shows column 1 with frequencies of zero). Each token starts with a 3-bit header, followed by some code bits.

The code bits (with three exceptions) are the two 1-bit codes 0 and 1, the four 2-bit codes 0 through 3, the eight 3-bit codes 0 through 7, the sixteen 4-bit codes, the thirty-two 5-bit codes, the sixty-four 6-bit codes, and the one hundred and twenty-eight 7-bit codes. This provides for a total of  $2 + 4 + 8 + 16 + 32 + 64 + 128 = 254$  codes. The three exceptions are the first two codes “000|0” and “000|1”, and the last code, which is “111|11111110” instead of the expected “111|11111111”.

When stage 2 starts, all 256 entries of column 1 are assigned frequency counts of zero. When the next byte  $B$  is read from the input stream (actually, it is read from the output of the first stage), the corresponding token is written to the output stream, and the frequency of entry  $B$  is incremented by 1. Following this, tokens may be swapped to ensure that table entries with large frequencies always have the shortest tokens (see the next section for details). Notice that only the tokens are swapped, not the frequency counts. Thus, the first entry always corresponds to byte “00000000” and contains its frequency count. The token of this byte, however, may change from the original “000|0” to something longer if other bytes achieve higher frequency counts.

Byte	Freq.	Token	Byte	Freq.	Token	Byte	Freq.	Token	Byte	Freq.	Token
0	0	000 0	9	0	011 001	26	0	111 1010	247	0	111 1110111
1	0	000 1	10	0	011 010	27	0	111 1011	248	0	111 1111000
2	0	001 0	11	0	011 011	28	0	111 1100	249	0	111 1111001
3	0	001 1	12	0	011 100	29	0	111 1101	250	0	111 1111010
4	0	010 00	13	0	011 101	30	0	111 1110	251	0	111 1111011
5	0	010 01	14	0	011 110	31	0	111 1111	252	0	111 1111100
6	0	010 10	15	0	011 111	32	0	101 00000	253	0	111 1111101
7	0	010 11	16	0	111 0000	33	0	101 00001	254	0	111 1111110
8	0	011 000	17	0	111 0001	34	0	101 00010	255	0	111 11111110

18 to 25 and 35 to 246 continue in the same pattern.

Table 5.25: The MNP5 Tokens.

The frequency counts are stored in 8-bit fields. Each time a count is incremented, the algorithm checks to see whether it has reached its maximum value. If yes, all the

counts are scaled down by dividing them by 2 (an integer division).

Another, subtle, point has to do with interaction between the two compression stages. Recall that each repetition of three or more characters is replaced, in stage 1, by three repetitions, followed by a byte with the repetition count. When these four bytes arrive at stage 2, they are replaced by tokens, but the fourth one does not cause an increment of a frequency count.

Example: Suppose that the character with ASCII code 52 repeats six times. Stage 1 will generate the four bytes 52, 52, 52, 3, and stage 2 will replace each with a token, will increment the entry for 52 (entry 53 in the table) by 3, but will not increment the entry for 3 (which is entry 4 in the table). (The three tokens for the three bytes of 52 may all be different, since tokens may be swapped after each 52 is read and processed.)

The output of stage 2 consists of tokens of different sizes, from 4 to 11 bits. This output is packed in groups of 8 bits, which get written into the output stream. At the end, a special code consisting of 11 bits of 1 (the flush token) is written, followed by as many 1 bits as necessary, to complete the last group of 8 bits.

The efficiency of MNP5 is a result of both stages. The efficiency of stage 1 depends heavily on the original data. Stage 2 also depends on the original data, but to a smaller extent. Stage 2 tends to identify the most frequent characters in the data and assign them the short codes. A look at Table 5.25 shows that 32 of the 256 characters have tokens that are 7 bits or fewer in length, thereby resulting in compression. The other 224 characters have tokens that are 8 bits or longer. When one of these characters is replaced by a long token, the result is no compression, or even expansion.

The efficiency of MNP5 therefore depends on how many characters dominate the original data. If all characters occur with the same frequency, expansion will result. In the other extreme case, if only four characters appear in the data, each will be assigned a 4-bit token, and the compression factor will be 2.

- ◇ **Exercise 5.16:** Assuming that all 256 characters appear in the original data with the same probability ( $1/256$  each), what will the expansion factor in stage 2 be?

### 5.4.1 Updating the Table

The process of updating the table of MNP5 codes by swapping rows can be done in two ways:

1. Sorting the entire table every time a frequency is incremented. This is simple in concept but too slow in practice, because the table is 256 entries long.
2. Using pointers in the table, and swapping pointers such that items with large frequencies will point to short codes. This approach is illustrated in Figure 5.26. The figure shows the code table organized in four columns labeled F, P, Q, and C. Columns F and C contain the frequencies and codes; columns P and Q contain pointers that always point to each other, so if P[i] contains index j (i.e., points to Q[j]), then Q[j] points to P[i]. The following paragraphs correspond to the nine different parts of the figure.
 

(a) The first data item a is read and F[a] is incremented from 0 to 1. The algorithm starts with pointer P[a] that contains, say, j. The algorithm examines pointer Q[j-1], which initially points to entry F[b], the one right above F[a]. Since F[a] > F[b], entry a has to be assigned a short code, and this is done by swapping pointers P[a] and P[b] (and also the corresponding Q pointers).



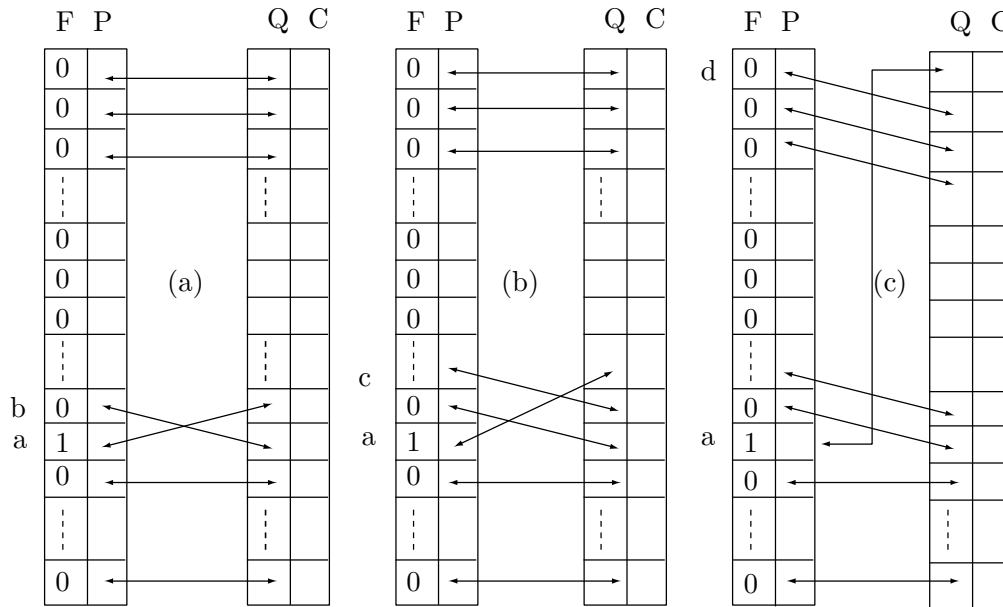


Figure 5.26: Swapping Pointers in the MNP5 Code Table (Part I).

(b) The same process is repeated. The algorithm again starts with pointer  $P[a]$ , which now points higher, to entry  $b$ . Assuming that  $P[a]$  contains the index  $k$ , the algorithm examines pointer  $Q[k-1]$ , which points to entry  $c$ . Since  $F[a] > F[c]$ , entry  $a$  should be assigned a code shorter than that of  $c$ . This again is done by swapping pointers, this time  $P[a]$  and  $P[c]$ .

(c) This process is repeated, and since  $F[a]$  is greater than all the frequencies above it, pointers are swapped until  $P[a]$  points to the top entry,  $d$ . At this point entry  $a$  has been assigned the shortest code.

(d) We now assume that the next data item has been input, and  $F[m]$  incremented to 1. Pointers  $P[m]$  and the one above it are swapped as in (a) above.

(e) After a few more swaps,  $P[m]$  is now pointing to entry  $n$  (the one just below  $a$ ). The next step performs  $j := P[m]$ ;  $b := Q[j-1]$ , and the algorithm compares  $F[m]$  to  $F[b]$ . Since  $F[m] > F[b]$ , pointers are swapped as shown in Figure 5.26f.

(f) After the swap,  $P[m]$  is pointing to entry  $a$  and  $P[b]$  is pointing to entry  $n$ .

(g) After a few more swaps, pointer  $P[m]$  points to the second entry  $p$ . This is how entry  $m$  is assigned the second-shortest code. Pointer  $P[m]$  is not swapped with  $P[a]$ , since they have the same frequencies.

(h) We now assume that the third data item has been input and  $F[x]$  incremented. Pointers  $P[x]$  and  $P[y]$  are swapped.

(i) After some more swaps, pointer  $P[x]$  points to the third table entry  $z$ . This is how entry  $x$  is assigned the third shortest code.

Assuming that  $F[x]$  is incremented next, the reader is invited to try to figure out how  $P[x]$  is swapped, first with  $P[m]$  and then with  $P[a]$ , so that entry  $x$  is assigned the shortest code.

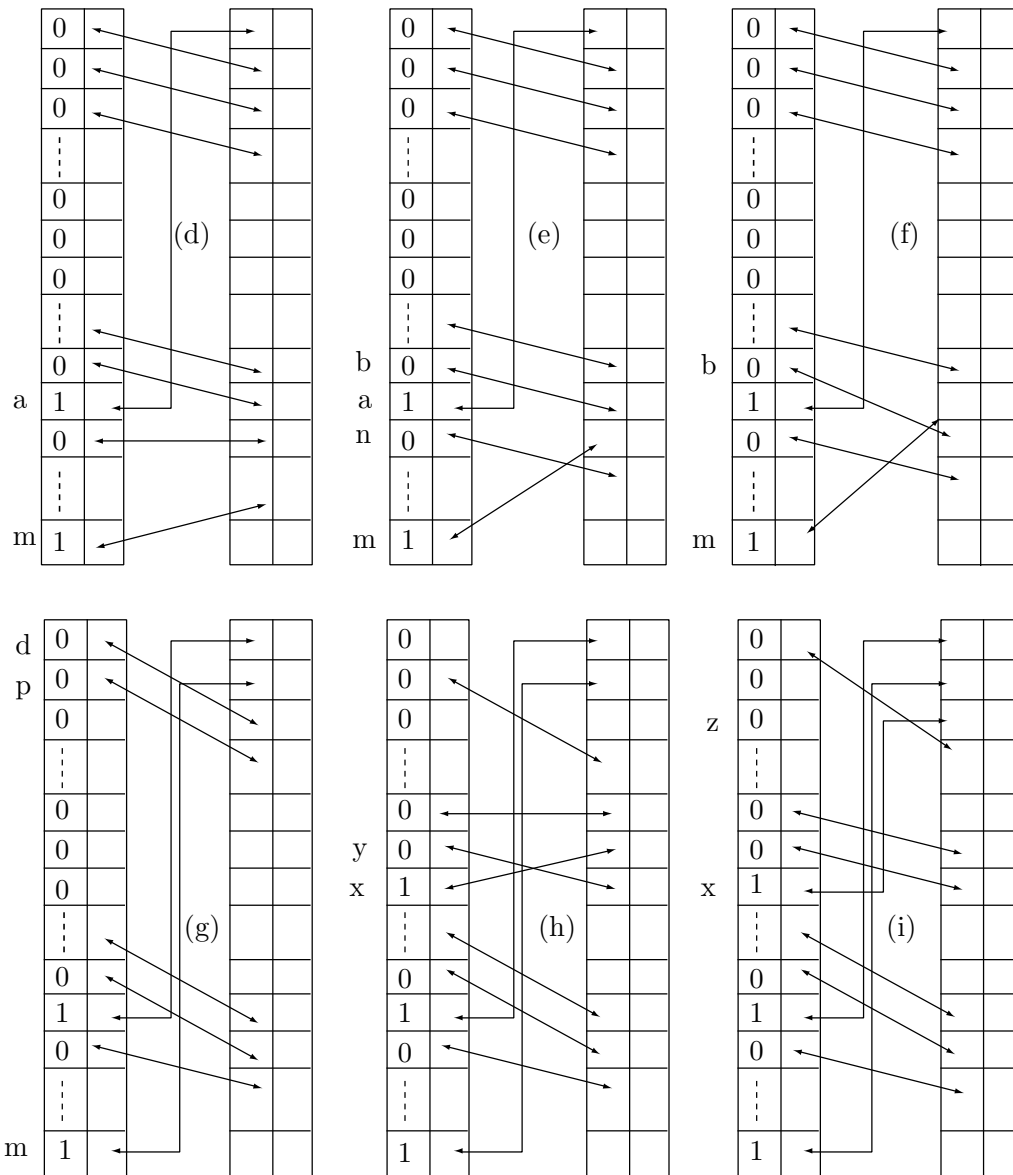


Figure 5.26 (Continued).

```

F[i]:=F[i]+1;
repeat forever
  j:=P[i];
  if j=1 then exit;
  j:=Q[j-1];
  if F[i]<=F[j] then exit
  else
    tmp:=P[i]; P[i]:=P[j]; P[j]:=tmp;
    tmp:=Q[P[i]]; Q[P[i]]:=Q[P[j]]; Q[P[j]]:=tmp
  endif;
end repeat

```

Figure 5.27: Swapping Pointers in MNP5.

The pseudo-code of Figure 5.27 summarizes the pointer swapping process.

Are no probabilities to be accepted, merely because they are not certainties?  
 —Jane Austen, *Sense and Sensibility*

## 5.5 MNP7

More complex and sophisticated than MNP5, MNP7 combines run-length encoding with a two-dimensional variant of adaptive Huffman coding. Stage 1 identifies runs and emits three copies of the run character, followed by a 4-bit count of the remaining characters in the run. A count of zero implies a run of length 3, and a count of 15 (the largest possible in a 4-bit nibble), a run of length 18. Stage 2 starts by assigning to each character a complete table with many variable-length codes. When a character  $C$  is read, one of the codes in its table is selected and output, depending on the *character preceding*  $C$  in the input stream. If this character is, say,  $P$ , then the frequency count of the pair (digram)  $PC$  is incremented by 1, and table rows may be swapped, using the same algorithm as for MNP5, to move the pair to a position in the table that has a shorter code.

MNP7 is therefore based on a first-order Markov model, where each item is processed depending on the item and one of its predecessors. In a  $k$ -order Markov model, an item is processed depending on itself and  $k$  of its predecessors (not necessarily the  $k$  immediate ones).

Here are the details. Each of the 256 8-bit bytes gets a table of codes assigned, of size  $256 \times 2$ , where each row corresponds to one of the 256 possible bytes. Column 1 of the table is initialized to the integers 0 through 255, and column 2 (the frequency counts) is initialized to all zeros. The result is 256 tables, each a double column of 256 rows (Table 5.28a). Variable-length codes are assigned to the rows, such that the first code is 1-bit long, and the others get longer towards the bottom of the table. The codes are stored in an additional code table that never changes.

When a character  $C$  is input (the current character to be compressed), its value is used as a pointer, to select one of the 256 tables. The first column of the table is

		Current character											
		0	1	2	...	254	255						
Preced. Char.		0 0	0 0	0 0	...	0 0	0 0						
		1 0	1 0	1 0	...	1 0	1 0						
		2 0	2 0	2 0	...	2 0	2 0	...	a	b	c	d	e...
		3 0	3 0	3 0	...	3 0	3 0		t	l	h	o	d
		⋮	⋮	⋮		⋮	⋮		h	e	o	a	r
		254 0	254 0	254 0	...	254 0	254 0		c	u	r	e	s
		255 0	255 0	255 0	...	255 0	255 0		⋮	⋮	⋮	⋮	⋮
	(a)												
	(b)												

Table 5.28: The MNP7 Code Tables.

searched, to find the row with the 8-bit value of the preceding character  $P$ . Once the row is found, the code from the same row in the code table is emitted and the count in the second column is incremented by 1. Rows in the table may be swapped if the new count of the digram  $PC$  is large enough.

After enough characters have been input and rows swapped, the tables start reflecting the true digram frequencies of the data. Table 5.28b shows a possible state assuming that the digrams **ta**, **ha**, **ca**, **lb**, **eb**, **ub**, **hc**, etc., are common. Since the top digram is encoded into 1 bit, MNP7 can be very efficient. If the original data consists of text in a natural language, where certain digrams are very common, MNP7 normally produces a high compression ratio.

His only other visitor was a woman: the woman who had attended his reading. At the time she had seemed to him to be the only person present who had paid the slightest attention to his words. With kittenish timidity she approached his table. Richard bade her welcome, and meant it, and went on meaning it as she extracted from her shoulder pouch a copy of a novel written not by Richard Tull but by Fyodor Dostoevsky. *The Idiot*. Standing beside him, leaning over him, her face awfully warm and near, she began to leaf through its pages, explaining. The book too was stained, not by gout of blood but by the vying colors of two highlighting pens, one blue, one pink. And not just two pages but the whole six hundred. Every time the letters *h* and *e* appeared together, as in *the*, *then*, *there*, as in *forehead*, *Pashlishtchev*, *sheepskin*, they were shaded in blue. Every time the letters *s*, *h*, and *e* appeared together, as in *she*, *sheer*, *ashen*, *sheepskin*, etc., they were shaded in pink. And since every *she* contained a *he*, the predominance was unarguably and unsurprisingly masculine. Which was exactly her point. “You see?” she said with her hot breath, breath redolent of metallic medications, of batteries and printing-plates. “You see?”... The organizers knew all about this woman—this unfortunate recurrence, this indefatigable drag—and kept coming over to try and coax her away.

—Martin Amis, *The Information*

## 5.6 Reliability

The chief downside of variable-length codes is their vulnerability to errors. The prefix property is used to decode those codes, so an error in a single bit can cause the decompressor to lose synchronization and be unable to decode the rest of the compressed stream. In the worst case, the decompressor may even read, decode, and interpret the rest of the compressed data incorrectly, without realizing that a problem has occurred.

Example: Using the code of Figure 5.5 the string **CARE** is coded into 10100 0011 0110 000 (without the spaces). Assuming the error 10000 0011 0110 000, the decompressor will not notice any problem but will decode the string as **HARE**.

- ◇ **Exercise 5.17:** What will happen in the case 11111 0011 0110 000 ... (the string **WARE** ... with one bad bit)?

Users of Huffman codes have noticed long ago that these codes recover quickly from an error. However, if Huffman codes are used to code run lengths, then this property does not help, since all runs would be shifted after an error.

A simple way of adding reliability to variable-length codes is to break a long compressed stream, as it is being transmitted, into groups of 7 bits and add a parity bit to each group. This way, the decompressor will at least be able to detect a problem and output an error message or ask for a retransmission. It is, of course, possible to add more than one parity bit to a group of data bits, thereby making it more reliable. However, reliability is, in a sense, the opposite of compression. Compression is done by decreasing redundancy, while reliability is achieved by increasing it. The more reliable a piece of data is, the less compressed it is, so care should be taken when the two operations are used together.

---

### Some Important Standards Groups and Organizations

ANSI (American National Standards Institute) is the private sector voluntary standardization system for the United States. Its members are professional societies, consumer groups, trade associations, and some government regulatory agencies (it is a federation). It collects and distributes standards developed by its members. Its mission is the enhancement of global competitiveness of U.S. business and the American quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems and promoting their integrity.

ANSI was founded in 1918 by five engineering societies and three government agencies, and it remains a private, nonprofit membership organization whose nearly 1,400 members are private and public sector organizations.

ANSI is located at 11 West 42nd Street, New York, NY 10036, USA. See also <http://web.ansi.org/>.

ISO (International Organization for Standardization, Organisation Internationale de Normalisation) is an agency of the United Nations whose members are standards organizations of some 100 member countries (one organization from each country). It develops a wide range of standards for industries in all fields.

Established in 1947, its mission is “to promote the development of standardization in the world with a view to facilitating the international exchange of goods and

services, and to developing cooperation in the spheres of intellectual, scientific, technological and economic activity.” This is the forum where it is agreed, for example, how thick your bank card should be, so every country in the world follows a compatible standard. The ISO is located at 1, rue de Varembe, CH-1211 Geneva 20, Switzerland, <http://www.iso.ch/>.

ITU (International Telecommunication Union) is another United Nations agency developing standards for telecommunications. Its members are mostly companies that make telecommunications equipment, groups interested in standards, and government agencies involved with telecommunications. The ITU is the successor of an organization founded by some 20 European nations in Paris in 1865.

IEC (International Electrotechnical Commission) is a nongovernmental international organization that prepares and publishes international standards for all electrical, electronic, and related technologies. The IEC was founded, in 1906, as a result of a resolution passed at the International Electrical Congress held in St. Louis (USA) in 1904. Its membership consists of more than 50 participating countries, including all the world’s major trading nations and a growing number of industrializing countries.

The IEC’s mission is to promote, through its members, international cooperation on all questions of electrotechnical standardization and related matters, such as the assessment of conformity to standards, in the fields of electricity, electronics and related technologies.

The IEC is located at 3, rue de Varembe, P.O. Box 131, CH-1211, Geneva 20, Switzerland, <http://www.iec.ch/>.

QIC (<http://www.qic.org/>) was an international trade association, incorporated in 1987, to encourage and promote the widespread use of quarter-inch tape cartridge technology. One hundred companies around the world were Members or Associates of QIC during its active years. The group became inactive in 1998 after more than 15 million QIC-compatible drives had been installed.

Most of the 15 million QIC tape drives in use worldwide were installed in business environments. QIC tape automation solutions enabled capacities well into the terabyte range, providing the hardware data compression and read-write features essential to network backup.

---

## 5.7 Facsimile Compression

Data compression is especially important when images are transmitted over a communications line because the user is often waiting at the receiver, eager to see something quickly. Documents transferred between fax machines are sent as bitmaps, so a standard data compression method was needed when those machines became popular. Several methods were developed and proposed by the ITU-T.

The ITU-T is one of four permanent parts of the International Telecommunications Union (ITU), based in Geneva, Switzerland (<http://www.itu.ch/>). It issues recommendations for standards applying to modems, packet switched interfaces, V.24 connectors, and similar devices. Although it has no power of enforcement, the standards it recommends are generally accepted and adopted by industry. Until March 1993, the ITU-T

CCITT: Can't Conceive Intelligent Thoughts Today

was known as the Consultative Committee for International Telephone and Telegraph (Comité Consultatif International Télégraphique et Téléphonique, or CCITT).

The first data compression standards developed by the ITU-T were T2 (also known as Group 1) and T3 (Group 2). These are now obsolete and have been replaced by T4 (Group 3) and T6 (Group 4). Group 3 is currently used by all fax machines designed to operate with the Public Switched Telephone Network (PSTN). These are the machines we have at home, and at the time of writing, they operate at maximum speeds of 9,600 baud. Group 4 is used by fax machines designed to operate on a digital network, such as ISDN. They have typical speeds of 64K baud. Both methods can produce compression factors of 10 or better, reducing the transmission time of a typical page to about a minute with the former, and a few seconds with the latter.

Some references for facsimile compression are [Anderson et al. 87], [Hunter and Robinson 80], [Marking 90], and [McConnell 92].

### 5.7.1 One-Dimensional Coding

A fax machine scans a document line by line, converting each line to small black and white dots called *pels* (from Picture ELEMENT). The horizontal resolution is always 8.05 pels per millimeter (about 205 pels per inch). An 8.5-inch-wide scan line is therefore converted to 1728 pels. The T4 standard, though, recommends to scan only about 8.2 inches, thereby producing 1664 pels per scan line (these numbers, as well as those in the next paragraph, are all to within  $\pm 1\%$  accuracy).

The vertical resolution is either 3.85 scan lines per millimeter (standard mode) or 7.7 lines/mm (fine mode). Many fax machines have also a very-fine mode, where they scan 15.4 lines/mm. Table 5.29 assumes a 10-inch-high page (254 mm), and shows the total number of pels per page, and typical transmission times for the three modes without compression. The times are long, illustrating how important data compression is in fax transmissions.

Scan lines	Pels per line	Pels per page	Time (sec.)	Time (min.)
978	1664	1.670M	170	2.82
1956	1664	3.255M	339	5.65
3912	1664	6.510M	678	11.3

Ten inches equal 254 mm. The number of pels is in the millions, and the transmission times, at 9600 baud without compression, are between 3 and 11 minutes, depending on the mode. However, if the page is shorter than 10 inches, or if most of it is white, the compression factor can be 10 or better, resulting in transmission times of between 17 and 68 seconds.

Table 5.29: Fax Transmission Times.

To derive the Group 3 code, the ITU-T counted all the run lengths of white and black pels in a set of eight “training” documents that they felt represent typical text and images sent by fax, and used the Huffman algorithm to assign a variable-length code to each run length. (The eight documents are described in Table 5.30. They are not shown because they are copyrighted by the ITU-T.) The most common run lengths were found to be 2, 3, and 4 black pixels, so they were assigned the shortest codes (Table 5.31). Next come run lengths of 2–7 white pixels, which were assigned slightly longer codes. Most run lengths were rare and were assigned long, 12-bit codes. Thus, Group 3 uses a combination of RLE and Huffman coding.

Image	Description
1	Typed business letter (English)
2	Circuit diagram (hand drawn)
3	Printed and typed invoice (French)
4	Densely typed report (French)
5	Printed technical article including figures and equations (French)
6	Graph with printed captions (French)
7	Dense document (Kanji)
8	Handwritten memo with very large white-on-black letters (English)

Table 5.30: The Eight CCITT Training Documents.

- ◇ **Exercise 5.18:** A run length of 1664 white pels was assigned the short code 011000. Why is this length so common?

Since run lengths can be long, the Huffman algorithm was modified. Codes were assigned to run lengths of 1 to 63 pels (they are the termination codes in Table 5.31a) and to run lengths that are multiples of 64 pels (the make-up codes in Table 5.31b). Group 3 is therefore a *modified Huffman code* (also called MH). The code of a run length is either a single termination code (if the run length is short) or one or more make-up codes, followed by one termination code (if it is long). Here are some examples:


1. A run length of 12 white pels is coded as 001000.
2. A run length of 76 white pels ( $= 64 + 12$ ) is coded as 11011|001000.
3. A run length of 140 white pels ( $= 128 + 12$ ) is coded as 10010|001000.
4. A run length of 64 black pels ( $= 64 + 0$ ) is coded as 0000001111|0000110111.
5. A run length of 2561 black pels ( $2560 + 1$ ) is coded as 000000011111|010.


- ◇ **Exercise 5.19:** There are no runs of length zero. Why then were codes assigned to runs of zero black and zero white pels?
- ◇ **Exercise 5.20:** An 8.5-inch-wide scan line results in 1728 pels, so how can there be a run of 2561 consecutive pels?

Each scan line is coded separately, and its code is terminated by the special 12-bit EOL code 000000000001. Each line also gets one white pel appended to it on the left when it is scanned. This is done to remove any ambiguity when the line is decoded on



the receiving end. After reading the EOL for the previous line, the receiver assumes that the new line starts with a run of white pels, and it ignores the first of them. Examples:

1. The 14-pel line  is coded as the run lengths 1w 3b 2w 2b 7w EOL, which become 000111|10|0111|11|1111|000000000001. The decoder ignores the single white pel at the start.

2. The line  is coded as the run lengths 3w 5b 5w 2b EOL, which becomes the binary string 1000|0011|1100|11|000000000001.

- ◇ **Exercise 5.21:** The group 3 code for a run length of five black pels (0011) is also the prefix of the codes for run lengths of 61, 62, and 63 white pels. Explain this.

The Group 3 code has no error correction, but many errors can be detected. Because of the nature of the Huffman code, even one bad bit in the transmission can cause the receiver to get out of synchronization, and to produce a string of wrong pels. This is why each scan line is encoded separately. If the receiver detects an error, it skips bits, looking for an EOL. This way, one error can cause at most one scan line to be received incorrectly. If the receiver does not see an EOL after a certain number of lines, it assumes a high error rate, and it aborts the process, notifying the transmitter. Since the codes are between 2 and 12 bits long, the receiver detects an error if it cannot decode a valid code after reading 12 bits.

Each page of the coded document is preceded by one EOL and is followed by six EOL codes. Because each line is coded separately, this method is a *one-dimensional coding* scheme. The compression ratio depends on the image. Images with large contiguous black or white areas (text or black and white images) can be highly compressed. Images with many short runs can sometimes produce negative compression. This is especially true in the case of images with shades of gray (such as scanned photographs). Such shades are produced by halftoning, which covers areas with many alternating black and white pels (runs of length one).

- ◇ **Exercise 5.22:** What is the compression ratio for runs of length one (i.e., strictly alternating pels)?

The T4 standard also allows for fill bits to be inserted between the data bits and the EOL. This is done in cases where a pause is necessary, or where the total number of bits transmitted for a scan line must be a multiple of 8. The fill bits are zeros.

Example: The binary string 000111|10|0111|11|1111|000000000001 becomes

000111|10|0111|11|1111|00|0000000001

after two zeros are added as fill bits, bringing the total length of the string to 32 bits ( $= 8 \times 4$ ). The decoder sees the two zeros of the fill, followed by the 11 zeros of the EOL, followed by the single 1, so it knows that it has encountered a fill followed by an EOL.

See <http://www.doclib.org/rfc/rfc804.html> for a description of group 3.

At the time of writing, the T.4 and T.6 recommendations can also be found at URL <ftp://sunsite.doc.ic.ac.uk/> as files 7\_3\_01.ps.gz and 7\_3\_02.ps.gz (the precise subdirectory seems to change every few years and it is recommended to locate it with a search engine).

(a)	Run length	White code- word	Black code- word	Run length	White code- word	Black code- word
	0	00110101	0000110111	32	00011011	000001101010
	1	000111	010	33	00010010	000001101011
	2	0111	11	34	00010011	000011010010
	3	1000	10	35	00010100	000011010011
	4	1011	011	36	00010101	000011010100
	5	1100	0011	37	00010110	000011010101
	6	1110	0010	38	00010111	000011010110
	7	1111	00011	39	00101000	000011010111
	8	10011	000101	40	00101001	000001101100
	9	10100	000100	41	00101010	000001101101
	10	00111	0000100	42	00101011	000011011010
	11	01000	0000101	43	00101100	000011011011
	12	001000	0000111	44	00101101	000001010100
	13	000011	00000100	45	00000100	000001010101
	14	110100	00000111	46	00000101	000001010110
	15	110101	000011000	47	00001010	000001010111
	16	101010	0000010111	48	00001011	000001100100
	17	101011	0000011000	49	01010010	000001100101
	18	0100111	0000001000	50	01010011	000001010010
	19	0001100	00001100111	51	01010100	000001010011
	20	0001000	00001101000	52	01010101	000000100100
	21	0010111	00001101100	53	00100100	000000110111
	22	0000011	00000110111	54	00100101	000000111000
	23	0000100	00000101000	55	01011000	000000100111
	24	0101000	00000010111	56	01011001	000000101000
	25	0101011	00000011000	57	01011010	000001011000
	26	0010011	000011001010	58	01011011	000001011001
	27	0100100	000011001011	59	01001010	000000101011
	28	0011000	000011001100	60	01001011	000000101100
	29	00000010	000011001101	61	00110010	000001011010
	30	00000011	000001101000	62	00110011	000001100110
	31	00011010	000001101001	63	00110100	000001100111
(b)	Run length	White code- word	Black code- word	Run length	White code- word	Black code- word
	64	11011	0000001111	1344	011011010	0000001010011
	128	10010	000011001000	1408	011011011	0000001010100
	192	010111	000011001001	1472	010011000	0000001010101
	256	0110111	000001011011	1536	010011001	0000001011010
	320	00110110	000000110011	1600	010011010	0000001011011
	384	00110111	000000110100	1664	011000	0000001100100
	448	01100100	000000110101	1728	010011011	0000001100101
	512	01100101	0000001101100	1792	00000001000	same as
	576	01101000	0000001101101	1856	00000001100	white
	640	01100111	0000001001010	1920	00000001101	from this
	704	011001100	0000001001011	1984	000000010010	point
	768	011001101	0000001001100	2048	000000010011	
	832	011010010	0000001001101	2112	000000010100	
	896	011010011	0000001110010	2176	000000010101	
	960	011010100	0000001110011	2240	000000010110	
	1024	011010101	0000001110100	2304	000000010111	
	1088	011010110	0000001110101	2368	000000011100	
	1152	011010111	0000001110110	2432	000000011101	
	1216	011011000	0000001110111	2496	000000011110	
	1280	011011001	0000001010010	2560	000000011111	

Table 5.31: Group 3 and 4 Fax Codes: (a) Termination Codes, (b) Make-Up Codes.

### 5.7.2 Two-Dimensional Coding

Two-dimensional coding was developed because one-dimensional coding does not produce good results for images with gray areas. Two-dimensional coding is optional on fax machines that use Group 3 but is the only method used by machines intended to work in a digital network. When a fax machine using Group 3 supports two-dimensional coding as an option, each EOL is followed by one extra bit, to indicate the compression method used for the next scan line. That bit is 1 if the next line is encoded with one-dimensional coding, and 0 if it is encoded with two-dimensional coding.

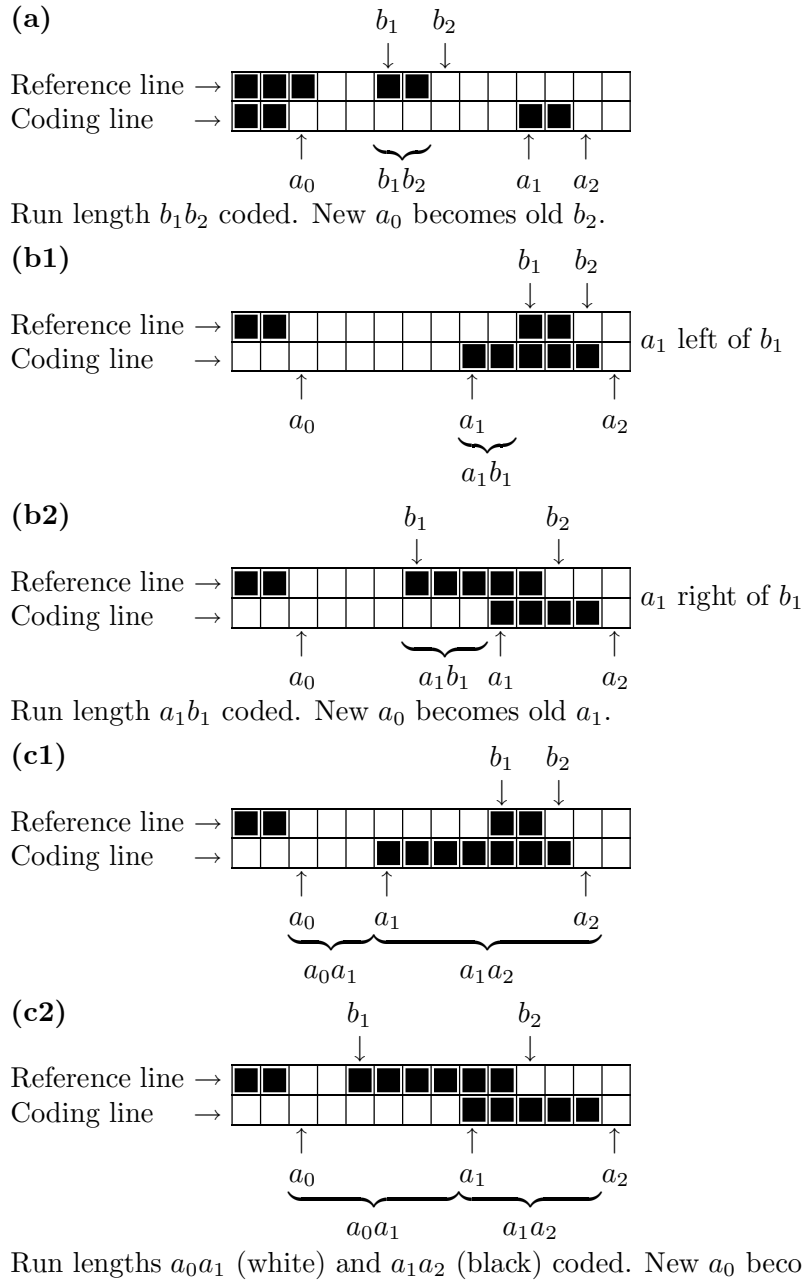
The two-dimensional coding method is also called MMR, for *modified modified READ*, where READ stands for *relative element address designate*. The term “modified modified” is used because this is a modification of one-dimensional coding, which itself is a modification of the original Huffman method. The two-dimensional coding method works by comparing the current scan line (called the *coding line*) to its predecessor (which is called the *reference line*) and recording the differences between them, the assumption being that two consecutive lines in a document will normally differ by just a few pels. The method assumes that there is an all-white line above the page, which is used as the reference line for the first scan line of the page. After coding the first line, it becomes the reference line, and the second scan line is coded. As in one-dimensional coding, each line is assumed to start with a white pel, which is ignored by the receiver.

The two-dimensional coding method is less reliable than one-dimensional coding, since an error in decoding a line will cause errors in decoding all its successors and will propagate through the entire document. This is why the T.4 (Group 3) standard includes a requirement that after a line is encoded with the one-dimensional method, at most  $K - 1$  lines will be encoded with the two-dimensional coding method. For standard resolution  $K = 2$ , and for fine resolution  $K = 4$ . The T.6 standard (Group 4) does not have this requirement, and uses two-dimensional coding exclusively.

Scanning the coding line and comparing it to the reference line results in three cases, or modes. The mode is identified by comparing the next run length on the reference line  $[(b_1b_2)]$  in Figure 5.33] with the current run length  $(a_0a_1)$  and the next one  $(a_1a_2)$  on the coding line. Each of these three runs can be black or white. The three modes are as follows (see also the flow chart of Figure 5.34):

1. **Pass mode.** This is the case where  $(b_1b_2)$  is to the left of  $(a_1a_2)$  and  $b_2$  is to the left of  $a_1$  (Figure 5.33a). This mode does not include the case where  $b_2$  is above  $a_1$ . When this mode is identified, the length of run  $(b_1b_2)$  is coded using the codes of Table 5.32 and is transmitted. Pointer  $a_0$  is moved below  $b_2$ , and the four values  $b_1$ ,  $b_2$ ,  $a_1$ , and  $a_2$  are updated.
2. **Vertical mode.**  $(b_1b_2)$  overlaps  $(a_1a_2)$  by not more than three pels (Figure 5.33b1, b2). Assuming that consecutive lines do not differ by much, this is the most common case. When this mode is identified, one of seven codes is produced (Table 5.32) and is transmitted. Pointers are updated as in case 1 above. The performance of the two-dimensional coding method depends on this case being common.
3. **Horizontal mode.**  $(b_1b_2)$  overlaps  $(a_1a_2)$  by more than three pels (Figure 5.33c1, c2). When this mode is identified, the lengths of runs  $(a_0a_1)$  and  $(a_1a_2)$  are coded using the codes of Table 5.32 and are transmitted. Pointers are updated as in cases 1 and 2 above.





Notes:

1.  $a_0$  is the first pel of a new codeword and can be black or white.
2.  $a_1$  is the first pel to the right of  $a_0$  with a different color.
3.  $a_2$  is the first pel to the right of  $a_1$  with a different color.
4.  $b_1$  is the first pel on the reference line to the right of  $a_0$  with a different color.
5.  $b_2$  is the first pel on the reference line to the right of  $b_1$  with a different color.

Figure 5.33: Five Run-Length Configurations: (a) Pass Mode, (b) Vertical Mode, (c) Horizontal Mode.

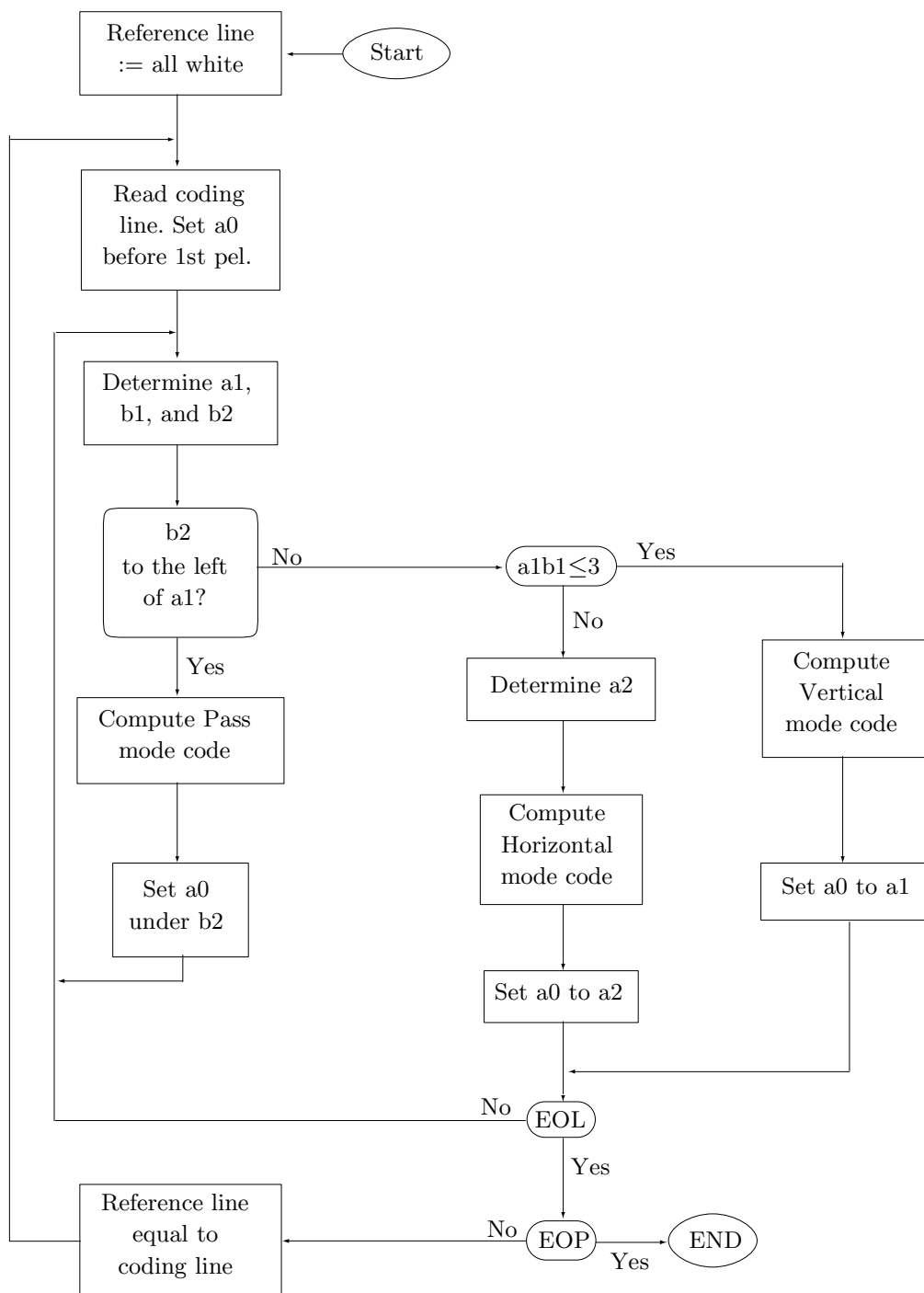


Figure 5.34: MMR Flow Chart.

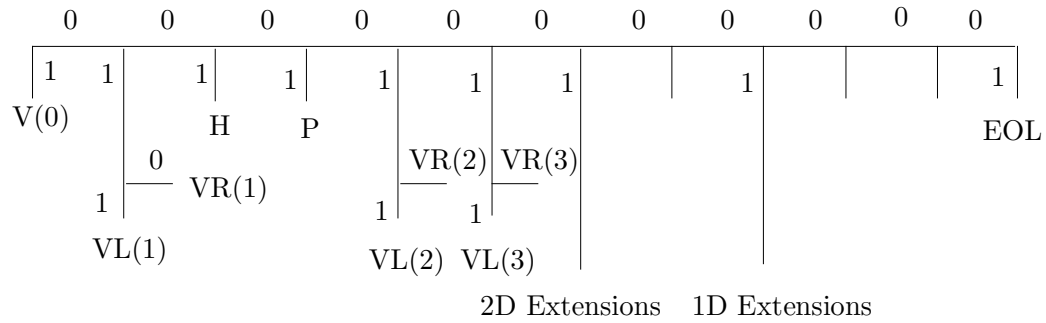


Figure 5.35: Tree of Group 3 Codes.

Mode	Elements to Be Coded		Notation	Codeword
Pass	$b_1, b_2$		P	0001
Horizontal	$a_0 a_1, a_1 a_2$		H	$001 + M(a_0 a_1) + M(a_1 a_2)$
Vertical	$a_1$ just under $b_1$	$a_1 b_1 = 0$	V(0)	1
	$a_1$ to the right of $b_1$	$a_1 b_1 = 1$	VR(1)	011
		$a_1 b_1 = 2$	VR(2)	000011
		$a_1 b_1 = 3$	VR(3)	0000011
	$a_1$ to the left of $b_1$	$a_1 b_1 = 1$	VL(1)	010
		$a_1 b_1 = 2$	VL(2)	000010
		$a_1 b_1 = 3$	VL(3)	0000010
	2D Extensions 1D Extensions			0000001xxx 000000001xxx
EOL			000000000001	
1D Coding of Next Line 2D Coding of Next Line			EOL+‘1’ EOL+‘0’	

Table 5.36: Group 4 Codes.

## 5.8 PK Font Compression

Obviously, *these words* are hard to read because the individual characters feature different styles and sizes. In a beautifully-typeset document, all the letters of a word and all the words of the document (with perhaps a few exceptions) should have the same size and style; they should belong to the same *font*.

Traditionally, the term “font” refers to a set of characters of type that are of the same size and style, such as Times Roman 12 point. A typeface is a set of fonts of different sizes but in the same style, like Times Roman. A typeface family is a set of typefaces in the same style, such as Times.

The size of a font is normally measured in points (more accurately, printer’s points, where 72.27 points equal one inch). The style of a font describes its appearance. Traditional styles are roman, **boldface**, *italic*, *slanted*, **typewriter**, and **sans serif**.

Old operating systems did not support fonts. Even the DOS operating system, which was widely used on IBM-PC-compatible personal computers from 1980 to 1995, did not support fonts and was based on ASCII codes instead.

It was not until the mid 1980s that digital fonts became a part of many operating systems. In 1984, Adobe launched the PostScript language, which supported two types of digital fonts. In the same year, Apple released the first models of the Macintosh computer, whose operating system used fonts (Figure 5.37 illustrates some of the early fonts included in the Macintosh OS 6). In 1985, Apple announced the LaserWriter, one of the first laser printers intended for the mass market. These developments paved the way for future operating systems to support digital fonts, thus enabling users to create, print, and publish beautiful, professionally-looking documents.



Figure 5.37: Old Macintosh OS 6 Fonts.

A digital font is a set of symbols or characters that can be stored in the computer’s memory or on an I/O device such as a disk. The symbols of the font are either displayed or printed. Each symbol consists of a glyph (the actual shape of the symbol) and bookkeeping information, such as the height, depth, and width of the symbol.

Modern fonts are referred to as outline fonts. A symbol in such a font is fully defined by a small set of control points that are connected by curves to form the outline of the symbol. The symbol  $\pi$  in Figure 5.38 is an example. It is easy to see how this symbol is fully specified by about 30 control points that are connected with a few smooth curves. An outline font is easy to store in a computer file simply by storing the coordinates of the control points. It is also easy to change the size of the font by scaling the coordinates of the points.

In contrast, early digital fonts were of the bitmap variety. The font designer would draw the glyph of each symbol on a grid of small squares (pixels) and then select the



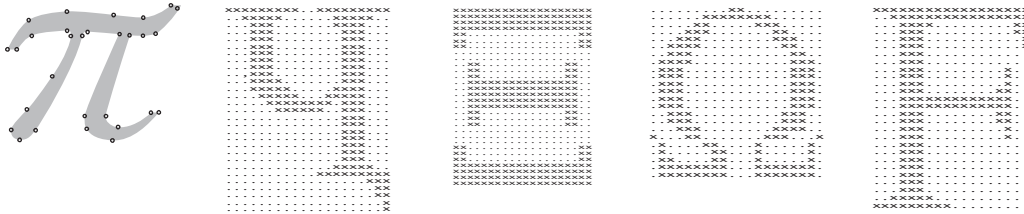


Figure 5.38: Outline and Bitmap Fonts.

best pixels for the symbol. Figure 5.38 shows four examples of bitmap symbols where the white pixels are represented by dots and the black pixels are shown as an  $\times$ . In the computer's memory, a bitmap symbol is stored as a map of bits, zeros for white pixels and 1's for black pixels. When such a font is written in raw format on a file, the file tends to be large and should be compressed. Also, a high-quality bitmap font has to be designed from scratch for each font size. Simply scaling a bitmap results in badly-looking characters.

The  $\text{\TeX}$  project, started by Donald Knuth in 1975, is an early example of the use of bitmap fonts. The goal was to develop software for typesetting beautiful documents, especially documents with a lot of mathematics. Such software requires a set of fonts, even if the operating system does not support fonts. Thus, **METAFONT**, an application to generate fonts, was developed in parallel with the  $\text{\TeX}$  application itself. **METAFONT** was then used to create a set of fonts that is referred to as computer modern (CM).

The CM set of fonts consists of 75 fonts that are described in volume E of [Knuth 86] as well as the “line,” “circle,” and symbol fonts associated with LaTeX.

The output of **METAFONT** is called a generic font (GF), to indicate that this file format does not follow the conventions of any font foundry. However, it is easy to convert GF font files to the special format required by most digital phototypesetting equipment.

In 1985, Tomas Rokicki, a student of Knuth's, developed the PK (for PacKed) font format. The idea was to develop a simple, fast compression method such that the fonts would be saved in small files and even the slow computers available at that time would be able to decompress a font, or any part of it, quickly. The main references for the format and organization of PK fonts are [Rokicki 95], [Rokicki 90], and [Haralambous 07].

The method selected for PK compression was based on run-length encoding (RLE) and on the special features of font bitmaps. The method is not very efficient, but it is described here because it offers an original approach to RLE, an approach that does not use Huffman codes and makes minimal use of variable-length codes (the packed numbers, described later, are the only variable-length code used by PK). A quick glance at the four bitmaps of Figure 5.38 shows two important features (1) they are narrow, resulting in mostly short runs of black and white pixels and (2) they tend to have consecutive identical rows of pixels.

This section discusses the compression of PK fonts, but a full understanding of this compression method requires a little knowledge of the organization of these fonts and of the way  $\text{\TeX}$  employs font information.

When a font of type is designed, the designer determines, for each symbol in the font, its glyph and its dimensions. Symbols in a font are two-dimensional, but each has

three dimensions, height, depth, and width. It is best to think of the symbol as if it is surrounded by a box (or a bitmap) as illustrated by the leftmost item of Figure 5.39. There is a baseline that separates the height and depth of the box, and there is a reference point at the left end of the baseline.

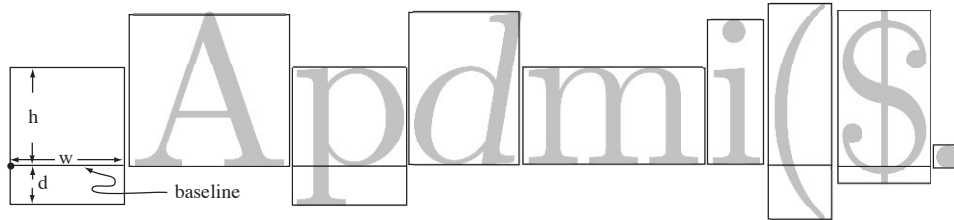


Figure 5.39: Three-Dimensional Character Boxes.

$\text{\TeX}$  uses the three dimensions of each font symbol but is not concerned with the actual shape of the symbol. As a result, symbols may stick out of their bitmap boxes, as illustrated by the italic *d* in the figure.  $\text{\TeX}$  reads the input text and strings boxes horizontally to construct a line of text. The boxes butt together (in Figure 5.39 there are small spaces between boxes for better readability), so the font designer leaves spaces to the left and right of each symbol, as illustrated in the figure. (In those rare cases where a symbol, such as an m-dash, should touch their neighbors, the bitmap box is as wide as the symbol and no spaces are left.)

When the current line of text is ready,  $\text{\TeX}$  starts on the next line. When that line is also ready, it is placed under the current line (and it then becomes the new current line) such that the baselines of the two lines are separated by a user-controlled parameter. If this causes the lines to overlap (because the top line has a symbol with a large depth and the bottom line has a symbol with a large height),  $\text{\TeX}$  leaves some space (Figure 5.40) between the bottom of the upper line (the symbol with the largest depth) and the top of the lower line (the symbol with the largest height). This space can also be controlled by the user, but the font designer does not have to worry about it. Thus, there are spaces to the left and right of font symbols in their bitmap boxes but no spaces above and below them.

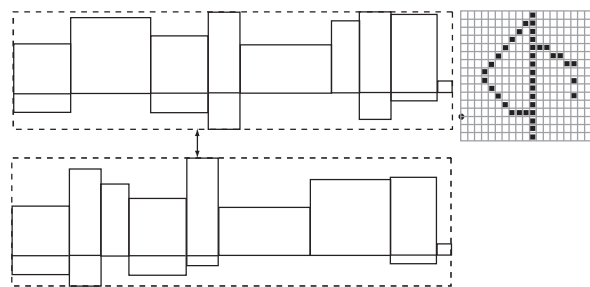


Figure 5.40: Two Lines of Text.

PK compression is based on run-length encoding. The method encodes the runs of black and white pixels in the bitmap of a symbol. In order to decrease the run

lengths and thereby increase compression performance, the encoder first determines the bounding box of a bitmap, as illustrated by the  $16 \times 20$  bitmap of Figure 5.40. The bounding box of a glyph bitmap is the smallest rectangle that encompasses all the black pixels of the glyph. It is easy to see that our bounding box is 15 pixels wide and 16 pixels tall. It starts on column 3 from the left and 13 of the 16 pixel rows are above the baseline. Once the bounding box is decompressed, the decoder needs the following dimensions to create the complete bitmap: The horizontal escapement (the width of the bitmap, 20), the width of the bounding box (15), the vertical escapement (the height of the bitmap, 16), the height of the bounding box (also 16), the horizontal offset (the distance between the reference point and leftmost column of the bounding box, 2), and the vertical offset (the distance from the reference point to the top of the bounding box, 13). Three of these dimensions are horizontal and three are vertical, but often only two vertical dimensions are needed, because the bitmap and bounding box tend to have the same height.

We are now ready to look at the details of the compression. A PK font file is organized in records, one for each symbol. A record starts with a character preamble that contains (in addition to other information) the five dimensions mentioned above and one bit that specifies the top-left pixel of the bounding box (1 for a black pixel and 0 for a white one). The character preamble is followed by the encoded run lengths of the symbol's pixels and by the preamble of the next character. Because of the use of nybbles to encode run lengths, the preamble may start in the middle of a byte. There are also a preamble and postamble for the entire file.

The first step of the encoder is to locate groups of repeating (i.e., identical consecutive) pixel rows in the bounding box. When a group of  $n$  such rows is located,  $n - 1$  of them are removed from the bounding box, and a repeat count of  $[n - 1]$  is suitably encoded somewhere between runs in the remaining row of the group. Figure 5.41 illustrates this process. Part (a) of the figure shows a  $5 \times 15$  bitmap (note that this is not a bounding box) where the three middle rows repeat. In part (b), two of these rows have been deleted and part (c) shows the valid points between runs where the repeat count [2] may be inserted.

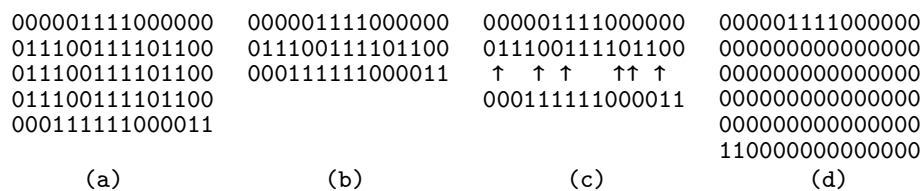


Figure 5.41: Repeating Rows in a Bitmap.

Thus, the run lengths of this example become the following sequence 5, 4, 7, [2], 3, 2, 4, 1, 2, 5, 6, 4, and 2. The repeat count was placed at the first valid point, but could have been placed at any of the other five points indicated in the figure. The repeat count for a group of identical rows is placed in the remaining row between runs of pixels, but this rule fails in bitmaps such as the one of Figure 5.41d, where the repeating rows are uniform and are preceded by a run of the same color pixels. In such a case, the encoder

does not delete the repeating rows and ends up encoding a long run (66 white pixels in the figure), which reduces the overall compression somewhat.

It is now clear that we have to encode two types of data, run lengths of pixels and repeat counts. Since a typical bounding box is small, we expect most run lengths to be short and most repeat counts to be small. Thus, the principle of PK compression is to pack, whenever possible, two items in the two nybbles of a byte. Often, the first nybble is a flag and the second nybble is a data item. A nybble is four bits long and can have 16 values. We expect to have more run lengths than repeat counts, so we reserve nybble values 14 and 15 to indicate repeat counts. A nybble of 15 indicates a repeat count of 1 (the most common) and a nybble of 14 will be followed by a repeat count stored as a packed number (a term to be discussed shortly).

The remaining nybble values 0 through 13 indicate run lengths. Value 0 indicates a long run, occupying three or more nybbles. The length of the run is stored in as many nybbles as needed, encoded as a packed number. Values 1 through 13 indicate the short and medium run lengths. In order to use these 13 values to maximum effect, the encoder counts the run lengths of the bounding box of the current character, and uses their values, in a fast, one-pass algorithm, to compute a variable *dyn*. This variable is computed for each character of the font and is stored in the character's preamble. Variable *dyn* is used as follows: run lengths 1 through *dyn* are considered short and are stored in one nybble each. The medium run lengths are stored in two nybbles, the first of which, to be denoted by *a*, is between *dyn* + 1 and 13 and the second one, *b*, can be any 4-bit value. The run length represented by *a* and *b* is defined as  $16(a - \text{dyn} - 1) + b + \text{dyn} + 1$ . The shortest run length is therefore *dyn* + 1 (for *a* = *dyn* + 1 and *b* = 0) and the longest is  $16(13 - \text{dyn}) + \text{dyn}$  (for *a* = 13 and *b* = 15). This convention is best illustrated by examples.

We first select *dyn* = 4. In this case, run lengths 1 through 4 are the short ones and are represented by one nybble each, thus 0001, 0010, 0011, and 0100. The medium run lengths are 5 (for *a* = 4 + 1 and *b* = 0) through  $16(13 - 4) + 4 = 148$  (for *a* = 13 and *b* = 15). Runs of more than 148 pixels are considered long and will start with a zero nybble.

We now select *dyn* = 12. In this case, run lengths 1 through 12 are short and are represented by one nybble each. The medium run lengths are 13 (for *a* = 12 + 1 and *b* = 0) through  $16(13 - 12) + 12 = 28$  (for *a* = 13 and *b* = 15). Runs of more than 28 pixels are considered long.

It is clear that the value of *dyn* is critical. Small *dyn* values allow for a large range of medium run lengths, while large values of *dyn* should be used in cases where most run lengths are short.

Next, we discuss the format of a packed number. Given an integer *i*, the idea is to create its hexadecimal representation (let's say it occupies *n* nybbles), remove any leading zero nybbles and prepend *n* - 1 zero nybbles. Thus, *i* values 1 through 15 occupy one nybble (nothing is prepended). Values  $16 \leq i \leq 255$  are two nybbles long, so one zero nybble should be prepended, for a total of three nybbles. Values  $256 \leq i \leq 4,095$  occupy three nybbles, so two nybbles should be prepended, for a total of five nybbles.

This variable-length format is used to encode repeat counts (indicated by a nybble flag of 14) and the long run lengths (indicated by a nybble flag of 0). However, the long run lengths are always greater than  $16(13 - \text{dyn}) + \text{dyn}$ . The shortest of the long run

lengths is therefore  $s \stackrel{\text{def}}{=} 16(13 - \text{dyn}) + \text{dyn} + 1$ , which is why it makes sense to subtract  $s$  from such a run length before it is encoded as a packed number. Recall that the long run lengths are indicated by a nybble flag of 0, and a packed integer in the interval  $[16, 255]$  is also preceded by a single zero nybble. Thus, it makes sense to add 16 to the long run lengths after  $s$  is subtracted.

Example. If  $\text{dyn} = 4$ , the longest medium run length is 148, so  $s = 149$ . Thus, a run length of 200 is long and is encoded by computing  $200 - 149 + 16 = 67 = 43_{16}$  and constructing the 3-nybble packed number  $043_{16}$ .

The algorithm for determining the optimal value of  $\text{dyn}$  (between 1 and 13) can now be described. This algorithm is executed after the sequence of run lengths has been determined. We start with a naive version of the algorithm. For each value of  $\text{dyn}$  between 1 and 13, it is easy to compute the ranges of the short, medium, and long runs. Each run in the sequence is examined, its type (short, medium, or long) is easily determined, and its length after being encoded is computed (short runs are one nybble long, medium runs are two nybbles, and long runs occupy three or more nybbles and their lengths take a bit more work to determine). The 13 total lengths for the values of  $\text{dyn}$  are saved and their minimum is then found.

A better version of this algorithm exploits the fact that as  $\text{dyn}$  is increased, the range of short run lengths increases while the range of medium run lengths decreases (for  $\text{dyn} = 4$  this range is  $[5, 148]$  but for  $\text{dyn} = 12$  it is only  $[13, 28]$ ). Thus, a run of pixels that was medium for a certain value of  $\text{dyn}$  may remain medium but may also become short or long when  $\text{dyn}$  is incremented by 1.

This version of the algorithm starts by setting  $\text{dyn}$  to zero. Thus, initially there are no short runs. The algorithm goes over all the run lengths. It determines the type and computes the encoded length of each run. The lengths are added into variable **total**. The algorithm then increments  $\text{dyn}$  by 1 and goes over all the runs again. If a run was short in the previous iteration, it will remain short. If it was medium and has now become short, it decreases the value of **total** by 1. If it was medium and has now become long, it increases the value of **total** by 1 (or in rare cases, by 2). After each iteration, the new value of **total** is saved in an array. After 13 iterations, the smallest **total** is located in the array and is used to determine the optimal value of  $\text{dyn}$ .

The PK compression method described so far is simple, but not very efficient. Given a bitmap with many short runs (such as a gray character, where black and white pixels alternate), this RLE-based method may produce large expansion. In such cases, the run length encoding described here is abandoned and the bitmap is simply written on the font file in raw format (one bit per pixel).

We end with a summarizing example. The middle bitmap of Figure 5.38 is the Greek letter  $\Xi$  (Xi). Its size is  $20 \times 29$  pixels and it has several groups of repeating rows, although only five groups are not uniform and can employ repeat counts. Counting the runs of pixels produces the sequence 82, [2], 16, 2, 42, [2], 2, 12, 2, 4, [3], 16, 4, [2], 2, 12, 2, 62, [2], 2, 16, and 82. The optimal value of  $\text{dyn}$  turns out to be 8, but in order to make this example useful and have short, medium, and long runs, we assume that  $\text{dyn}$  is set to 12. Thus, the short runs are 1 to 12 pixels, the medium runs are 13 to 28 pixels, and the long runs are longer than 28 pixels.

The first run is 82; a long run. Subtracting  $s = 29$  and adding 16 yields  $69 = 45_{16}$ . This is encoded as the three nybbles  $045$ . The repeat count of 2 is encoded as the nybble

$14 = E_{16}$ , followed by the packed number representation of 2, thus **E2**. The medium run of 16 is encoded in two nybbles *ab*, but since *a* is between  $dyn + 1$  and 13, its value must be  $13 = D_{16}$ , implying that  $b = 4$  and the run of 16 is encoded as **D4**. The next run, 2, is short and is encoded as the single nybble 2. The run 42 is long. We first compute  $42 - 29 + 16 = 29 = 1D_{16}$  and then encode 01D. The next two runs [2] and 2 are encoded as **E22**. So far we have 045E2D4201DE22. The next run, 12, is short and is encoded as the single nybble **C**. The remaining runs do not provide any new examples and should be encoded by the reader as an exercise.

## 5.9 Arithmetic Coding

The Huffman method is simple, efficient, and produces the best codes for the individual data symbols. However, Section 5.2 shows that the only case where it produces ideal variable-length 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 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 first example involves the three symbols  $a_1$ ,  $a_2$ , and  $a_3$ , with probabilities  $P_1 = 0.4$ ,  $P_2 = 0.5$ , and  $P_3 = 0.1$ , respectively. The interval  $[0, 1)$  is divided among the three symbols by assigning each a subinterval proportional in size to its probability. The order of the subintervals is immaterial. In our example, the three symbols are assigned the subintervals  $[0, 0.4)$ ,  $[0.4, 0.9)$ , and  $[0.9, 1.0)$ . To encode the string “ $a_2a_2a_2a_3$ ”, we start with the interval  $[0, 1)$ . The first symbol  $a_2$  reduces this interval to the subinterval from its 40% point to its 90% point. The result is  $[0.4, 0.9)$ . The second  $a_2$  reduces  $[0.4, 0.9)$  in the same way (see note below) to  $[0.6, 0.85)$ , the third  $a_2$  reduces this to  $[0.7, 0.825)$ , and the  $a_3$  reduces this to the stretch from the 90% point of  $[0.7, 0.825)$  to its 100% point, producing  $[0.8125, 0.8250)$ . The final code our method produces can be any number in this final range.

(Note: The subinterval  $[0.6, 0.85)$  is obtained from the interval  $[0.4, 0.9)$  by  $0.4 + (0.9 - 0.4) \times 0.4 = 0.6$  and  $0.4 + (0.9 - 0.4) \times 0.9 = 0.85$ .)

With this example in mind, it should be easy to understand the following rules, which summarize the main steps of arithmetic coding:

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 (Section 5.10).

A theory has only the alternative of being right or wrong. A model has a third possibility: it may be right, but irrelevant.

—Manfred Eigen, *The Physicist’s Conception of Nature*

The next example is a little more involved. We show the compression steps for the short string SWISS\_MISS. Table 5.42 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). The cumulative frequencies column is used by the decoding algorithm on page 271.

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 5.42: Frequencies and Probabilities of Five Symbols.

The symbols and frequencies in Table 5.42 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

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

where  $\text{Range} = \text{OldHigh} - \text{OldLow}$  and  $\text{LowRange}(X)$ ,  $\text{HighRange}(X)$  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  $\text{Low} := 0.5 + (1.0 - 0.5) \times 0.4 = 0.70$ ,  $\text{High} := 0.5 + (1.0 - 0.5) \times 0.5 = 0.75$ . The new interval [0.70, 0.75) covers the stretch [40%, 50%) of the subrange of **S**. Table 5.43 shows all the steps involved in coding the string **SWISS□MISS** (the first three steps are illustrated graphically in Figure 5.56a). 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 (but see later for a modification of this statement).

The decoder works in reverse. It starts by inputting the symbols and their ranges, and reconstructing Table 5.42. 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....



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	$0.715 + (0.72 - 0.715) \times 1.0 = 0.72$
□	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 5.43: The Process of Arithmetic Encoding.

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  $\text{Code} := (\text{Code} - \text{LowRange}(\text{X})) / \text{Range}$ , where **Range** is the width of the subrange of **X**. Table 5.45 summarizes the steps for decoding our example string (notice that it has two rows per symbol).

The next example is of three symbols with probabilities as shown in Table 5.46a. Notice that the probabilities are very different. One is large (97.5%) and the others much smaller. This is a case of *skewed probabilities*.

Encoding the string  $a_2a_2a_1a_3a_3$  produces the strange numbers (accurate to 16 digits) in Table 5.47, where the two rows for each symbol correspond to the **Low** and **High** values, respectively. Figure 5.44 lists the *Mathematica* code that computed the table.

At first glance, it seems that the resulting code is longer than the original string, but Section 5.9.3 shows how to figure out the true compression achieved by arithmetic coding.

Decoding this string is shown in Table 5.48 and involves a special problem. After eliminating the effect of  $a_1$ , on line 3, the result is 0. Earlier, we implicitly assumed

that this means the end of the decoding process, but now we know that there are two more occurrences of  $a_3$  that should be decoded. These are shown on lines 4, 5 of the table. This problem always occurs when the last symbol in the input stream is the one whose subrange starts at zero. In order to distinguish between such a symbol and the end of the input stream, we need to define an additional symbol, the end-of-input (or end-of-file, eof). This symbol should be added, with a small probability, to the frequency table (see Table 5.46b), and it should be encoded at the end of the input stream.

```
lowRange={0.998162,0.023162,0.};
highRange={1.,0.998162,0.023162};
low=0.; high=1.;
enc[i_]:=Module[{nlow,nhigh,range},
range=high-low;
nhigh=low+range highRange[[i]];
nlow=low+range lowRange[[i]];
low=nlow; high=nhigh;
Print["r=",N[range,25]," l=",N[low,17]," h=",N[high,17]]]
enc[2]
enc[2]
enc[1]
enc[3]
enc[3]
```

Figure 5.44: *Mathematica* Code for Table 5.47.

Tables 5.49 and 5.50 show how the string  $a_3a_3a_3a_3\text{eof}$  is encoded into the number 0.0000002878086184764172, and then decoded properly. Without the eof symbol, a string of all  $a_3$ s would have been encoded into a 0.

Notice how the low value is 0 until the eof is input and processed, and how the high value quickly approaches 0. Now is the time to mention that the final code does not have to be the final low value but can be any number between the final low and high values. In the example of  $a_3a_3a_3a_3\text{eof}$ , the final code can be the much shorter number 0.0000002878086 (or 0.0000002878087 or even 0.0000002878088).

- ◇ **Exercise 5.24:** Encode the string  $a_2a_2a_2a_2$  and summarize the results in a table similar to Table 5.49. How do the results differ from those of the string  $a_3a_3a_3a_3$ ?

If the size of the input stream is known, it is possible to do without an eof symbol. The encoder can start by writing this size (unencoded) on the output stream. The decoder reads the size, starts decoding, and stops when the decoded stream reaches this size. If the decoder reads the compressed stream byte by byte, the encoder may have to add some zeros at the end, to make sure the compressed stream can be read in groups of 8 bits.

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	$0.50675 - 0.5 = 0.00675$	$/0.5 = 0.0135$
□	$0.0135 - 0 = 0.0135$	$/0.1 = 0.135$
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 5.45: The Process of Arithmetic Decoding.

Char	Prob.	Range	Char	Prob.	Range
$a_1$	0.001838	[0.998162, 1.0)	eof	0.000001	[0.999999, 1.0)
$a_2$	0.975	[0.023162, 0.998162)	$a_1$	0.001837	[0.998162, 0.999999)
$a_3$	0.023162	[0.0, 0.023162)	$a_2$	0.975	[0.023162, 0.998162)
			$a_3$	0.023162	[0.0, 0.023162)

(a)
(b)

Table 5.46: (Skewed) Probabilities of Three Symbols.

$a_2$	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$
	$0.0 + (1.0 - 0.0) \times 0.998162 = 0.998162$
$a_2$	$0.023162 + .975 \times 0.023162 = 0.04574495$
	$0.023162 + .975 \times 0.998162 = 0.99636995$
$a_1$	$0.04574495 + 0.950625 \times 0.998162 = 0.99462270125$
	$0.04574495 + 0.950625 \times 1.0 = 0.99636995$
$a_3$	$0.99462270125 + 0.00174724875 \times 0.0 = 0.99462270125$
	$0.99462270125 + 0.00174724875 \times 0.023162 = 0.994663171025547$
$a_3$	$0.99462270125 + 0.00004046977554749998 \times 0.0 = 0.99462270125$
	$0.99462270125 + 0.00004046977554749998 \times 0.023162 = 0.994623638610941$

Table 5.47: Encoding the String  $a_2a_2a_1a_3a_3$ .

Char.	Code—low	Range
$a_2$	$0.99462270125 - 0.023162 = 0.97146170125$	$/0.975 = 0.99636995$
$a_2$	$0.99636995 - 0.023162 = 0.97320795$	$/0.975 = 0.998162$
$a_1$	$0.998162 - 0.998162 = 0.0$	$/0.00138 = 0.0$
$a_3$	$0.0 - 0.0 = 0.0$	$/0.023162 = 0.0$
$a_3$	$0.0 - 0.0 = 0.0$	$/0.023162 = 0.0$

Table 5.48: Decoding the String  $a_2a_2a_1a_3a_3$ .

$a_3$	$0.0 + (1.0 - 0.0) \times 0.0 = 0.0$
	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$
$a_3$	$0.0 + .023162 \times 0.0 = 0.0$
	$0.0 + .023162 \times 0.023162 = 0.000536478244$
$a_3$	$0.0 + 0.000536478244 \times 0.0 = 0.0$
	$0.0 + 0.000536478244 \times 0.023162 = 0.000012425909087528$
$a_3$	$0.0 + 0.000012425909087528 \times 0.0 = 0.0$
	$0.0 + 0.000012425909087528 \times 0.023162 = 0.0000002878089062853235$
eof	$0.0 + 0.0000002878089062853235 \times 0.999999 = 0.0000002878086184764172$
	$0.0 + 0.0000002878089062853235 \times 1.0 = 0.0000002878089062853235$

Table 5.49: Encoding the String  $a_3a_3a_3a_3$ eof.

Char.	Code—low	Range
$a_3$	$0.0000002878086184764172 - 0 = 0.0000002878086184764172$	$/0.023162 = 0.00001242589666161891247$
$a_3$	$0.00001242589666161891247 - 0 = 0.00001242589666161891247$	$/0.023162 = 0.000536477707521756$
$a_3$	$0.000536477707521756 - 0 = 0.000536477707521756$	$/0.023162 = 0.023161976838$
$a_3$	$0.023161976838 - 0 = 0.023161976838$	$/0.023162 = 0.999999$
eof	$0.999999 - 0.999999 = 0.0$	$/0.000001 = 0.0$

Table 5.50: Decoding the String  $a_3a_3a_3a_3$ eof.

### 5.9.1 Implementation Details

The encoding process described earlier is not practical, since it assumes that numbers of unlimited precision can be stored in **Low** and **High**. The decoding process described on page 267 (“The decoder then eliminates the effect of the **S** from the code by subtracting... and dividing...” ) is simple in principle but also impractical. The code, which is a single number, is normally long and may also be very long. A 1 Mbyte file may be encoded into, say, a 500 Kbyte file that consists of a single number. Dividing a 500 Kbyte number is complex and slow.

Any practical implementation of arithmetic coding should use just integers (because floating-point arithmetic is slow and precision is lost), and they should not be very long

(preferably just single precision). We describe such an implementation here, using two integer variables **Low** and **High**. In our example they are four decimal digits long, but in practice they might be 16 or 32 bits long. These variables hold the low and high limits of the current subinterval, but we don't let them grow too much. A glance at Table 5.43 shows that once the leftmost digits of **Low** and **High** become identical, they never change. We therefore shift such digits out of the two variables and write one digit on the output stream. This way, the two variables don't have to hold the entire code, just the most-recent part of it. As digits are shifted out of the two variables, a zero is shifted into the right end of **Low** and a 9 into the right end of **High**. A good way to understand this is to think of each of the two variables as the left end of an infinitely long number. **Low** contains *xxxx00...*, and **High**= *yyyy99...*.

One problem is that **High** should be initialized to 1, but the contents of **Low** and **High** should be interpreted as fractions less than 1. The solution is to initialize **High** to 9999..., since the infinite fraction 0.999... equals 1.

(This is easy to prove. If  $0.999... < 1$ , then their average  $a = (1 + 0.999...)/2$  would be a number between 0.999... and 1, but there is no way to write  $a$ . It is impossible to give it more digits than to 0.999..., since the latter already has an infinite number of digits. It is impossible to make the digits any bigger, since they are already 9's. This is why the infinite fraction 0.999... must equal 1.)

◇ **Exercise 5.25:** Write the number 0.5 in binary.

Table 5.51 describes the encoding process of the string **SWISS\_MISS**. Column 1 shows the next input symbol. Column 2 shows the new values of **Low** and **High**. Column 3 shows these values as scaled integers, after **High** has been decremented by 1. Column 4 shows the next digit sent to the output stream. Column 5 shows the new values of **Low** and **High** after being shifted to the left. Notice how the last step sends the four digits 3750 to the output stream. The final output is 717533750.

Decoding is the opposite of encoding. We start with **Low**=0000, **High**=9999, and **Code**=7175 (the first four digits of the compressed stream). These are updated at each step of the decoding loop. **Low** and **High** approach each other (and both approach **Code**) until their most significant digits are the same. They are then shifted to the left, which separates them again, and **Code** is also shifted at that time. An index is calculated at each step and is used to search the cumulative frequencies column of Table 5.42 to figure out the current symbol.

Each iteration of the loop consists of the following steps:

1. Calculate  $\text{index} := ((\text{Code} - \text{Low} + 1) \times 10 - 1) / (\text{High} - \text{Low} + 1)$  and truncate it to the nearest integer. (The number 10 is the total cumulative frequency in our example.)
2. Use **index** to find the next symbol by comparing it to the cumulative frequencies column in Table 5.42. In the example below, the first value of **index** is 7.1759, truncated to 7. Seven is between the 5 and the 10 in the table, so it selects the **S**.
3. Update **Low** and **High** according to

```
Low:=Low+(High-Low+1)LowCumFreq[X]/10;
High:=Low+(High-Low+1)HighCumFreq[X]/10-1;
```

where **LowCumFreq[X]** and **HighCumFreq[X]** are the cumulative frequencies of symbol **X** and of the symbol above it in Table 5.42.

	1	2	3	4	5
S	$L = 0+(1 - 0) \times 0.5 = 0.5$	5000	5000		
	$H = 0+(1 - 0) \times 1.0 = 1.0$	9999	9999		
W	$L = 0.5+(1 - .5) \times 0.4 = 0.7$	7000	7	0000	
	$H = 0.5+(1 - .5) \times 0.5 = 0.75$	7499	7	4999	
I	$L = 0+(0.5 - 0) \times 0.2 = 0.1$	1000	1	0000	
	$H = 0+(0.5 - 0) \times 0.4 = 0.2$	1999	1	9999	
S	$L = 0+(1 - 0) \times 0.5 = 0.5$	5000		5000	
	$H = 0+(1 - 0) \times 1.0 = 1.0$	9999		9999	
S	$L = 0.5+(1 - 0.5) \times 0.5 = 0.75$	7500		7500	
	$H = 0.5+(1 - 0.5) \times 1.0 = 1.0$	9999		9999	
□	$L = 0.75+(1 - 0.75) \times 0.0 = 0.75$	7500	7	5000	
	$H = 0.75+(1 - 0.75) \times 0.1 = 0.775$	7749	7	7499	
M	$L = 0.5+(0.75 - 0.5) \times 0.1 = 0.525$	5250	5	2500	
	$H = 0.5+(0.75 - 0.5) \times 0.2 = 0.55$	5499	5	4999	
I	$L = 0.25+(0.5 - 0.25) \times 0.2 = 0.3$	3000	3	0000	
	$H = 0.25+(0.5 - 0.25) \times 0.4 = 0.35$	3499	3	4999	
S	$L = 0+(0.5 - 0) \times 0.5 = .25$	2500		2500	
	$H = 0+(0.5 - 0) \times 1.0 = 0.5$	4999		4999	
S	$L = 0.25+(0.5 - 0.25) \times 0.5 = 0.375$	3750		3750	
	$H = 0.25+(0.5 - 0.25) \times 1.0 = 0.5$	4999		4999	

Table 5.51: Encoding SWISS□MISS by Shifting.

4. If the leftmost digits of **Low** and **High** are identical, shift **Low**, **High**, and **Code** one position to the left. **Low** gets a 0 entered on the right, **High** gets a 9, and **Code** gets the next input digit from the compressed stream.

Here are all the decoding steps for our example:

0. Initialize **Low**=0000, **High**=9999, and **Code**=7175.

1.  $\text{index} = [(7175 - 0 + 1) \times 10 - 1] / (9999 - 0 + 1) = 7.1759 \rightarrow 7$ . Symbol **S** is selected. **Low** =  $0 + (9999 - 0 + 1) \times 5/10 = 5000$ . **High** =  $0 + (9999 - 0 + 1) \times 10/10 - 1 = 9999$ .

2.  $\text{index} = [(7175 - 5000 + 1) \times 10 - 1] / (9999 - 5000 + 1) = 4.3518 \rightarrow 4$ . Symbol **W** is selected. **Low** =  $5000 + (9999 - 5000 + 1) \times 4/10 = 7000$ . **High** =  $5000 + (9999 - 5000 + 1) \times 5/10 - 1 = 7499$ .

After the 7 is shifted out, we have **Low**=0000, **High**=4999, and **Code**=1753.

3.  $\text{index} = [(1753 - 0 + 1) \times 10 - 1] / (4999 - 0 + 1) = 3.5078 \rightarrow 3$ . Symbol **I** is selected. **Low** =  $0 + (4999 - 0 + 1) \times 2/10 = 1000$ . **High** =  $0 + (4999 - 0 + 1) \times 4/10 - 1 = 1999$ . After the 1 is shifted out, we have **Low**=0000, **High**=9999, and **Code**=7533.

4.  $\text{index} = [(7533 - 0 + 1) \times 10 - 1] / (9999 - 0 + 1) = 7.5339 \rightarrow 7$ . Symbol **S** is selected. **Low** =  $0 + (9999 - 0 + 1) \times 5/10 = 5000$ . **High** =  $0 + (9999 - 0 + 1) \times 10/10 - 1 = 9999$ .

5.  $\text{index} = [(7533 - 5000 + 1) \times 10 - 1] / (9999 - 5000 + 1) = 5.0678 \rightarrow 5$ . Symbol **S** is selected.

$\text{Low} = 5000 + (9999 - 5000 + 1) \times 5 / 10 = 7500$ .  $\text{High} = 5000 + (9999 - 5000 + 1) \times 10 / 10 - 1 = 9999$ .

6.  $\text{index} = [(7533 - 7500 + 1) \times 10 - 1] / (9999 - 7500 + 1) = 0.1356 \rightarrow 0$ . Symbol **□** is selected.

$\text{Low} = 7500 + (9999 - 7500 + 1) \times 0 / 10 = 7500$ .  $\text{High} = 7500 + (9999 - 7500 + 1) \times 1 / 10 - 1 = 7749$ .

After the 7 is shifted out, we have  $\text{Low}=5000$ ,  $\text{High}=7499$ , and  $\text{Code}=5337$ .

7.  $\text{index} = [(5337 - 5000 + 1) \times 10 - 1] / (7499 - 5000 + 1) = 1.3516 \rightarrow 1$ . Symbol **M** is selected.

$\text{Low} = 5000 + (7499 - 5000 + 1) \times 1 / 10 = 5250$ .  $\text{High} = 5000 + (7499 - 5000 + 1) \times 2 / 10 - 1 = 5499$ .

After the 5 is shifted out we have  $\text{Low}=2500$ ,  $\text{High}=4999$ , and  $\text{Code}=3375$ .

8.  $\text{index} = [(3375 - 2500 + 1) \times 10 - 1] / (4999 - 2500 + 1) = 3.5036 \rightarrow 3$ . Symbol **I** is selected.

$\text{Low} = 2500 + (4999 - 2500 + 1) \times 2 / 10 = 3000$ .  $\text{High} = 2500 + (4999 - 2500 + 1) \times 4 / 10 - 1 = 3499$ .

After the 3 is shifted out we have  $\text{Low}=0000$ ,  $\text{High}=4999$ , and  $\text{Code}=3750$ .

9.  $\text{index} = [(3750 - 0 + 1) \times 10 - 1] / (4999 - 0 + 1) = 7.5018 \rightarrow 7$ . Symbol **S** is selected.

$\text{Low} = 0 + (4999 - 0 + 1) \times 5 / 10 = 2500$ .  $\text{High} = 0 + (4999 - 0 + 1) \times 10 / 10 - 1 = 4999$ .

10.  $\text{index} = [(3750 - 2500 + 1) \times 10 - 1] / (4999 - 2500 + 1) = 5.0036 \rightarrow 5$ . Symbol **S** is selected.

$\text{Low} = 2500 + (4999 - 2500 + 1) \times 5 / 10 = 3750$ .  $\text{High} = 2500 + (4999 - 2500 + 1) \times 10 / 10 - 1 = 4999$ .

◇ **Exercise 5.26:** How does the decoder know to stop the loop at this point?

1	2	3	4	5
1	$L=0+(1-0) \times 0.0 = 0.0$	000000	0	000000
	$H=0+(1-0) \times 0.023162 = 0.023162$	023162	0	231629
2	$L=0+(0.231629-0) \times 0.0 = 0.0$	000000	0	000000
	$H=0+(0.231629-0) \times 0.023162 = 0.00536478244$	005364	0	053649
3	$L=0+(0.053649-0) \times 0.0 = 0.0$	000000	0	000000
	$H=0+(0.053649-0) \times 0.023162 = 0.00124261813$	001242	0	012429
4	$L=0+(0.012429-0) \times 0.0 = 0.0$	000000	0	000000
	$H=0+(0.012429-0) \times 0.023162 = 0.00028788049$	000287	0	002879
5	$L=0+(0.002879-0) \times 0.0 = 0.0$	000000	0	000000
	$H=0+(0.002879-0) \times 0.023162 = 0.00006668339$	000066	0	000669

Table 5.52: Encoding  $a_3a_3a_3a_3$  by Shifting.

### 5.9.2 Underflow

Table 5.52 shows the steps in encoding the string  $a_3a_3a_3a_3a_3$  by shifting. This table is similar to Table 5.51, and it illustrates the problem of underflow. **Low** and **High** approach each other, and since **Low** is always 0 in this example, **High** loses its significant digits as it approaches **Low**.

Underflow may happen not just in this case but in any case where **Low** and **High** need to converge very closely. Because of the finite size of the **Low** and **High** variables, they may reach values of, say, 499996 and 500003, and from there, instead of reaching values where their most significant digits are identical, they reach the values 499999 and 500000. Since the most significant digits are different, the algorithm will not output anything, there will not be any shifts, and the next iteration will only add digits beyond the first six ones. Those digits will be lost, and the first six digits will not change. The algorithm will iterate without generating any output until it reaches the eof.

The solution to this problem is to detect such a case early and *rescale* both variables. In the example above, rescaling should be done when the two variables reach values of 49xxxx and 50yyyy. Rescaling should squeeze out the second most significant digits, end up with 4xxxx0 and 5yyyy9, and increment a counter **cntr**. The algorithm may have to rescale several times before the most-significant digits become equal. At that point, the most-significant digit (which can be either 4 or 5) should be output, followed by **cntr** zeros (if the two variables converged to 4) or nines (if they converged to 5).

### 5.9.3 Final Remarks

All the examples so far have been in decimal, since the computations involved are easier to understand in this number base. It turns out that all the algorithms and rules described above apply to the binary case as well and can be used with only one change: Every occurrence of 9 (the largest decimal digit) should be replaced by 1 (the largest binary digit).

The examples above don't seem to show any compression at all. It seems that the three example strings **SWISS\_MISS**,  $a_2a_2a_1a_3a_3$ , and  $a_3a_3a_3a_3eof$  are encoded into very long numbers. In fact, it seems that the length of the final code depends on the probabilities involved. The long probabilities of Table 5.46a generate long numbers in the encoding process, whereas the shorter probabilities of Table 5.42 result in the more reasonable **Low** and **High** values of Table 5.43. This behavior demands an explanation.

<p>I am ashamed to tell you to how many figures I carried these computations, having no other business at that time.</p>
--------------------------------------------------------------------------------------------------------------------------

—Isaac Newton

To figure out the kind of compression achieved by arithmetic coding, we have to consider two facts: (1) In practice, all the operations are performed on binary numbers, so we have to translate the final results to binary before we can estimate the efficiency of the compression; (2) since the last symbol encoded is the eof, the final code does not have to be the final value of **Low**; it can be any value between **Low** and **High**. This makes it possible to select a shorter number as the final code that's being output.

Table 5.43 encodes the string **SWISS\_MISS** into the final **Low** and **High** values 0.71753375 and 0.717535. The approximate binary values of these numbers are



0.10110111101100000100101010111 and 0.1011011110110000010111111011, so we can select the number 10110111101100000100 as our final, compressed output. The ten-symbol string has thus been encoded into a 20-bit number. Does this represent good compression?

The answer is yes. Using the probabilities of Table 5.42, it is easy to calculate the probability of the string `SWISS_MISS`. It is  $P = 0.5^5 \times 0.1 \times 0.2^2 \times 0.1 \times 0.1 = 1.25 \times 10^{-6}$ . The entropy of this string is therefore  $-\log_2 P = 19.6096$ . Twenty bits are therefore the minimum needed in practice to encode the string.

The symbols in Table 5.46a have probabilities 0.975, 0.001838, and 0.023162. These numbers require quite a few decimal digits, and as a result, the final Low and High values in Table 5.47 are the numbers 0.99462270125 and 0.994623638610941. Again it seems that there is no compression, but an analysis similar to the above shows compression that's very close to the entropy.

The probability of the string  $a_2 a_2 a_1 a_3 a_3$  is  $0.975^2 \times 0.001838 \times 0.023162^2 \approx 9.37361 \times 10^{-7}$ , and  $-\log_2 9.37361 \times 10^{-7} \approx 20.0249$ .

The binary representations of the final values of Low and High in Table 5.47 are 0.11111110100111111001011111001 and 0.111111101001111110100111101. We can select any number between these two, so we select 1111111010011111100, a 19-bit number. (This should have been a 21-bit number, but the numbers in Table 5.47 have limited precision and are not exact.)

- ◇ **Exercise 5.27:** Given the three symbols  $a_1$ ,  $a_2$ , and eof, with probabilities  $P_1 = 0.4$ ,  $P_2 = 0.5$ , and  $P_{\text{eof}} = 0.1$ , encode the string  $a_2 a_2 a_2 \text{eof}$  and show that the size of the final code equals the (practical) minimum.

The following argument shows why arithmetic coding can, in principle, be a very efficient compression method. We denote by  $s$  a sequence of symbols to be encoded, and by  $b$  the number of bits required to encode it. As  $s$  gets longer, its probability  $P(s)$  gets smaller and  $b$  gets larger. Since the logarithm is the information function, it is easy to see that  $b$  should grow at the same rate that  $\log_2 P(s)$  shrinks. Their product should therefore be constant, or close to a constant. Information theory shows that  $b$  and  $P(s)$  satisfy the double inequality

$$2 \leq 2^b P(s) < 4,$$

which implies

$$1 - \log_2 P(s) \leq b < 2 - \log_2 P(s). \quad (5.1)$$

As  $s$  gets longer, its probability  $P(s)$  shrinks, the quantity  $-\log_2 P(s)$  becomes a large positive number, and the double inequality of Equation (5.1) shows that in the limit,  $b$  approaches  $-\log_2 P(s)$ . This is why arithmetic coding can, in principle, compress a string of symbols to its theoretical limit.

For more information on this topic, see [Moffat et al. 98] and [Witten et al. 87].

## 5.10 Adaptive Arithmetic Coding

Two features of arithmetic coding make it easy to extend:

1. One of the main encoding steps (page 266) updates `NewLow` and `NewHigh`. Similarly, one of the main decoding steps (step 3 on page 271) updates `Low` and `High` according to

```
Low:=Low+(High-Low+1)LowCumFreq[X]/10;
High:=Low+(High-Low+1)HighCumFreq[X]/10-1;
```

This means that in order to encode symbol `X`, the encoder should be given the cumulative frequencies of the symbol and of the one above it (see Table 5.42 for an example of cumulative frequencies). This also implies that the frequency of `X` (or, equivalently, its probability) could be changed each time it is encoded, provided that the encoder and the decoder agree on how to do this.

2. The order of the symbols in Table 5.42 is unimportant. They can even be swapped in the table during the encoding process as long as the encoder and decoder do it in the same way.

With this in mind, it is easy to understand how adaptive arithmetic coding works. The encoding algorithm has two parts: the probability model and the arithmetic encoder. The model reads the next symbol from the input stream and invokes the encoder, sending it the symbol and the two required cumulative frequencies. The model then increments the count of the symbol and updates the cumulative frequencies. The point is that the symbol's probability is determined by the model from its *old* count, and the count is incremented only after the symbol has been encoded. This makes it possible for the decoder to mirror the encoder's operations. The encoder knows what the symbol is even before it is encoded, but the decoder has to decode the symbol in order to find out what it is. The decoder can therefore use only the old counts when decoding a symbol. Once the symbol has been decoded, the decoder increments its count and updates the cumulative frequencies in exactly the same way as the encoder.

The model should keep the symbols, their counts (frequencies of occurrence), and their cumulative frequencies in an array. This array should be kept in sorted order of the counts. Each time a symbol is read and its count is incremented, the model updates the cumulative frequencies, then checks to see whether it is necessary to swap the symbol with another one, to keep the counts in sorted order.

It turns out that there is a simple data structure that allows for both easy search and update. This structure is a balanced binary tree housed in an array. (A balanced binary tree is a complete binary tree where some of the bottom-right nodes may be missing.) The tree should have a node for every symbol in the alphabet, and since it is balanced, its height is  $\lceil \log_2 n \rceil$ , where  $n$  is the size of the alphabet. For  $n = 256$  the height of the balanced binary tree is 8, so starting at the root and searching for a node takes at most eight steps. The tree is arranged such that the most probable symbols (the ones with high counts) are located near the root, which speeds up searches. Table 5.53a shows an example of a ten-symbol alphabet with counts. Table 5.53b shows the same symbols sorted by count.

The sorted array “houses” the balanced binary tree of Figure 5.55a. This is a simple, elegant way to build a tree. A balanced binary tree can be housed in an array without the use of any pointers. The rule is that the first array location (with index 1) houses

$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_{10}$
11	12	12	2	5	1	2	19	12	8

(a)

$a_8$	$a_2$	$a_3$	$a_9$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	12	12	12	11	8	5	2	2	1

(b)

Table 5.53: A Ten-Symbol Alphabet With Counts.

the root, the two children of the node at array location  $i$  are housed at locations  $2i$  and  $2i + 1$ , and the parent of the node at array location  $j$  is housed at location  $\lfloor j/2 \rfloor$ . It is easy to see how sorting the array has placed the symbols with largest counts at and near the root.

In addition to a symbol and its count, another value is now added to each tree node, the total counts of its left subtree. This will be used to compute cumulative frequencies. The corresponding array is shown in Table 5.54a.

Assume that the next symbol read from the input stream is  $a_9$ . Its count is incremented from 12 to 13. The model keeps the array in sorted order by searching for the farthest array element left of  $a_9$  that has a count smaller than that of  $a_9$ . This search can be a straight linear search if the array is short enough, or a binary search if the array is long. In our case, symbols  $a_9$  and  $a_2$  should be swapped (Table 5.54b). Figure 5.55b shows the tree after the swap. Notice how the left-subtree counts have been updated.

$a_8$	$a_2$	$a_3$	$a_9$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	12	12	12	11	8	5	2	2	1
40	16	8	2	1	0	0	0	0	0

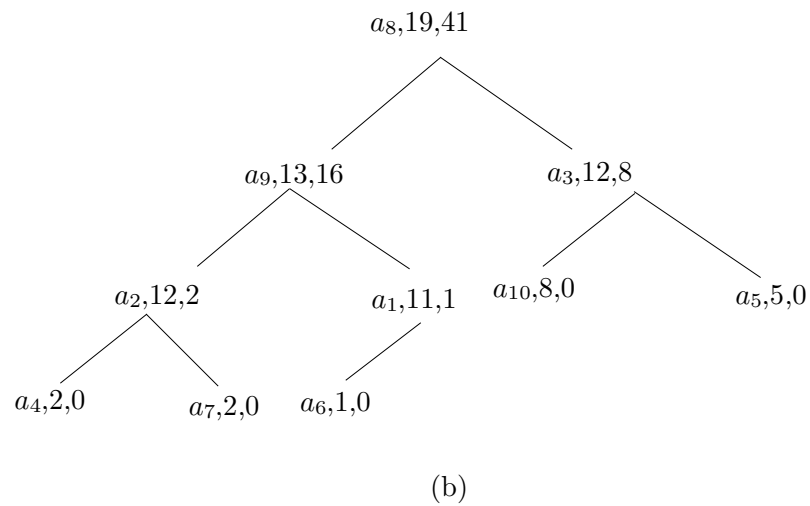
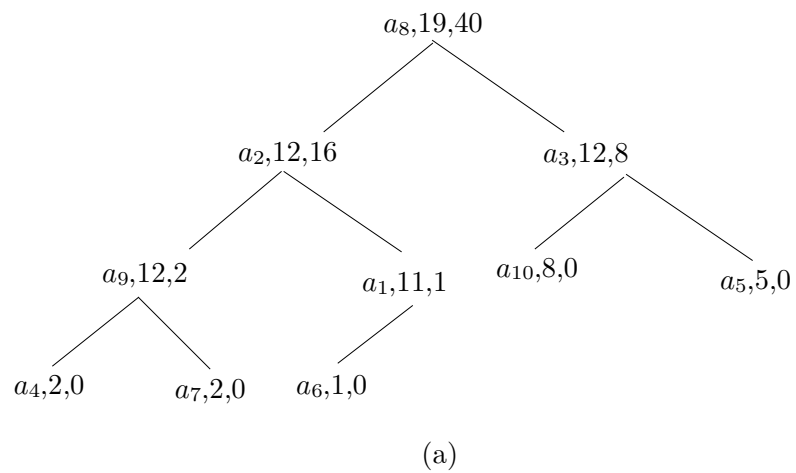
(a)

$a_8$	$a_9$	$a_3$	$a_2$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	13	12	12	11	8	5	2	2	1
41	16	8	2	1	0	0	0	0	0

(b)

Tables 5.54: A Ten-Symbol Alphabet With Counts.

Finally, here is how the cumulative frequencies are computed from this tree. When the cumulative frequency for a symbol  $X$  is needed, the model follows the tree branches from the root to the node containing  $X$  while adding numbers into an integer **af**. Each time a right branch is taken from an interior node  $N$ , **af** is incremented by the two numbers (the count and the left-subtree count) found in that node. When a left branch is taken, **af** is not modified. When the node containing  $X$  is reached, the left-subtree count of  $X$  is added to **af**, and **af** then contains the quantity **LowCumFreq**[ $X$ ].



$a_4$	2	0—1
$a_9$	12	2—13
$a_7$	2	14—15
$a_2$	12	16—27
$a_6$	1	28—28
$a_1$	11	29—39
$a_8$	19	40—58
$a_{10}$	8	59—66
$a_3$	12	67—78
$a_5$	5	79—83

(c)

Figure 5.55: Adaptive Arithmetic Coding.

As an example, we trace the tree of Figure 5.55a from the root to symbol  $a_6$ , whose cumulative frequency is 28. A right branch is taken at node  $a_2$ , adding 12 and 16 to **af**. A left branch is taken at node  $a_1$ , adding nothing to **af**. When reaching  $a_6$ , its left-subtree count, 0, is added to **af**. The result in **af** is  $12 + 16 = 28$ , as can be verified from Figure 5.55c. The quantity **HighCumFreq**[X] is obtained by adding the count of  $a_6$  (which is 1) to **LowCumFreq**[X].

To trace the tree and find the path from the root to  $a_6$ , the algorithm performs the following steps:

1. Find  $a_6$  in the array housing the tree by means of a binary search. In our example the node with  $a_6$  is found at array location 10.
2. Integer-divide 10 by 2. The remainder is 0, which means that  $a_6$  is the left child of its parent. The quotient is 5, which is the array location of the parent.
3. Location 5 of the array contains  $a_1$ . Integer-divide 5 by 2. The remainder is 1, which means that  $a_1$  is the right child of its parent. The quotient is 2, which is the array location of  $a_1$ 's parent.
4. Location 2 of the array contains  $a_2$ . Integer-divide 2 by 2. The remainder is 0, which means that  $a_2$  is the left child of its parent. The quotient is 1, the array location of the root, so the process stops.

The PPM compression method, Section 5.14, is a good example of a statistical model that invokes an arithmetic encoder in the way described here.

The driver held out a letter. Boldwood seized it and opened it, expecting another anonymous one—so greatly are people's ideas of probability a mere sense that precedent will repeat itself. "I don't think it is for you, sir," said the man, when he saw Boldwood's action. "Though there is no name I think it is for your shepherd."

—Thomas Hardy, *Far From The Madding Crowd*

### 5.10.1 Range Encoding

The use of integers in arithmetic coding is a must in any practical implementation, but it results in slow encoding because of the need for frequent renormalizations. The main steps in any integer-based arithmetic coding implementation are (1) proportional range reduction and (2) range expansion (renormalization).

Range encoding (or range coding) is an improvement to arithmetic coding that reduces the number of renormalizations and thereby speeds up integer-based arithmetic coding by factors of up to 2. The main references are [Schindler 98] and [Campos 06], and the description here is based on the former.

The main idea is to treat the output not as a binary number, but as a number to another base (256 is commonly used as a base, implying that each digit is a byte). This requires fewer renormalizations and no bitwise operations. The following analysis may shed light on this method.

At any point during arithmetic coding, the output consists of four parts as follows:

1. The part already written on the output. This part will not change.
2. One digit (bit, byte, or a digit to another base) that may be modified by at most one carry when adding to the lower end of the interval. (There cannot be two carries

because when this digit was originally determined, the range was less than or equal to one unit. Two carries require a range greater than one unit.)

3. A (possibly empty) block of digits that passes on a carry (1 in binary, 9 in decimal, 255 for base-256, etc.) and are represented by a counter counting their number.

4. The low variable of the encoder.

The following states can occur while data is encoded:

- No renormalization is needed because the range is in the desired interval.
- The low end plus the range (this is the upper end of the interval) will not produce any carry. In this case the second and third parts can be output because they will never change.
- The digit produced will become part two, and part three will be empty. The low end has already produced a carry. In this case, the (modified) second and third parts can be output; there will not be another carry. Set the second and third part as before.
- The digit produced will pass on a possible future carry, so it is added to the block of digits of part three.

The difference between conventional integer-based arithmetic coding and range coding is that in the latter, part two, which may be modified by a carry, has to be stored explicitly. With binary output this part is always 0 since the 1's are always added to the carry-passing-block. Implementing that is straightforward.

More information and code can be found in [Campos 06]. Range coding is used in LZMA (Section 6.26).

## 5.11 The QM Coder

JPEG (Section 7.10) is an important image compression method. It uses arithmetic coding, but not in the way described in Section 5.9. The arithmetic coder of JPEG is called the QM-coder and is described in this section. It is designed for simplicity and speed, so it is limited to input symbols that are single bits and it uses an approximation instead of a multiplication. It also uses fixed-precision integer arithmetic, so it has to resort to *renormalization* of the probability interval from time to time, in order for the approximation to remain close to the true multiplication. For more information on this method, see [IBM 88], [Pennebaker and Mitchell 88a], and [Pennebaker and Mitchell 88b].

A slight confusion arises because the arithmetic coder of JPEG 2000 (Section 8.19) and JBIG2 (Section 7.15) is called the MQ-coder and is not the same as the QM-coder (we are indebted to Christopher M. Brislawn for pointing this out).

- ◇ **Exercise 5.28:** The QM-coder is limited to input symbols that are single bits. Suggest a way to convert an arbitrary set of symbols to a stream of bits.

The main idea behind the QM-coder is to classify each input symbol (which is a single bit) as either the more probable symbol (MPS) or the less probable symbol (LPS). Before the next bit is input, the QM-encoder uses a statistical model to determine

whether a 0 or a 1 is more probable at that point. It then inputs the next bit and classifies it according to its actual value. If the model predicts, for example, that a 0 is more probable, and the next bit turns out to be a 1, the encoder classifies it as an LPS. It is important to understand that the only information encoded in the compressed stream is whether the next bit is MPS or LPS. When the stream is decoded, all that the decoder knows is whether the bit that has just been decoded is an MPS or an LPS. The decoder has to use the same statistical model to determine the current relation between MPS/LPS and 0/1. This relation changes, of course, from bit to bit, since the model is updated identically (in lockstep) by the encoder and decoder each time a bit is input by the former or decoded by the latter.

The statistical model also computes a probability  $Qe$  for the LPS, so the probability of the MPS is  $(1 - Qe)$ . Since  $Qe$  is the probability of the *less probable* symbol, it is in the range  $[0, 0.5]$ . The encoder divides the probability interval  $A$  into two subintervals according to  $Qe$  and places the LPS subinterval (whose size is  $A \times Qe$ ) above the MPS subinterval [whose size is  $A(1 - Qe)$ ], as shown in Figure 5.56b. Notice that the two subintervals in the figure are closed at the bottom and open at the top. This should be compared with the way a conventional arithmetic encoder divides the same interval (Figure 5.56a, where the numbers are taken from Table 5.43).

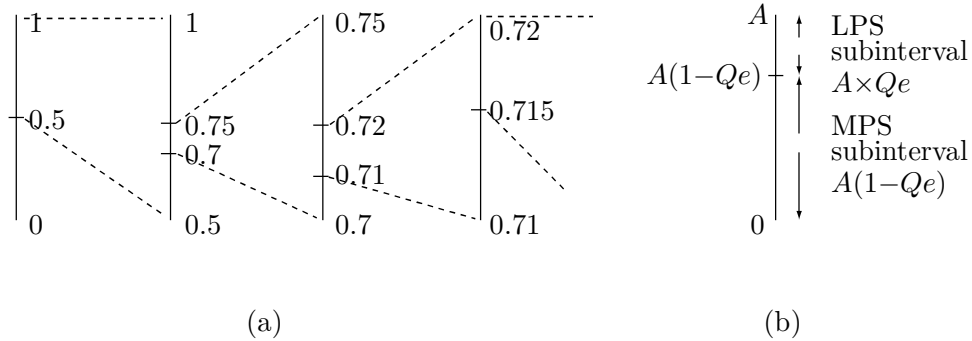


Figure 5.56: Division of the Probability Interval.

In conventional arithmetic coding, the interval is narrowed all the time, and the final output is any number inside the final subinterval. In the QM-coder, for simplicity, each step adds the bottom of the selected subinterval to the output-so-far. We denote the output string by  $C$ . If the current bit read from the input is the MPS, the bottom of the MPS subinterval (i.e., the number 0) is added to  $C$ . If the current bit is the LPS, the bottom of the LPS subinterval [i.e., the number  $A(1 - Qe)$ ] is added to  $C$ . After  $C$  is updated in this way, the current probability interval  $A$  is shrunk to the size of the selected subinterval. The probability interval is always in the range  $[0, A]$ , and  $A$  gets smaller at each step. This is the main principle of the QM-encoder, and it is expressed by the rules

$$\begin{aligned} \text{After MPS: } & C \text{ is unchanged, } A \leftarrow A(1 - Qe), \\ \text{After LPS: } & C \leftarrow C + A(1 - Qe), \quad A \leftarrow A \times Qe. \end{aligned} \quad (5.2)$$

These rules set  $C$  to point to the bottom of the MPS or the LPS subinterval, depending on the classification of the current input bit. They also set  $A$  to the new size of the subinterval.

Table 5.57 lists the values of  $A$  and  $C$  when four symbols, each a single bit, are encoded. We assume that they alternate between an LPS and an MPS and that  $Qe = 0.5$  for all four steps (normally, of course, the statistical model yields different values of  $Qe$  all the time). It is easy to see how the probability interval  $A$  shrinks from 1 to 0.0625, and how the output  $C$  grows from 0 to 0.625. Table 5.59 is similar, but uses  $Qe = 0.1$  for all four steps. Again  $A$  shrinks, to 0.0081, and  $C$  grows, to 0.981. Figures 5.58 and 5.60 illustrate graphically the division of the probability interval  $A$  into an LPS and an MPS.

- ◇ **Exercise 5.29:** Repeat these calculations for the case where all four symbols are LPS and  $Qe = 0.5$ , then for the case where they are MPS and  $Qe = 0.1$ .

The principle of the QM-encoder is simple and easy to understand, but it involves two problems. The first is the fact that the interval  $A$ , which starts at 1, shrinks all the time and requires high precision to distinguish it from zero. The solution to this problem is to maintain  $A$  as an integer and double it every time it gets too small. This is called *renormalization*. It is fast, since it is done by a logical left shift; no multiplication is needed. Each time  $A$  is doubled,  $C$  is also doubled. The second problem is the multiplication  $A \times Qe$  used in subdividing the probability interval  $A$ . A fast compression method should avoid multiplications and divisions and should try to replace them with additions, subtractions, and shifts. It turns out that the second problem is also solved by renormalization. The idea is to keep the value of  $A$  close to 1, so that  $Qe$  will not be very different from the product  $A \times Qe$ . The multiplication is *approximated* by  $Qe$ .

How can we use renormalization to keep  $A$  close to 1? The first idea that comes to mind is to double  $A$  when it gets just a little below 1, say to 0.9. The problem is that doubling 0.9 yields 1.8, closer to 2 than to 1. If we let  $A$  get below 0.5 before doubling it, the result will be less than 1. It does not take long to realize that 0.75 is a good minimum value for renormalization. If  $A$  reaches this value at a certain step, it is doubled, to 1.5. If it reaches a smaller value, such as 0.6 or 0.55, it ends up even closer to 1 when doubled.

If  $A$  reaches a value less than 0.5 at a certain step, it has to be renormalized by doubling it several times, each time also doubling  $C$ . An example is the second row of Table 5.59, where  $A$  shrinks from 1 to 0.1 in one step, because of a very small probability  $Qe$ . In this case,  $A$  has to be doubled three times, from 0.1 to 0.2, to 0.4, to 0.8, in order to bring it into the desired range [0.75, 1.5). We conclude that  $A$  can go down to 0 (or very close to 0) and can be at most 1.5 (actually, less than 1.5, since our intervals are always open at the high end).

- ◇ **Exercise 5.30:** In what case does  $A$  always have to be renormalized?

Approximating the multiplication  $A \times Qe$  by  $Qe$  changes the main rules of the QM-encoder to

After MPS:  $C$  is unchanged,  $A \leftarrow A(1 - Qe) \approx A - Qe$ ,

After LPS:  $C \leftarrow C + A(1 - Qe) \approx C + A - Qe$ ,  $A \leftarrow A \times Qe \approx Qe$ .



Symbol	$C$	$A$
Initially	0	1
s1 (LPS)	$0 + 1(1 - 0.5) = 0.5$	$1 \times 0.5 = 0.5$
s2 (MPS)	unchanged	$0.5 \times (1 - 0.5) = 0.25$
s3 (LPS)	$0.5 + 0.25(1 - 0.5) = 0.625$	$0.25 \times 0.5 = 0.125$
s4 (MPS)	unchanged	$0.125 \times (1 - 0.5) = 0.0625$

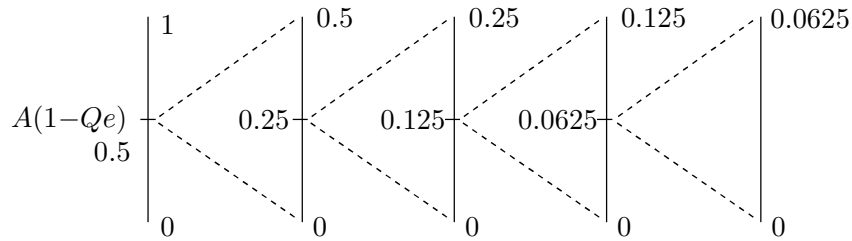
Table 5.57: Encoding Four Symbols With  $Qe = 0.5$ .

Figure 5.58: Division of the Probability Interval.

Symbol	$C$	$A$
Initially	0	1
s1 (LPS)	$0 + 1(1 - 0.1) = 0.9$	$1 \times 0.1 = 0.1$
s2 (MPS)	unchanged	$0.1 \times (1 - 0.1) = 0.09$
s3 (LPS)	$0.9 + 0.09(1 - 0.1) = 0.981$	$0.09 \times 0.1 = 0.009$
s4 (MPS)	unchanged	$0.009 \times (1 - 0.1) = 0.0081$

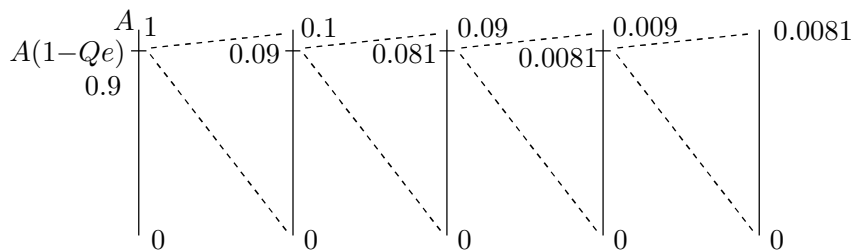
Table 5.59: Encoding Four Symbols With  $Qe = 0.1$ .

Figure 5.60: Division of the Probability Interval.

In order to include renormalization in these rules, we have to choose an integer representation for  $A$  where real values in the range  $[0, 1.5)$  are represented as integers. Since many current and old computers have 16-bit words, it makes sense to choose a representation where 0 is represented by a word of 16 zero bits and 1.5 is represented by the smallest 17-bit number, which is

$$2^{16} = 65536_{10} = 10000_{16} = 1 \underbrace{0 \dots 0}_{16} 2.$$

This way we can represent 65536 real values in the range  $[0, 1.5)$  as 16-bit integers, where the largest 16-bit integer, 65535, represents a real value slightly less than 1.5. Here are a few important examples of such values:

$$\begin{aligned} 0.75 &= 1.5/2 = 2^{15} = 32768_{10} = 8000_{16}, & 1 &= 0.75(4/3) = 43690_{10} = AAAA_{16}, \\ 0.5 &= 43690/2 = 21845_{10} = 5555_{16}, & 0.25 &= 21845/2 = 10923_{10} = 2AAB_{16}. \end{aligned}$$

(The optimal value of 1 in this representation is  $AAAA_{16}$ , but the way we associate the real values of  $A$  with the 16-bit integers is somewhat arbitrary. The important thing about this representation is to achieve accurate interval subdivision, and the subdivision is done by either  $A \leftarrow A - Qe$  or  $A \leftarrow Qe$ . The accuracy of the subdivision depends, therefore, on the relative values of  $A$  and  $Qe$ , and it has been found experimentally that the average value of  $A$  is  $B55A_{16}$ , so this value, instead of  $AAAA_{16}$ , is associated in the JPEG QM-coder with  $A = 1$ . The difference between the two values  $AAAA$  and  $B55A$  is  $AB0_{16} = 2736_{10}$ . The JBIG QM-coder uses a slightly different value for 1.)

Renormalization can now be included in the main rules of the QM-encoder, which become

$$\begin{aligned} \text{After MPS: } C \text{ is unchanged, } A &\leftarrow A - Qe, \\ &\text{if } A < 8000_{16} \text{ renormalize } A \text{ and } C. \\ \text{After LPS: } C &\leftarrow C + A - Qe, \quad A \leftarrow Qe, \\ &\text{renormalize } A \text{ and } C. \end{aligned} \tag{5.3}$$

Tables 5.61 and 5.62 list the results of applying these rules to the examples shown in Tables 5.57 and 5.59, respectively.

- ◇ **Exercise 5.31:** Repeat these calculations with renormalization for the case where all four symbols are LPS and  $Qe = 0.5$ . Following this, repeat the calculations for the case where they are all MPS and  $Qe = 0.1$ . (Compare this exercise with Exercise 5.29.)

The next point that has to be considered in the design of the QM-encoder is the problem of *interval inversion*. This is the case where the size of the subinterval allocated to the MPS becomes smaller than the LPS subinterval. This problem may occur when  $Qe$  is close to 0.5 and is a result of the approximation to the multiplication. It is illustrated in Table 5.63, where four MPS symbols are encoded with  $Qe = 0.45$ . In the third row of the table the interval  $A$  is doubled from 0.65 to 1.3. In the fourth row it is reduced to 0.85. This value is greater than 0.75, so no renormalization takes place; yet the subinterval allocated to the MPS becomes  $A - Qe = 0.85 - 0.45 = 0.40$ , which is smaller than the LPS subinterval, which is  $Qe = 0.45$ . Clearly, the problem occurs when  $Qe > A/2$ , a relation that can also be expressed as  $Qe > A - Qe$ .

Symbol	$C$	$A$	Renor. A	Renor. C
Initially	0	1		
s1 (LPS)	$0 + 1 - 0.5 = 0.5$	0.5	1	1
s2 (MPS)	unchanged	$1 - 0.5 = 0.5$	1	2
s3 (LPS)	$2 + 1 - 0.5 = 2.5$	0.5	1	5
s4 (MPS)	unchanged	$1 - 0.5 = 0.5$	1	10

Table 5.61: Renormalization Added to Table 5.57.

Symbol	$C$	$A$	Renor. A	Renor. C
Initially	0	1		
s1 (LPS)	$0 + 1 - 0.1 = 0.9$	0.1	0.8	$0.9 \cdot 2^3 = 7.2$
s2 (MPS)	unchanged 7.2	$0.8 - 0.1 = 0.7$	1.4	$7.2 \cdot 2 = 14.4$
s3 (LPS)	$14.4 + 1.4 - 0.1 = 15.7$	0.1	0.8	$15.7 \cdot 2^3 = 125.6$
s4 (MPS)	unchanged	$0.8 - 0.1 = 0.7$	1.4	$125.6 \cdot 2 = 251.2$

Table 5.62: Renormalization Added to Table 5.59.

Symbol	$C$	$A$	Renor. A	Renor. C
Initially	0	1		
s1 (MPS)	0	$1 - 0.45 = 0.55$	1.1	0
s2 (MPS)	0	$1.1 - 0.45 = 0.65$	1.3	0
s3 (MPS)	0	$1.3 - 0.45 = 0.85$		
s4 (MPS)	0	$0.85 - 0.45 = 0.40$	0.8	0

Table 5.63: Illustrating Interval Inversion.

The solution is to interchange the two subintervals whenever the LPS subinterval becomes greater than the MPS subinterval. This is called *conditional exchange*. The condition for interval inversion is  $Qe > A - Qe$ , but since  $Qe \leq 0.5$ , we get  $A - Qe < Qe \leq 0.5$ , and it becomes obvious that both  $Qe$  and  $A - Qe$  (i.e., both the LPS and MPS subintervals) are less than 0.75, so renormalization must take place. This is why the test for conditional exchange is performed only *after* the encoder has decided that renormalization is needed. The new rules for the QM-encoder are shown in Figure 5.64.

**The QM-Decoder:** The QM-decoder is the reverse of the QM-encoder. For simplicity we ignore renormalization and conditional exchange, and we assume that the QM-encoder operates by the rules of Equation (5.2). Reversing the way  $C$  is updated in those rules yields the rules for the QM-decoder (the interval  $A$  is updated in the same way):

$$\begin{aligned}
 \text{After MPS: } & C \text{ is unchanged, } A \leftarrow A(1 - Qe), \\
 \text{After LPS: } & C \leftarrow C - A(1 - Qe), \quad A \leftarrow A \times Qe.
 \end{aligned} \tag{5.4}$$

These rules are demonstrated using the data of Table 5.57. The four decoding steps are as follows:

After MPS:

```

C is unchanged
A ← A − Qe;           % The MPS subinterval
if A < 800016 then % if renormalization needed
  if A < Qe then      % if inversion needed
    C ← C + A;        % point to bottom of LPS
    A ← Qe            % Set A to LPS subinterval
  endif;
renormalize A and C;
endif;

```

After LPS:

```

A ← A − Qe;           % The MPS subinterval
if A ≥ Qe then % if interval sizes not inverted
  C ← C + A;          % point to bottom of LPS
  A ← Qe              % Set A to LPS subinterval
endif;
renormalize A and C;

```

Figure 5.64: QM-Encoder Rules With Interval Inversion.

*Step 1:*  $C = 0.625$ ,  $A = 1$ , the dividing line is  $A(1 - Qe) = 1(1 - 0.5) = 0.5$ , so the LPS and MPS subintervals are  $[0, 0.5)$  and  $[0.5, 1)$ . Since  $C$  points to the upper subinterval, an LPS is decoded. The new  $C$  is  $0.625 - 1(1 - 0.5) = 0.125$  and the new  $A$  is  $1 \times 0.5 = 0.5$ .

*Step 2:*  $C = 0.125$ ,  $A = 0.5$ , the dividing line is  $A(1 - Qe) = 0.5(1 - 0.5) = 0.25$ , so the LPS and MPS subintervals are  $[0, 0.25)$  and  $[0.25, 0.5)$ , and an MPS is decoded.  $C$  is unchanged, and the new  $A$  is  $0.5(1 - 0.5) = 0.25$ .

*Step 3:*  $C = 0.125$ ,  $A = 0.25$ , the dividing line is  $A(1 - Qe) = 0.25(1 - 0.5) = 0.125$ , so the LPS and MPS subintervals are  $[0, 0.125)$  and  $[0.125, 0.25)$ , and an LPS is decoded. The new  $C$  is  $0.125 - 0.25(1 - 0.5) = 0$ , and the new  $A$  is  $0.25 \times 0.5 = 0.125$ .

*Step 4:*  $C = 0$ ,  $A = 0.125$ , the dividing line is  $A(1 - Qe) = 0.125(1 - 0.5) = 0.0625$ , so the LPS and MPS subintervals are  $[0, 0.0625)$  and  $[0.0625, 0.125)$ , and an MPS is decoded.  $C$  is unchanged, and the new  $A$  is  $0.125(1 - 0.5) = 0.0625$ .

- ◇ **Exercise 5.32:** Use the rules of Equation (5.4) to decode the four symbols encoded in Table 5.59.

**Probability Estimation:** The QM-encoder uses a novel, interesting, and little-understood method for estimating the probability  $Qe$  of the LPS. The first method that comes to mind in trying to estimate the probability of the next input bit is to initialize  $Qe$  to 0.5 and update it by counting the numbers of zeros and ones that have been input so far. If, for example, 1000 bits have been input so far, and 700 of them were zeros, then 0 is the current MPS, with probability 0.7, and the probability of the LPS is  $Qe = 0.3$ . Notice that  $Qe$  should be updated *before* the next input bit is read and encoded, since otherwise the decoder would not be able to mirror this operation (the decoder does not know what the next bit is). This method produces good results, but is

slow, since  $Qe$  should be updated often (ideally, for each input bit), and the calculation involves a division (dividing 700/1000 in our example).

The method used by the QM-encoder is based on a table of preset  $Qe$  values.  $Qe$  is initialized to 0.5 and is modified when renormalization takes place, not for every input bit. Table 5.65 illustrates the process. The  $Qe$  index is initialized to zero, so the first value of  $Qe$  is  $0AC1_{16}$  or very close to 0.5. After the first MPS renormalization, the  $Qe$  index is incremented by 1, as indicated by column “Incr MPS.” A  $Qe$  index of 1 implies a  $Qe$  value of  $0A81_{16}$  or 0.49237, slightly smaller than the original, reflecting the fact that the renormalization occurred because of an MPS. If, for example, the current  $Qe$  index is 26, and the next renormalization is LPS, the index is decremented by 3, as indicated by column “Decr LPS,” reducing  $Qe$  to 0.00421. The method is not applied very often, and it involves only table lookups and incrementing or decrementing the  $Qe$  index: fast, simple operations.

$Qe$ index	Hex $Qe$	Dec $Qe$	Decr LPS	Incr MPS	MPS exch	$Qe$ index	Hex $Qe$	Dec $Qe$	Decr LPS	Incr MPS	MPS exch
0	0AC1	0.50409	0	1	1	15	0181	0.07050	2	1	0
1	0A81	0.49237	1	1	0	16	0121	0.05295	2	1	0
2	0A01	0.46893	1	1	0	17	00E1	0.04120	2	1	0
3	0901	0.42206	1	1	0	18	00A1	0.02948	2	1	0
4	0701	0.32831	1	1	0	19	0071	0.02069	2	1	0
5	0681	0.30487	1	1	0	20	0059	0.01630	2	1	0
6	0601	0.28143	1	1	0	21	0053	0.01520	2	1	0
7	0501	0.23456	2	1	0	22	0027	0.00714	2	1	0
8	0481	0.21112	2	1	0	23	0017	0.00421	2	1	0
9	0441	0.19940	2	1	0	24	0013	0.00348	3	1	0
10	0381	0.16425	2	1	0	25	000B	0.00201	2	1	0
11	0301	0.14081	2	1	0	26	0007	0.00128	3	1	0
12	02C1	0.12909	2	1	0	27	0005	0.00092	2	1	0
13	0281	0.11737	2	1	0	28	0003	0.00055	3	1	0
14	0241	0.10565	2	1	0	29	0001	0.00018	2	0	0

Table 5.65: Probability Estimation Table (Illustrative).

The column labeled “MPS exch” in Table 5.65 contains the information for the conditional exchange of the MPS and LPS definitions at  $Qe = 0.5$ . The zero value at the bottom of column “Incr MPS” should also be noted. If the  $Qe$  index is 29 and an MPS renormalization occurs, this zero causes the index to stay at 29 (corresponding to the smallest  $Qe$  value).

Table 5.65 is used here for illustrative purposes only. The JPEG QM-encoder uses Table 5.66, which has the same format but is harder to understand, since its  $Qe$  values are not listed in sorted order. This table was prepared using probability estimation concepts based on Bayesian statistics.

We now justify this probability estimation method with an approximate calculation that suggests that the  $Qe$  values obtained by this method will adapt to and closely approach the correct LPS probability of the binary input stream. The method updates

$Qe$  each time a renormalization occurs, and we know, from Equation (5.3), that this happens every time an LPS is input, but not for all MPS values. We therefore imagine an ideal balanced input stream where for each LPS bit there is a sequence of consecutive MPS bits. We denote the true (but unknown) LPS probability by  $q$ , and we try to show that the  $Qe$  values produced by the method for this ideal case are close to  $q$ .

Equation (5.3) lists the main rules of the QM-encoder and shows how the probability interval  $A$  is decremented by  $Qe$  each time an MPS is input and encoded. Imagine a renormalization that brings  $A$  to a value  $A_1$  (between 1 and 1.5), followed by a sequence of  $N$  consecutive MPS bits that reduce  $A$  in steps of  $Qe$  from  $A_1$  to a value  $A_2$  that requires another renormalization (i.e.,  $A_2$  is less than 0.75). It is clear that

$$N = \left\lfloor \frac{\Delta A}{Qe} \right\rfloor,$$

where  $\Delta A = A_1 - A_2$ . Since  $q$  is the true probability of an LPS, the probability of having  $N$  MPS bits in a row is  $P = (1 - q)^N$ . This implies  $\ln P = N \ln(1 - q)$ , which, for a small  $q$ , can be approximated by

$$\ln P \approx N(-q) = -\frac{\Delta A}{Qe}q, \text{ or } P \approx \exp\left(-\frac{\Delta A}{Qe}q\right). \quad (5.5)$$

Since we are dealing with an ideal balanced input stream, we are interested in the value  $P = 0.5$ , because it implies equal numbers of LPS and MPS renormalizations. From  $P = 0.5$  we get  $\ln P = -\ln 2$ , which, when combined with Equation (5.5), yields

$$Qe = \frac{\Delta A}{\ln 2}q.$$

This is fortuitous because  $\ln 2 \approx 0.693$  and  $\Delta A$  is typically a little less than 0.75. We can say that for our ideal balanced input stream,  $Qe \approx q$ , providing one justification for our estimation method. Another justification is provided by the way  $P$  depends on  $Qe$  [shown in Equation (5.5)]. If  $Qe$  gets larger than  $q$ ,  $P$  also gets large, and the table tends to move to smaller  $Qe$  values. In the opposite case, the table tends to select larger  $Qe$  values.

$Q_e$ index	Hex $Q_e$	Next-Index LPS	MPS	MPS exch	$Q_e$ index	Hex $Q_e$	Next-Index LPS	MPS	MPS exch
0	5A1D	1	1	1	57	01A4	55	58	0
1	2586	14	2	0	58	0160	56	59	0
2	1114	16	3	0	59	0125	57	60	0
3	080B	18	4	0	60	00F6	58	61	0
4	03D8	20	5	0	61	00CB	59	62	0
5	01DA	23	6	0	62	00AB	61	63	0
6	00E5	25	7	0	63	008F	61	32	0
7	006F	28	8	0	64	5B12	65	65	1
8	0036	30	9	0	65	4D04	80	66	0
9	001A	33	10	0	66	412C	81	67	0
10	000D	35	11	0	67	37D8	82	68	0
11	0006	9	12	0	68	2FE8	83	69	0
12	0003	10	13	0	69	293C	84	70	0
13	0001	12	13	0	70	2379	86	71	0
14	5A7F	15	15	1	71	1EDF	87	72	0
15	3F25	36	16	0	72	1AA9	87	73	0
16	2CF2	38	17	0	73	174E	72	74	0
17	207C	39	18	0	74	1424	72	75	0
18	17B9	40	19	0	75	119C	74	76	0
19	1182	42	20	0	76	0F6B	74	77	0
20	0CEF	43	21	0	77	0D51	75	78	0
21	09A1	45	22	0	78	0BB6	77	79	0
22	072F	46	23	0	79	0A40	77	48	0
23	055C	48	24	0	80	5832	80	81	1
24	0406	49	25	0	81	4D1C	88	82	0
25	0303	51	26	0	82	438E	89	83	0
26	0240	52	27	0	83	3BDD	90	84	0
27	01B1	54	28	0	84	34EE	91	85	0
28	0144	56	29	0	85	2EAE	92	86	0
29	00F5	57	30	0	86	299A	93	87	0
30	00B7	59	31	0	87	2516	86	71	0
31	008A	60	32	0	88	5570	88	89	1
32	0068	62	33	0	89	4CA9	95	90	0
33	004E	63	34	0	90	44D9	96	91	0
34	003B	32	35	0	91	3E22	97	92	0
35	002C	33	9	0	92	3824	99	93	0
36	5AE1	37	37	1	93	32B4	99	94	0
37	484C	64	38	0	94	2E17	93	86	0
38	3A0D	65	39	0	95	56A8	95	96	1
39	2EF1	67	40	0	96	4F46	101	97	0
40	261F	68	41	0	97	47E5	102	98	0
41	1F33	69	42	0	98	41CF	103	99	0
42	19A8	70	43	0	99	3C3D	104	100	0
43	1518	72	44	0	100	375E	99	93	0
44	1177	73	45	0	101	5231	105	102	0
45	0E74	74	46	0	102	4C0F	106	103	0
46	0BFB	75	47	0	103	4639	107	104	0
47	09F8	77	48	0	104	415E	103	99	0
48	0861	78	49	0	105	5627	105	106	1
49	0706	79	50	0	106	50E7	108	107	0
50	05CD	48	51	0	107	4B85	109	103	0
51	04DE	50	52	0	108	5597	110	109	0
52	040F	50	53	0	109	504F	111	107	0
53	0363	51	54	0	110	5A10	110	111	1
54	02D4	52	55	0	111	5522	112	109	0
55	025C	53	56	0	112	59EB	112	111	1
56	01F8	54	57	0					

Table 5.66: The QM-Encoder Probability Estimation Table.

## 5.12 Text Compression

Before delving into the details of the next method, here is a general discussion of text compression. Most text compression methods are either statistical or dictionary based. The latter class breaks the text into fragments that are saved in a data structure called a dictionary. When a fragment of new text is found to be identical to one of the dictionary entries, a pointer to that entry is written on the compressed stream, to become the compression of the new fragment. The former class, on the other hand, consists of methods that develop statistical *models* of the text.

A common statistical method consists of a modeling stage followed by a coding stage. The model assigns probabilities to the input symbols, and the coding stage actually codes the symbols based on those probabilities. The model can be static or dynamic (adaptive). Most models are based on one of the following two approaches.

**Frequency:** The model assigns probabilities to the text symbols based on their frequencies of occurrence, such that commonly-occurring symbols are assigned short codes. A static model uses fixed probabilities, whereas a dynamic model modifies the probabilities “on the fly” while text is being input and compressed.

**Context:** The model considers the context of a symbol when assigning it a probability. Since the decoder does not have access to future text, both encoder and decoder must limit the context to past text, i.e., to symbols that have already been input and processed. In practice, the context of a symbol is the  $N$  symbols preceding it (where  $N$  is a parameter). We therefore say that a context-based text compression method uses the context of a symbol to *predict* it (i.e., to assign it a probability). Technically, such a method is said to use an “order- $N$ ” Markov model. The PPM method, Section 5.14, is an excellent example of a context-based compression method, although the concept of context can also be used to compress images.

Some modern context-based text compression methods perform a transformation on the input data and then apply a statistical model to assign probabilities to the transformed symbols. Good examples of such methods are the Burrows-Wheeler method, Section 11.1, also known as the Burrows-Wheeler transform, or *block sorting*; the technique of symbol ranking, Section 11.2; and the ACB method, Section 11.3, which uses an associative dictionary.

Reference [Bell et al. 90] is an excellent introduction to text compression. It also describes many important algorithms and approaches to this important problem.

## 5.13 The Hutter Prize

Among the different types of digital data, text is the smallest in the sense that text files are much smaller than audio, image, or video files. A typical 300–400-page book may contain about 250,000 words or about a million characters, so it fits in a 1 MB file. A raw (uncompressed)  $1,024 \times 1,024$  color image, on the other hand, contains one mega pixels and therefore becomes a 3 MB file (three bytes for the color components of each pixel). Video files are much bigger. This is why many workers in the compression field concentrate their efforts on video and image compression, but there is a group of “hard core” researchers who are determined to squeeze out the last bit of redundancy from



text and compress text all the way down to its entropy. In order to encourage this group of enthusiasts, scientists, and programmers, Marcus Hutter decided to offer the prize that now bears his name.

In August 2006, Hutter selected a specific 100-million-character text file with text from the English wikipedia, named it **enwik8**, and announced a prize with initial funding of 50,000 euros [Hutter 08]. Two other organizers of the prize are Matt Mahoney and Jim Bowery.

Specifically, the prize awards 500 euros for each 1% improvement in the compression of **enwik8**. The following is a quotation from [wikiHutter 08].

The goal of the Hutter Prize is to encourage research in artificial intelligence (AI). The organizers believe that text compression and AI are equivalent problems. Hutter proved that the optimal behavior of a goal seeking agent in an unknown but computable environment is to guess at each step that the environment is controlled by the shortest program consistent with all interaction so far. Unfortunately, there is no general solution because Kolmogorov complexity is not computable. Hutter proved that in the restricted case (called AIXItl) where the environment is restricted to time  $t$  and space  $l$ , a solution can be computed in time  $O(t \times 2^l)$ , which is still intractable.

The organizers further believe that compressing natural language text is a hard AI problem, equivalent to passing the Turing test. Thus, progress toward one goal represents progress toward the other. They argue that predicting which characters are most likely to occur next in a text sequence requires vast real-world knowledge. A text compressor must solve the same problem in order to assign the shortest codes to the most likely text sequences.

Anyone can participate in this competition. A competitor has to submit either an encoder and decoder or a compressed **enwik8** and a decoder (there are certain time and memory constraints on the decoder). The source code is not required and the compression algorithm does not have to be general; it may be optimized for **enwik8**. The total size of the compressed **enwik8** plus the decoder should not exceed 99% of the previous winning entry. For each step of 1% improvement, the lucky winner receives 500 euros.

Initially, the compression baseline was the impressive 18,324,887 bytes, representing a compression ratio of 0.183 (achieved by PAQ8F, Section 5.15).

So far, only Matt Mahoney and Alexander Ratushnyak, won prizes for improvements to the baseline, with Ratushnyak winning twice!

The awards are computed by the simple expression  $Z(L - S)/L$ , where  $Z$  is the total prize funding (starting at 50,000 euros and possibly increasing in the future),  $S$  is the new record (size of the encoder or size of the compressed file plus the decoder), and  $L$  is the previous record. The minimum award is 3% of  $Z$ .

## 5.14 PPM

The PPM method is a sophisticated, state of the art compression method originally developed by J. Cleary and I. Witten [Cleary and Witten 84], with extensions and an implementation by A. Moffat [Moffat 90]. The method is based on an encoder that maintains a statistical model of the text. The encoder inputs the next symbol  $S$ , assigns it a probability  $P$ , and sends  $S$  to an adaptive arithmetic encoder, to be encoded with probability  $P$ .

The simplest *statistical model* counts the number of times each symbol has occurred in the past and assigns the symbol a probability based on that. Assume that 1217 symbols have been input and encoded so far, and 34 of them were the letter  $q$ . If the next symbol is a  $q$ , it is assigned a probability of  $34/1217$  and its count is incremented by 1. The next time  $q$  is seen, it will be assigned a probability of  $35/t$ , where  $t$  is the total number of symbols input up to that point (not including the last  $q$ ).

The next model up is a *context-based* statistical model. The idea is to assign a probability to symbol  $S$  depending not just on the frequency of the symbol but on the contexts in which it has occurred so far. The letter  $h$ , for example, occurs in “typical” English text (Table Intro.1) with a probability of about 5%. On average, we expect to see an  $h$  about 5% of the time. However, if the current symbol is  $t$ , there is a high probability (about 30%) that the next symbol will be  $h$ , since the digram  $th$  is common in English. We say that the model of typical English **predicts** an  $h$  in such a case. If the next symbol is in fact  $h$ , it is assigned a large probability. In cases where an  $h$  is the second letter of an unlikely digram, say  $xh$ , the  $h$  is assigned a smaller probability. Notice that the word “predicts” is used here to mean “estimate the probability of.” A similar example is the letter  $u$ , which has a probability of about 2%. When a  $q$  is encountered, however, there is a probability of more than 99% that the next letter will be a  $u$ .

- ◇ **Exercise 5.33:** We know that in English, a  $q$  must be followed by a  $u$ . Why not just say that the probability of the digram  $qu$  is 100%?

A *static* context-based modeler always uses the same probabilities. It contains static tables with the probabilities of all the possible digrams (or trigrams) of the alphabet and uses the tables to assign a probability to the next symbol  $S$  depending on the symbol (or, in general, on the context)  $C$  preceding it. We can imagine  $S$  and  $C$  being used as indices for a row and a column of a static frequency table. The table itself can be constructed by accumulating digram or trigram frequencies from large quantities of text. Such a modeler is simple and produces good results on average, but has two problems. The first is that some input streams may be statistically very different from the data originally used to prepare the table. A static encoder may create considerable expansion in such a case. The second problem is zero probabilities.

What if after reading and analyzing huge amounts of English text, we still have never encountered the trigram  $qqz$ ? The cell corresponding to  $qqz$  in the trigram frequency table will contain zero. The arithmetic encoder, Sections 5.9 and 5.10, requires all symbols to have nonzero probabilities. Even if a different encoder, such as Huffman, is used, all the probabilities involved must be nonzero. (Recall that the Huffman method works by combining two low-probability symbols into one high-probability symbol. If

two zero-probability symbols are combined, the resulting symbol will have the same zero probability.) Another reason why a symbol must have nonzero probability is that its entropy (the smallest number of bits into which it can be encoded) depends on  $\log_2 P$ , which is undefined for  $P = 0$  (but gets very large when  $P \rightarrow 0$ ). This *zero-probability problem* faces any model, static or adaptive, that uses probabilities of occurrence of symbols to achieve compression. Two simple solutions are traditionally adopted for this problem, but neither has any theoretical justification.

1. After analyzing a large quantity of data and counting frequencies, go over the frequency table, looking for empty cells. Each empty cell is assigned a frequency count of 1, and the total count is also incremented by 1. This method pretends that every digram and trigram has been seen at least once.
2. Add 1 to the total count and divide this single 1 among all the empty cells. Each will get a count that's less than 1 and, as a result, a very small probability. This assigns a very small probability to anything that hasn't been seen in the training data used for the analysis.

An *adaptive* context-based modeler also maintains tables with the probabilities of all the possible digrams (or trigrams or even longer contexts) of the alphabet and uses the tables to assign a probability to the next symbol  $S$  depending on a few symbols immediately preceding it (its context  $C$ ). The tables are updated all the time as more data is being input, which adapts the probabilities to the particular data being compressed.

Such a model is slower and more complex than the static one but produces better compression, since it uses the correct probabilities even when the input has data with probabilities much different from the average.

A text that skews letter probabilities is called a *lipogram*. (Would a computer program without any `goto` statements be considered a lipogram?) The word comes from the Greek stem *λείπω* (lipo or leipo) meaning to miss, to lack, combined with the Greek *γράμμα* (gramma), meaning “letter” or “of letters.” Together they form *λιπογράμματος*. There are not many examples of literary works that are lipograms:

1. Perhaps the best-known lipogram in English is *Gadsby*, a full-length novel [Wright 39], by Ernest V. Wright, that does not contain any occurrences of the letter E.
2. *Alphabetical Africa* by Walter Abish (W. W. Norton, 1974) is a readable lipogram where the reader is supposed to discover the unusual writing style while reading. This style has to do with the initial letters of words. The book consists of 52 chapters. In the first, all words begin with **a**; in the second, words start with either **a** or **b**, etc., until, in Chapter 26, all letters are allowed at the start of a word. In the remaining 26 chapters, the letters are taken away one by one. Various readers have commented on how little or how much they have missed the word “**the**” and how they felt on finally seeing it (in Chapter 20).
3. The novel *La Disparition* is a 1969 French lipogram by Georges Perec that does not contain the letter E (this letter actually appears several times, outside the main text, in words that the publisher had to include, and these are all printed in red). *La Disparition* has been translated to English, where it is titled *A Void*, by Gilbert Adair. Perec also wrote a univocalic (text employing just one vowel) titled *Les Revenentes* employing only the vowel E. The title of the English translation (by Ian Monk) is *The Exeter Text, Jewels, Secrets, Sex*. (Perec also wrote a short history of lipograms; see [Motte 98].)

A Quotation from the Preface to *Gadsby*

People as a rule will not stop to realize what a task such an attempt actually is. As I wrote along, in long-hand at first, a whole army of little E's gathered around my desk, all eagerly expecting to be called upon. But gradually as they saw me writing on and on, without even noticing them, they grew uneasy; and, with excited whisperings among themselves, began hopping up and riding on my pen, looking down constantly for a chance to drop off into some word; for all the world like sea birds perched, watching for a passing fish! But when they saw that I had covered 138 pages of typewriter size paper, they slid off unto the floor, walking sadly away, arm in arm; but shouting back: "You certainly must have a hodge-podge of a yarn there without Us! Why, man! We are in every story ever written, *hundreds and thousands of times!* This is the first time we ever were shut out!"

—Ernest V. Wright

4. Gottlob Burmann, a German poet, created our next example of a lipogram. He wrote 130 poems, consisting of about 20,000 words, without the use of the letter R. It is also believed that during the last 17 years of his life, he even omitted this letter from his daily conversation.

5. A Portuguese lipogram is found in five stories written by Alonso Alcala y Herrera, a Portuguese writer, in 1641, each suppressing one vowel.

6. Other examples, in Spanish, are found in the writings of Francisco Navarrete y Ribera (1659), Fernando Jacinto de Zurita y Haro (1654), and Manuel Lorenzo de Lizarazu y Berbinzana (also 1654).

An order- $N$  adaptive context-based modeler reads the next symbol  $S$  from the input stream and considers the  $N$  symbols preceding  $S$  the current order- $N$  context  $C$  of  $S$ . The model then estimates the probability  $P$  that  $S$  appears in the input data following the particular context  $C$ . Theoretically, the larger  $N$ , the better the probability estimate (the *prediction*). To get an intuitive feeling, imagine the case  $N = 20,000$ . It is hard to imagine a situation where a group of 20,000 symbols in the input stream is followed by a symbol  $S$ , but another group of the same 20,000 symbols, found later in the same input stream, is followed by a different symbol. Thus,  $N = 20,000$  allows the model to predict the next symbol (to estimate its probability) with high accuracy. However, large values of  $N$  have three disadvantages:

1. If we encode a symbol based on the 20,000 symbols preceding it, how do we encode the first 20,000 symbols in the input stream? They may have to be written on the output stream as raw ASCII codes, thereby reducing the overall compression.
2. For large values of  $N$ , there may be too many possible contexts. If our symbols are the 7-bit ASCII codes, the alphabet size is  $2^7 = 128$  symbols. There are therefore  $128^2 = 16,384$  order-2 contexts,  $128^3 = 2,097,152$  order-3 contexts, and so on. The number of contexts grows exponentially, since it is  $128^N$  or, in general,  $A^N$ , where  $A$  is the alphabet size.

I pounded the keys so hard that night that the letter **e** flew off the part of the machine that hits the paper. Not wanting to waste the night, I went next door to a neighbor who, I knew, had an elaborate workshop in his cellar. He attempted to solder my **e** back, but when I started to work again, it flew off like a bumblebee. For the rest of the night I inserted each **e** by hand, and in the morning I took the last dollars from our savings account to buy a new typewriter. Nothing could be allowed to delay the arrival of my greatest triumph.

—Sloan Wilson, *What Shall We Wear to This Party*, (1976)

- ◇ **Exercise 5.34:** What is the number of order-2 and order-3 contexts for an alphabet of size  $2^8 = 256$ ?

- ◇ **Exercise 5.35:** What would be a practical example of a 16-symbol alphabet?

3. A very long context retains information about the nature of old data. Experience shows that large data files tend to feature topic-specific content. Such a file may contain different distributions of symbols in different parts (a good example is a history book, where one chapter may commonly use words such as “Greek,” “Athens,” and “Troy,” while the following chapter may use “Roman,” “empire,” and “legion”). Such data is termed *nonstationary*. Better compression can therefore be achieved if the model takes into account the “age” of data i.e, if it assigns less significance to information gathered from old data and more weight to fresh, recent data. Such an effect is achieved by a short context.

- ◇ **Exercise 5.36:** Show an example of a common binary file where different parts may have different bit distributions.

As a result, relatively short contexts, in the range of 2 to 10, are used in practice. Any practical algorithm requires a carefully designed data structure that provides fast search and easy update, while holding many thousands of symbols and strings (Section 5.14.5).

We now turn to the next point in the discussion. Assume a context-based encoder that uses order-3 contexts. Early in the compression process, the word **here** was seen several times, but the word **there** is now seen for the first time. Assume that the next symbol is the **r** of **there**. The encoder will not find any instances of the order-3 context **the** followed by **r** (the **r** has 0 probability in this context). The encoder may simply write **r** on the compressed stream as a literal, resulting in no compression, but we know that **r** was seen several times in the past following the order-2 context **he** (**r** has nonzero probability in this context). The PPM method takes advantage of this knowledge.

“uvulopalatopharangoplasty” is the name of a surgical procedure to correct sleep apnea. It is rumored to be the longest (English?) word without any **e**’s.

### 5.14.1 PPM Principles

The central idea of PPM is to use this knowledge. The PPM encoder switches to a shorter context when a longer one has resulted in 0 probability. Thus, PPM starts with an order- $N$  context. It searches its data structure for a previous occurrence of the current context  $C$  followed by the next symbol  $S$ . If it finds no such occurrence (i.e., if the probability of this particular  $C$  followed by this  $S$  is 0), it switches to order  $N - 1$  and tries the same thing. Let  $C'$  be the string consisting of the rightmost  $N - 1$  symbols of  $C$ . The PPM encoder searches its data structure for a previous occurrence of the current context  $C'$  followed by symbol  $S$ . PPM therefore tries to use smaller and smaller parts of the context  $C$ , which is the reason for its name. The name PPM stands for “prediction with partial string matching.” Here is the process in some detail.

The encoder reads the next symbol  $S$  from the input stream, looks at the current order- $N$  context  $C$  (the last  $N$  symbols read), and based on input data that has been seen in the past, determines the probability  $P$  that  $S$  will appear following the particular context  $C$ . The encoder then invokes an adaptive arithmetic coding algorithm to encode symbol  $S$  with probability  $P$ . In practice, the adaptive arithmetic encoder is a procedure that receives the quantities `HighCumFreq[X]` and `LowCumFreq[X]` (Section 5.10) as parameters from the PPM encoder.

As an example, suppose that the current order-3 context is the string `the`, which has already been seen 27 times in the past and was followed by the letters `r` (11 times), `s` (9 times), `n` (6 times), and `m` (just once). The encoder assigns these cases the probabilities 11/27, 9/27, 6/27, and 1/27, respectively. If the next symbol read is `r`, it is sent to the arithmetic encoder with a probability of 11/27, and the probabilities are updated to 12/28, 9/28, 6/28, and 1/28.

What if the next symbol read is `a`? The context `the` was never seen followed by an `a`, so the probability of this case is 0. This zero-probability problem is solved in PPM by switching to a shorter context. The PPM encoder asks; How many times was the order-2 context `he` seen in the past and by what symbols was it followed? The answer may be as follows: Seen 54 times, followed by `a` (26 times), by `r` (12 times), etc. The PPM encoder now sends the `a` to the arithmetic encoder with a probability of 26/54.

If the next symbol  $S$  was never seen before following the order-2 context `he`, the PPM encoder switches to order-1 context. Was  $S$  seen before following the string `e`? If yes, a nonzero probability is assigned to  $S$  depending on how many times it (and other symbols) was seen following `e`. Otherwise, the PPM encoder switches to order-0 context. It asks itself how many times symbol  $S$  was seen in the past, regardless of any contexts. If it was seen 87 times out of 574 symbols read, it is assigned a probability of 87/574. If the symbol  $S$  has never been seen before (a common situation at the start of any compression process), the PPM encoder switches to a mode called order  $-1$  context, where  $S$  is assigned the fixed probability  $1/(\text{size of the alphabet})$ .

To predict is one thing. To predict correctly is another.

—Unknown

Table 5.67 shows contexts and frequency counts for orders 4 through 0 after the 11-symbol string `xyzzxyxyzzx` has been input and encoded. To understand the operation of the PPM encoder, let's assume that the 12th symbol is `z`. The order-4 context is now `yzzx`, which earlier was seen followed by `y` but never by `z`. The encoder therefore switches to the order-3 context, which is `zzx`, but even this hasn't been seen earlier followed by `z`. The next lower context, `zx`, is of order 2, and it also fails. The encoder then switches to order 1, where it checks context `x`. Symbol `x` was found three times in the past but was always followed by `y`. Order 0 is checked next, where `z` has a frequency count of 4 (out of a total count of 11). Symbol `z` is therefore sent to the adaptive arithmetic encoder, to be encoded with probability  $4/11$  (the PPM encoder "predicts" that it will appear  $4/11$  of the time).

Order 4	Order 3	Order 2	Order 1	Order 0
xyzz→x 2	xyz→z 2	xy→z 2	x→y 3	x 4
yzzx→y 1	yzz→x 2	→x 1	y→z 2	y 3
zzxy→x 1	zzx→y 1	yz→z 2	→x 1	z 4
zxyx→y 1	zxy→x 1	zz→x 2	z→z 2	
xyxy→z 1	xyx→y 1	zx→y 1	→x 2	
yxyz→z 1	yxy→z 1	yx→y 1		

(a)

Order 4	Order 3	Order 2	Order 1	Order 0
xyzz→x 2	xyz→z 2	xy→z 2	x→y 3	x 4
yzzx→y 1	yzz→x 2	xy→x 1	→z 1	y 3
→z 1	zzx→y 1	yz→z 2	y→z 2	z 5
zzxy→x 1	→z 1	zz→x 2	→x 1	
zxyx→y 1	zxy→x 1	zx→y 1	z→z 2	
xyxy→z 1	xyx→y 1	→z 1	→x 2	
yxyz→z 1	yxy→z 1	yx→y 1		

(b)

Table 5.67: (a) Contexts and Counts for "xyzzxyxyzzx". (b) Updated After Another `z` Is Input.

Next, we consider the PPM decoder. There is a fundamental difference between the way the PPM encoder and decoder work. The encoder can always look at the next symbol and base its next step on what that symbol is. The job of the decoder is to find out what the next symbol is. The encoder decides to switch to a shorter context based on what the next symbol is. The decoder cannot mirror this, since it does not know what the next symbol is. The algorithm needs an additional feature that will make it possible for the decoder to stay in lockstep with the encoder. The feature used by PPM is to reserve one symbol of the alphabet as an *escape symbol*. When the encoder decides to switch to a shorter context, it first writes the escape symbol (arithmetically encoded) on the output stream. The decoder can decode the escape symbol, since it is encoded



in the present context. After decoding an escape, the decoder also switches to a shorter context.

The worst that can happen with an order- $N$  encoder is to encounter a symbol  $S$  for the first time (this happens mostly at the start of the compression process). The symbol hasn't been seen before in any context, not even in order-0 context (i.e., by itself). In such a case, the encoder ends up sending  $N + 1$  consecutive escapes to be arithmetically encoded and output, switching all the way down to order  $-1$ , followed by the symbol  $S$  encoded with the fixed probability  $1/(\text{size of the alphabet})$ . Since the escape symbol may be output many times by the encoder, it is important to assign it a reasonable probability. Initially, the escape probability should be high, but it should drop as more symbols are input and decoded and more information is collected by the modeler about contexts in the particular data being compressed.

- ◇ **Exercise 5.37:** The escape is just a symbol of the alphabet, reserved to indicate a context switch. What if the data uses every symbol in the alphabet and none can be reserved? A common example is image compression, where a pixel is represented by a byte (256 grayscales or colors). Since pixels can have any values between 0 and 255, what value can be reserved for the escape symbol in this case?

Table 5.68 shows one way of assigning probabilities to the escape symbol (this is variant PPMC of PPM). The table shows the contexts (up to order 2) collected while reading and encoding the 14-symbol string **assanissimassa**. (In the movie “8 1/2,” Italian children utter this string as a magic spell. They pronounce it **assa-neesee-massa**.) We assume that the alphabet consists of the 26 letters, the blank space, and the escape symbol, a total of 28 symbols. The probability of a symbol in order  $-1$  is therefore  $1/28$ . Notice that it takes 5 bits to encode 1 of 28 symbols without compression.

Each context seen in the past is placed in the table in a separate group together with the escape symbol. The order-2 context **as**, e.g., was seen twice in the past and was followed by **s** both times. It is assigned a frequency of 2 and is placed in a group together with the escape symbol, which is assigned frequency 1. The probabilities of **as** and the escape in this group are therefore  $2/3$  and  $1/3$ , respectively. Context **ss** was seen three times, twice followed by **a** and once by **i**. These two occurrences are assigned frequencies 2 and 1 and are placed in a group together with the escape, which is now assigned frequency 2 (because it is in a group of 2 members). The probabilities of the three members of this group are therefore  $2/5$ ,  $1/5$ , and  $2/5$ , respectively.

The justification for this method of assigning escape probabilities is the following: Suppose that context **abc** was seen ten times in the past and was always followed by **x**. This suggests that the same context will be followed by the same **x** in the future, so the encoder will only rarely have to switch down to a lower context. The escape symbol can therefore be assigned the small probability  $1/11$ . However, if every occurrence of context **abc** in the past was followed by a different symbol (suggesting that the data varies a lot), then there is a good chance that the next occurrence will also be followed by a different symbol, forcing the encoder to switch to a lower context (and thus to emit an escape) more often. The escape is therefore assigned the higher probability  $10/20$ .

- ◇ **Exercise 5.38:** Explain the numbers  $1/11$  and  $10/20$ .



Order 2			Order 1			Order 0		
Context	$f$	$p$	Context	$f$	$p$	Symbol	$f$	$p$
<b>as</b> → <b>s</b>	2	2/3	<b>a</b> → <b>s</b>	2	2/5	<b>a</b>	4	4/19
<b>esc</b>	1	1/3	<b>a</b> → <b>n</b>	1	1/5	<b>s</b>	6	6/19
			<b>esc</b> →	2	2/5	<b>n</b>	1	1/19
<b>ss</b> → <b>a</b>	2	2/5				<b>i</b>	2	2/19
<b>ss</b> → <b>i</b>	1	1/5	<b>s</b> → <b>s</b>	3	3/9	<b>m</b>	1	1/19
<b>esc</b>	2	2/5	<b>s</b> → <b>a</b>	2	2/9	<b>esc</b>	5	5/19
			<b>s</b> → <b>i</b>	1	1/9			
<b>sa</b> → <b>n</b>	1	1/2	<b>esc</b>	3	3/9			
<b>esc</b>	1	1/2						
			<b>n</b> → <b>i</b>	1	1/2			
<b>an</b> → <b>i</b>	1	1/2	<b>esc</b>	1	1/2			
<b>esc</b>	1	1/2						
			<b>i</b> → <b>s</b>	1	1/4			
<b>ni</b> → <b>s</b>	1	1/2	<b>i</b> → <b>m</b>	1	1/4			
<b>esc</b>	1	1/2	<b>esc</b>	2	2/4			
<b>is</b> → <b>s</b>	1	1/2	<b>m</b> → <b>a</b>	1	1/2			
<b>esc</b>	1	1/2	<b>esc</b>	1	1/2			
<b>si</b> → <b>m</b>	1	1/2						
<b>esc</b>	1	1/2						
<b>im</b> → <b>a</b>	1	1/2						
<b>esc</b>	1	1/2						
<b>ma</b> → <b>s</b>	1	1/2						
<b>esc</b>	1	1/2						

Table 5.68: Contexts, Counts ( $f$ ), and Probabilities ( $p$ ) for “as-sanissimassa”.

Order 0 consists of the five different symbols **asnim** seen in the input string, followed by an escape, which is assigned frequency 5. Thus, probabilities range from 4/19 (for **a**) to 5/19 (for the escape symbol).

Wall Street indexes predicted nine out of the last five recessions.  
—Paul A. Samuelson, *Newsweek* (19 September 1966)

### 5.14.2 Examples

We are now ready to look at actual examples of new symbols being read and encoded. We assume that the 14-symbol string **assanissimassa** has been completely input and encoded, so the current order-2 context is “**sa**”. Here are four typical cases:

1. The next symbol is **n**. The PPM encoder finds that **sa** followed by **n** has been seen before and has probability 1/2. The **n** is encoded by the arithmetic encoder with this

probability, which takes, since arithmetic encoding normally compresses at or close to the entropy,  $-\log_2(1/2) = 1$  bit.

2. The next symbol is **s**. The PPM encoder finds that **sa** was not seen before followed by an **s**. The encoder therefore sends the escape symbol to the arithmetic encoder, together with the probability  $(1/2)$  predicted by the order-2 context of **sa**. It therefore takes 1 bit to encode this escape. Switching down to order 1, the current context becomes **a**, and the PPM encoder finds that an **a** followed by an **s** was seen before and currently has probability  $2/5$  assigned. The **s** is then sent to the arithmetic encoder to be encoded with probability  $2/5$ , which produces another 1.32 bits. In total,  $1 + 1.32 = 2.32$  bits are generated to encode the **s**.

3. The next symbol is **m**. The PPM encoder finds that **sa** was never seen before followed by an **m**. It therefore sends the escape symbol to the arithmetic encoder, as in Case 2, generating 1 bit so far. It then switches to order 1, finds that **a** has never been seen followed by an **m**, so it sends another escape symbol, this time using the escape probability for the order-1 **a**, which is  $2/5$ . This is encoded in 1.32 bits. Switching to order 0, the PPM encoder finds **m**, which has probability  $1/19$  and sends it to be encoded in  $-\log_2(1/19) = 4.25$  bits. The total number of bits produced is thus  $1 + 1.32 + 4.25 = 6.57$ .

4. The next symbol is **d**. The PPM encoder switches from order 2 to order 1 to order 0, sending two escapes as in Case 3. Since **d** hasn't been seen before, it is not found in order 0, and the PPM encoder switches to order  $-1$  after sending a third escape with the escape probability of order 0, of  $5/19$  (this produces  $-\log_2(5/19) = 1.93$  bits). The **d** itself is sent to the arithmetic encoder with its order  $-1$  probability, which is  $1/28$ , so it gets encoded in 4.8 bits. The total number of bits necessary to encode this first **d** is  $1 + 1.32 + 1.93 + 4.8 = 9.05$ , more than the five bits that would have been necessary without any compression.

- ◇ **Exercise 5.39:** Suppose that Case 4 has actually occurred (i.e., the 15th symbol to be input was a **d**). Show the new state of the order-0 contexts.
- ◇ **Exercise 5.40:** Suppose that Case 4 has actually occurred and the 16th symbol is also a **d**. How many bits would it take to encode this second **d**?
- ◇ **Exercise 5.41:** Show how the results of the above four cases are affected if we assume an alphabet size of 256 symbols.

### 5.14.3 Exclusion

When switching down from order 2 to order 1, the PPM encoder can use the information found in order 2 in order to exclude certain order-1 cases that are now known to be impossible. This increases the order-1 probabilities and thereby improves compression. The same thing can be done when switching down from any order. Here are two detailed examples.

In Case 2, the next symbol is **s**. The PPM encoder finds that **sa** was seen before followed by **n** but not by **s**. The encoder sends an escape and switches to order 1. The current context becomes **a**, and the encoder checks to see whether an **a** followed by an **s** was seen before. The answer is yes (with frequency 2), but the fact that **sa** was seen

before followed by **n** implies that the current symbol cannot be **n** (if it were, it would be encoded in order 2).

The encoder can therefore *exclude* the case of an **a** followed by **n** in order-1 contexts [we can say that there is no need to reserve “room” (or “space”) for the probability of this case, since it is impossible]. This reduces the total frequency of the order-1 group “**a**→” from 5 to 4, which increases the probability assigned to **s** from 2/5 to 2/4. Based on our knowledge from order 2, the **s** can now be encoded in  $-\log_2(2/4) = 1$  bit instead of 1.32 (a total of two bits is produced, since the escape also requires 1 bit).

Another example is Case 4, modified for exclusions. When switching from order 2 to order 1, the probability of the escape is, as before, 1/2. When in order 1, the case of **a** followed by **n** is excluded, increasing the probability of the escape from 2/5 to 2/4. After switching to order 0, both **s** and **n** represent impossible cases and can be excluded. This leaves the order 0 with the four symbols **a**, **i**, **m**, and escape, with frequencies 4, 2, 1, and 5, respectively. The total frequency is 12, so the escape is assigned probability 5/12 (1.26 bits) instead of the original 5/19 (1.93 bits). This escape is sent to the arithmetic encoder, and the PPM encoder switches to order  $-1$ . Here it excludes all five symbols **asnim** that have already been seen in order 1 and are therefore impossible in order  $-1$ . The **d** can now be encoded with probability  $1/(28 - 5) \approx 0.043$  (4.52 bits instead of 4.8) or  $1/(256 - 5) \approx 0.004$  (7.97 bits instead of 8), depending on the alphabet size.

Exact and careful model building should embody constraints that the final answer had in any case to satisfy.

—Francis Crick, *What Mad Pursuit*, (1988)

#### 5.14.4 Four PPM Variants

The particular method described earlier for assigning escape probabilities is called PPMC. Four more methods, titled PPMA, PPMB, PPMP, and PPMX, have also been developed in attempts to assign precise escape probabilities in PPM. All five methods have been selected based on the vast experience that the developers had with data compression. The last two are based on Poisson distribution [Witten and Bell 91], which is the reason for the “P” in PPMP (the “X” comes from “approximate,” since PPMX is an approximate variant of PPMP).

Suppose that a group of contexts in Table 5.68 has total frequencies  $n$  (excluding the escape symbol). PPMA assigns the escape symbol a probability of  $1/(n + 1)$ . This is equivalent to always assigning it a count of 1. The other members of the group are still assigned their original probabilities of  $x/n$ , and these probabilities add up to 1 (not including the escape probability).

PPMB is similar to PPMC with one difference. It assigns a probability to symbol  $S$  following context  $C$  only after  $S$  has been seen **twice** in context  $C$ . This is done by subtracting 1 from the frequency counts. If, for example, context **abc** was seen three times, twice followed by **x** and once by **y**, then **x** is assigned probability  $(2 - 1)/3$ , and **y** (which should be assigned probability  $(1 - 1)/3 = 0$ ) is not assigned any probability (i.e., does not get included in Table 5.68 or its equivalent). Instead, the escape symbol “gets” the two counts subtracted from **x** and **y**, and it ends up being assigned probability 2/3. This method is based on the belief that “seeing twice is believing.”

PPMP is based on a different principle. It considers the appearance of each symbol a separate Poisson process. Suppose that there are  $q$  different symbols in the input stream. At a certain point during compression,  $n$  symbols have been read, and symbol  $i$  has been input  $c_i$  times (so  $\sum c_i = n$ ). Some of the  $c_i$ s are zero (this is the zero-probability problem). PPMP is based on the assumption that symbol  $i$  appears according to a Poisson distribution with an expected value (average)  $\lambda_i$ . The statistical problem considered by PPMP is to estimate  $q$  by extrapolating from the  $n$ -symbol sample input so far to the entire input stream of  $N$  symbols (or, in general, to a larger sample). If we express  $N$  in terms of  $n$  in the form  $N = (1 + \theta)n$ , then a lengthy analysis shows that the number of symbols that haven't appeared in our  $n$ -symbol sample is given by  $t_1\theta - t_2\theta^2 + t_3\theta^3 - \dots$ , where  $t_1$  is the number of symbols that appeared exactly once in our sample,  $t_2$  is the number of symbols that appeared twice, and so on.

Hapax legomena: words or forms that occur only once in the writings of a given language; such words are extremely difficult, if not impossible, to translate.

In the special case where  $N$  is not the entire input stream but the slightly larger sample of size  $n + 1$ , the expected number of new symbols is  $t_1 \frac{1}{n} - t_2 \frac{1}{n^2} + t_3 \frac{1}{n^3} - \dots$ . This expression becomes the probability that the next symbol is novel, so it is used in PPMP as the escape probability. Notice that when  $t_1$  happens to be zero, this expression is normally negative and cannot be used as a probability. Also, the case  $t_1 = n$  results in an escape probability of 1 and should be avoided. Both cases require corrections to the sum above.

PPMX uses the approximate value  $t_1/n$  (the first term of the sum) as the escape probability. This expression also breaks down when  $t_1$  happens to be 0 or  $n$ , so in these cases PPMX is modified to PPMXC, which uses the same escape probability as PPMC.

Experiments with all five variants show that the differences between them are small. Version X is indistinguishable from P, and both are slightly better than A-B-C. Version C is slightly but consistently better than A and B.

It should again be noted that the way escape probabilities are assigned in the A-B-C variants is based on experience and intuition, not on any underlying theory. Experience with these variants indicates that the basic PPM algorithm is robust and is not affected much by the precise way of computing escape probabilities. Variants P and X are based on theory, but even they don't significantly improve the performance of PPM.

### 5.14.5 Implementation Details

The main problem in any practical implementation of PPM is to maintain a data structure where all contexts (orders 0 through  $N$ ) of every symbol read from the input stream are stored and can be located fast. The structure described here is a special type of tree, called a *trie*. This is a tree in which the branching structure at any level is determined by just part of a data item, not by the entire item (page 357). In the case of PPM, an order- $N$  context is a string that includes all the shorter contexts of orders  $N - 1$  through 0, so each context effectively adds just one symbol to the trie.

Figure 5.69 shows how such a trie is constructed for the string “zxzyzxyzx” assuming  $N = 2$ . A quick glance shows that the tree grows in width but not in depth. Its depth remains  $N + 1 = 3$  regardless of how much input data has been read. Its width grows as more and more symbols are input, but not at a constant rate. Sometimes, no

new nodes are added, such as in case 10, when the last *x* is read. At other times, up to three nodes are added, such as in cases 3 and 4, when the second *z* and the first *y* are added.

Level 1 of the trie (just below the root) contains one node for each symbol read so far. These are the order-1 contexts. Level 2 contains all the order-2 contexts, and so on. Every context can be found by starting at the root and sliding down to one of the leaves. In case 3, for example, the two contexts are *xz* (symbol *z* preceded by the order-1 context *x*) and *zxz* (symbol *z* preceded by the order-2 context *zx*). In case 10, there are seven contexts ranging from *xxxy* and *xyz* on the left to *zxxz* and *zyz* on the right.

The numbers in the nodes are context counts. The “*z,4*” on the right branch of case 10 implies that *z* has been seen four times. The “*x,3*” and “*y,1*” below it mean that these four occurrences were followed by *x* three times and by *y* once. The circled nodes show the different orders of the context of the last symbol added to the trie. In case 3, for example, the second *z* has just been read and added to the trie. It was added twice, below the *x* of the left branch and the *x* of the right branch (the latter is indicated by the arrow). Also, the count of the original *z* has been incremented to 2. This shows that the new *z* follows the two contexts *x* (of order 1) and *zx* (order 2).

It should now be easy for the reader to follow the ten steps of constructing the tree and to understand intuitively how nodes are added and counts updated. Notice that three nodes (or, in general,  $N+1$  nodes, one at each level of the trie) are involved in each step (except the first few steps when the trie hasn’t reached its final height yet). Some of the three are new nodes added to the trie; the others have their counts incremented.

The next point that should be discussed is how the algorithm decides which nodes to update and which to add. To simplify the algorithm, one more pointer is added to each node, pointing backward to the node representing the next shorter context. A pointer that points backward in a tree is called a *vine pointer*.

Figure 5.70 shows the first ten steps in the construction of the PPM trie for the 14-symbol string “*assanissimassa*”. Each of the ten steps shows the new vine pointers (the dashed lines in the figure) constructed by the trie updating algorithm while that step was executed. Notice that old vine pointers are not deleted; they are just not shown in later diagrams. In general, a vine pointer points from a node *X* on level *n* to a node with the same symbol *X* on level *n* – 1. All nodes on level 1 point to the root.

A node in the PPM trie therefore consists of the following fields:

1. The code (ASCII or other) of the symbol.
2. The count.
3. A down pointer, pointing to the leftmost child of the node. In Figure 5.70, Case 10, for example, the leftmost son of the root is “*a,2*”. That of “*a,2*” is “*n,1*” and that of “*s,4*” is “*a,1*”.
4. A right pointer, pointing to the next sibling of the node. The root has no right sibling. The next sibling of node “*a,2*” is “*i,2*” and that of “*i,2*” is “*m,1*”.
5. A vine pointer. These are shown as dashed arrows in Figure 5.70.

- ◇ **Exercise 5.42:** Complete the construction of this trie and show it after all 14 characters have been input.

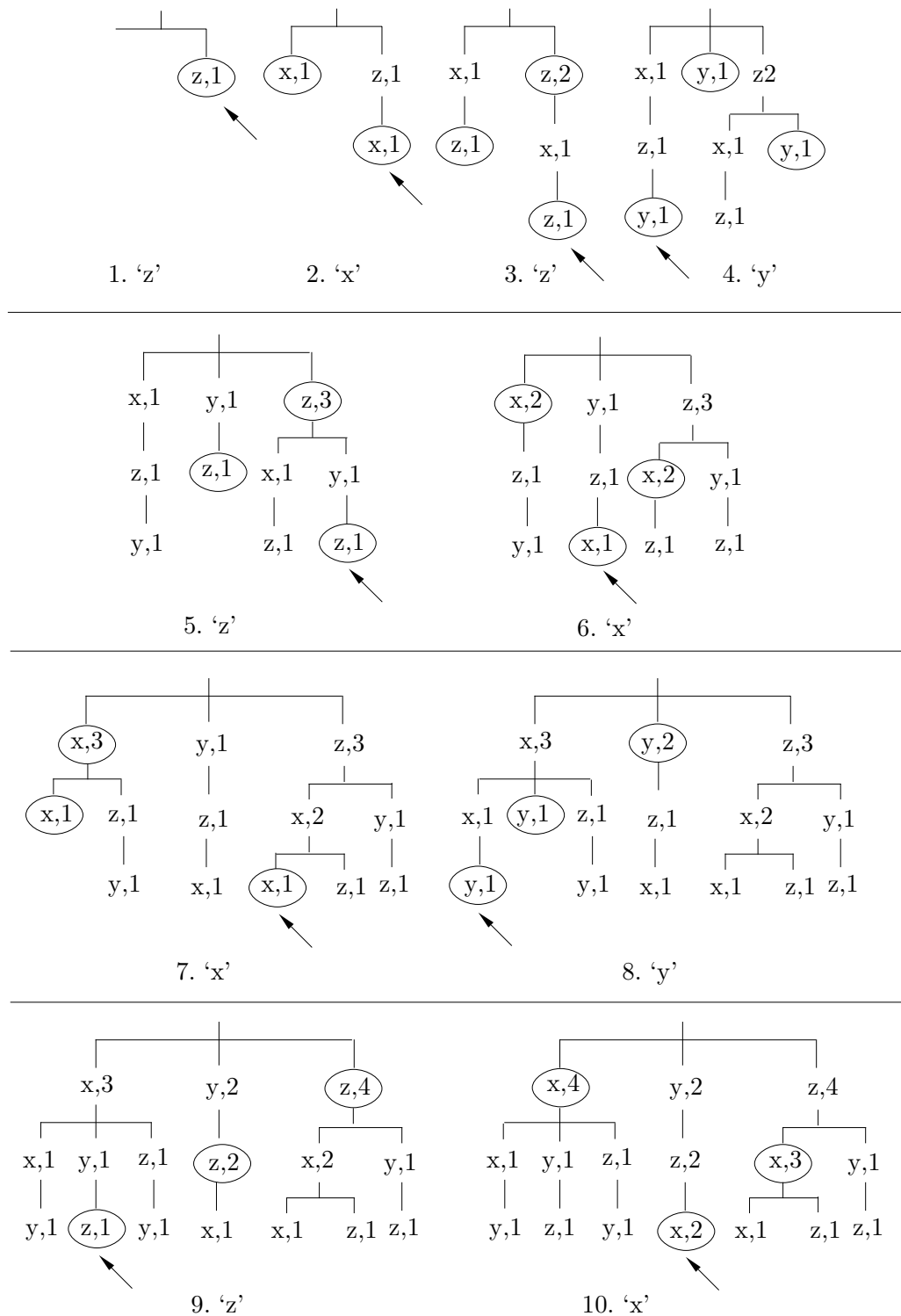


Figure 5.69: Ten Tries of "zxzyzxxxyzx".

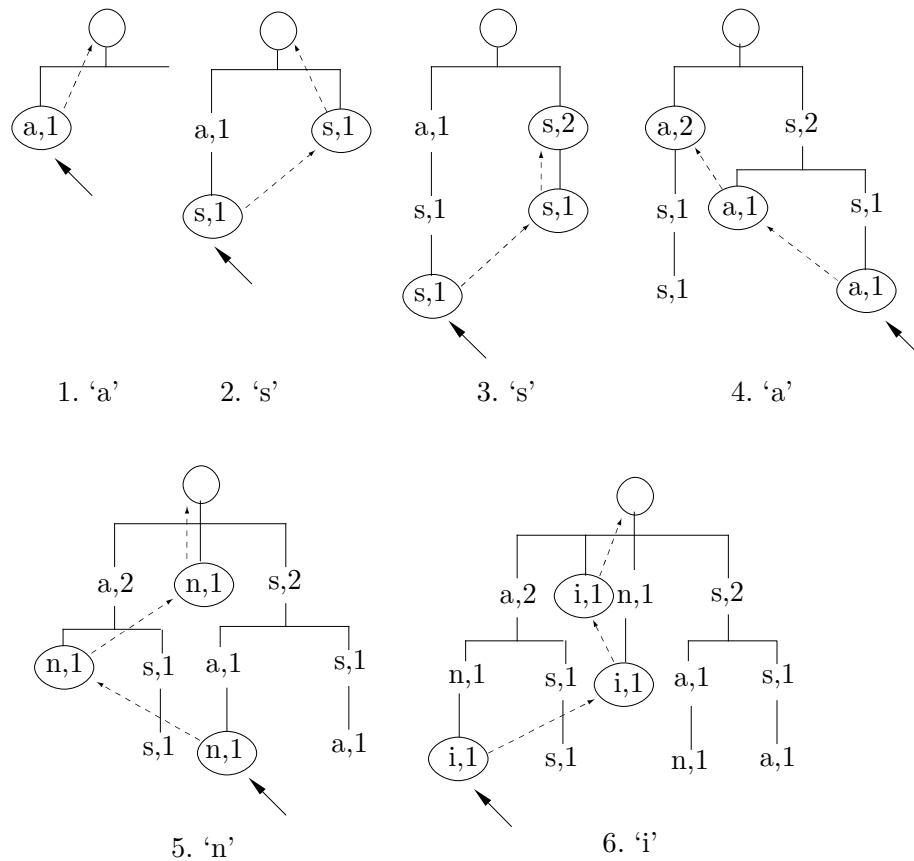


Figure 5.70: Part I. First Six Tries of "assanissimassa".

At any step during the trie construction, one pointer, called the *base*, is maintained that points to the last node added or updated in the previous step. This pointer is shown as a solid arrow in Figure 5.70. Suppose that symbol  $S$  has been input and the trie should be updated at this point. The algorithm for adding and/or updating nodes is as follows:

1. Follow the base pointer to node  $X$ . Follow the vine pointer from  $X$  to  $Y$  (notice that  $Y$  can be the root). Add  $S$  as a new child node of  $Y$  and set the base to point to it. However, if  $Y$  already has a child node with  $S$ , increment the count of that node by 1 (and also set the base to point to it). Call this node  $A$ .
2. Repeat the same step but without updating the base. Follow the vine pointer from  $Y$  to  $Z$ , add  $S$  as a new child node of  $Z$ , or update an existing child. Call this node  $B$ . If there is no vine pointer from  $A$  to  $B$ , install one. (If both  $A$  and  $B$  are old nodes, there will already be a vine pointer from  $A$  to  $B$ .)
3. Repeat until you have added (or incremented) a node at level 1.

During these steps, the PPM encoder also collects the counts that are needed to compute the probability of the new symbol  $S$ . Figure 5.70 shows the trie after the last

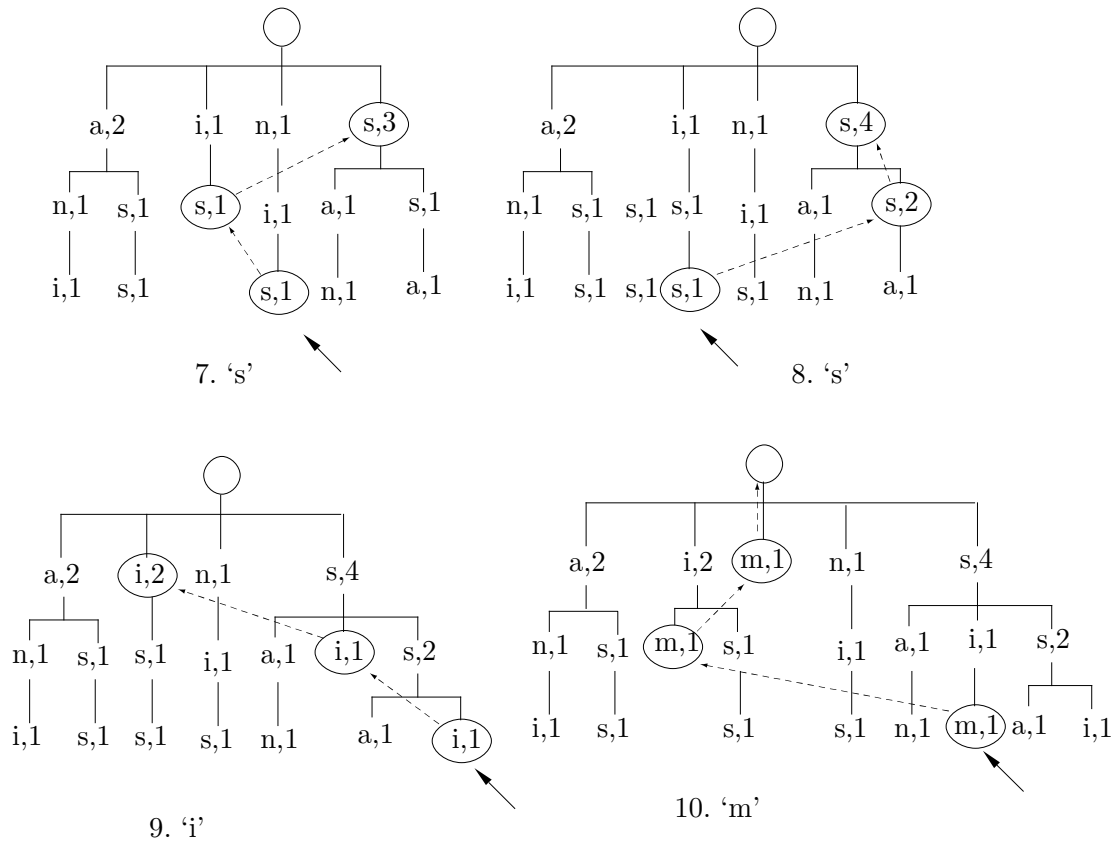


Figure 5.70: (Continued) Next Four Tries of "assanissimassa".

two symbols **s** and **a** were added. In Figure 5.70, Case 13, a vine pointer was followed from node "**s,2**", to node "**s,3**", which already had the two children "**a,1**" and "**i,1**". The first child was incremented to "**a,2**". In Figure 5.70, Case 14, the subtree with the three nodes "**s,3**", "**a,2**", and "**i,1**" tells the encoder that **a** was seen following context **ss** twice and **i** was seen following the same context once. Since the tree has two children, the escape symbol gets a count of 2, bringing the total count to 5. The probability of **a** is therefore  $2/5$  (compare with Table 5.68). Notice that steps 11 and 12 are not shown. The serious reader should draw the tries for these steps as a voluntary exercise (i.e., without an answer).

It is now easy to understand the reason why this particular trie is so useful. Each time a symbol is input, it takes the algorithm at most  $N + 1$  steps to update the trie and collect the necessary counts by going from the base pointer toward the root. Adding a symbol to the trie and encoding it takes  $O(N)$  steps regardless of the size of the trie. Since  $N$  is small (typically 4 or 5), an implementation can be made fast enough for practical use even if the trie is very large. If the user specifies that the algorithm should use exclusions, it becomes more complex, since it has to maintain, at each step, a list of symbols to be excluded.



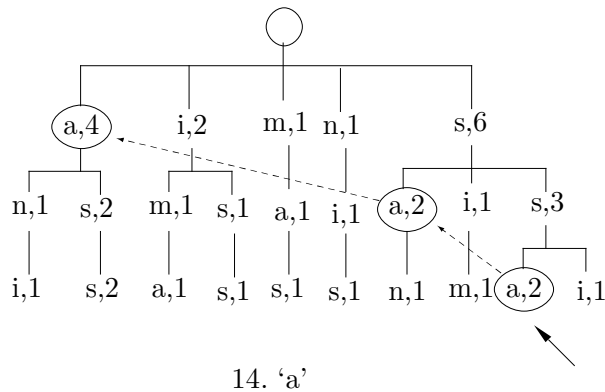
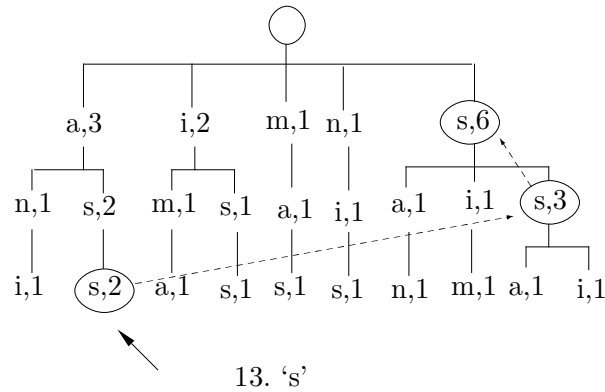


Figure 5.70: (Continued) Final Two Tries of “assanissimassa”.

As has been noted, between 0 and 3 nodes are added to the trie for each input symbol encoded (in general, between 0 and  $N + 1$  nodes). The trie can therefore grow very large and fill up any available memory space. One elegant solution, adopted in [Moffat 90], is to discard the trie when it gets full and start constructing a new one. In order to bring the new trie “up to speed” fast, the last 2048 input symbols are always saved in a circular buffer in memory and are used to construct the new trie. This reduces the amount of inefficient code generated when tries are replaced.

### 5.14.6 PPM\*

An important feature of the original PPM method is its use of a fixed-length, bounded initial context. The method selects a value  $N$  for the context length and always tries to predict (i.e., to assign probability to) the next symbol  $S$  by starting with an order- $N$  context  $C$ . If  $S$  hasn’t been seen so far in context  $C$ , PPM switches to a shorter context. Intuitively it seems that a long context (large value of  $N$ ) may result in better prediction, but Section 5.14 explains the drawbacks of long contexts. In practice, PPM implementations tend to use  $N$  values of 5 or 6 (Figure 5.71).

The PPM\* method, due to [Cleary et al. 95] and [Cleary and Teahan 97], tries to extend the value of  $N$  indefinitely. The developers tried to find ways to use unbounded

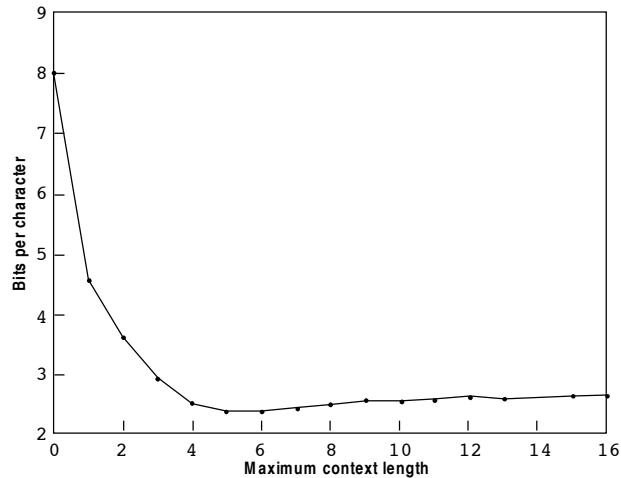


Figure 5.71: Compression Ratio as a Function of Maximum Context Length.

values for  $N$  in order to improve compression. The resulting method requires a new trie data structure and more computational resources than the original PPM, but in return it provides compression improvement of about 6% over PPMC.

(In mathematics, when a set  $S$  consists of symbols  $a_i$ , the notation  $S^*$  is used for the set of all the strings of symbols  $a_i$ .)

One problem with long contexts is the escape symbols. If the encoder inputs the next symbol  $S$ , starts with an order-100 context, and does not find any past string of 100 symbols that's followed by  $S$ , then it has to emit an escape and try an order-99 context. Such an algorithm may result in up to 100 consecutive escape symbols being emitted by the encoder, which can cause considerable expansion. It is therefore important to allow for contexts of various lengths, not only very long contexts, and decide on the length of a context depending on the current situation. The only restriction is that the decoder should be able to figure out the length of the context used by the encoder for each symbol encoded. This idea is the main principle behind the design of PPM\*.

An early idea was to maintain a record, for each context, of its past performance. Such a record can be mirrored by the decoder, so both encoder and decoder can use, at any point, that context that behaved best in the past. This idea did not seem to work and the developers of PPM\* were also faced with the task of having to explain why it did not work as expected.

The algorithm finally selected for PPM\* depends on the concept of a deterministic context. A context is defined as deterministic when it gives only one prediction. For example, the context `this_is_my_` is deterministic if every appearance of it so far in the input has been followed by the same symbol. Experiments indicate that if a context  $C$  is deterministic, the chance that when it is seen next time, it will be followed by a novel symbol is smaller than what is expected from a uniform prior distribution of the symbols. This feature suggests the use of deterministic contexts for prediction in the new version of PPM.

Based on experience with deterministic contexts, the developers have arrived at the following algorithm for PPM\*. When the next symbol  $S$  is input, search all its contexts

trying to find deterministic contexts of  $S$ . If any such contexts are found, use the shortest of them. If no deterministic contexts are found, use the longest nondeterministic context.

The result of this strategy for PPM\* is that nondeterministic contexts are used most of the time, and they are almost always 5–6 symbols long, the same as those used by traditional PPM. However, from time to time deterministic contexts are used and they get longer as more input is read and processed. (In experiments performed by the developers, deterministic contexts started at length 10 and became as long as 20–25 symbols after about 30,000 symbols were input.) The use of deterministic contexts results in very accurate prediction, which is the main contributor to the slightly better performance of PPM\* over PPMC.

A practical implementation of PPM\* has to solve the problem of keeping track of long contexts. Each time a symbol  $S$  is input, all its past occurrences have to be checked, together with all the contexts, short and long, deterministic or not, for each occurrence. In principle, this can be done by simply keeping the entire data file in memory and checking back for each symbol. Imagine the symbol in position  $i$  in the input file. It is preceded by  $i - 1$  symbols, so  $i - 1$  steps are needed to search and find all its contexts. The total number of steps for  $n$  symbols is therefore  $1 + 2 + \dots + (n - 1) = n(n - 1)/2$ . For large  $n$ , this amounts to  $O(n^2)$  complexity—too slow for practical implementations. This problem was solved by a special trie, termed a context-trie, where a leaf node points back to the input string whenever a context is unique. Each node corresponds to a symbol that follows some context and the frequency count of the symbol is stored in the node.

PPM\* uses the same escape mechanism as the original PPM. The implementation reported in the PPM publications uses the PPMC algorithm to assign probabilities to the various escape symbols. Notice that the original PPM uses escapes less and less over time, as more data is input and more context predictions become available. In contrast, PPM\* has to use escapes very often, regardless of the amount of data already input, particularly because of the use of deterministic contexts. This fact makes the problem of computing escape probabilities especially acute.

Compressing the entire (concatenated) Calgary corpus by PPM\* resulted in an average of 2.34 bpc, compared to 2.48 bpc achieved by PPMC. This represents compression improvement of about 6% because 2.34 is 94.4% of 2.48.

### 5.14.7 PPMZ

The PPMZ variant, originated and implemented by Charles Bloom [Bloom 98], is an attempt to improve the original PPM algorithm. It starts from the premise that PPM is a powerful algorithm that can, in principle, compress data to its entropy, especially when presented with large amounts of input data, but performs less than optimal in practice because of poor handling of features such as deterministic contexts, unbounded-length contexts, and local order estimation. PPMZ attempts to handle these features in an optimal way, and it ends up achieving superior performance.

The PPMZ algorithm starts, similar to PPM\*, by examining the maximum deterministic context of the current symbol. If no deterministic context is found, the PPMZ encoder executes a local-order-estimation (LOE) procedure, to compute an order in the interval  $[0, 12]$  and use it to predict the current symbol as the original PPM algorithm does. In addition, PPMZ uses a secondary model to predict the probabilities of the

various escape symbols.

The originator of the method noticed that the various PPM implementations compress data to about 2 bpc, where most characters are compressed to 1 bpc each, and the remaining characters represent either the start of the input stream or random data. The natural conclusion is that any small improvements in the probability estimation of the most common characters can lead to significant improvements in the overall performance of the method. We start by discussing the way PPMZ handles unbounded contexts.

Figure 5.72a shows a situation where the current character is **e** and its 12-order context is `ll_assume_th`. The context is hashed into a pointer *P* that points to a linked list. The nodes of the list point to all the 12-character strings in the input stream that happen to hash to the same pointer *P*. (Each node also has a count field indicating the number of times the string pointed to by the node has been a match.) The encoder follows the pointers, looking for a match whose minimum length varies from context to context. Assuming that the minimum match length in our case is 15, the encoder will find the 15-character match `e_all_assume_th` (the preceding **w** makes this a 16-character match, and it may even be longer). The current character **e** is encoded with a probability determined by the number of times this match has been found in the past, and the match count (in the corresponding node in the list) is updated.

Figure 5.72b shows a situation where no deterministic match is found. The current character is again an **e** and its order-12 context is the same `ll_assume_th`, but the only string `ll_assume_th` in the data file is preceded by the three characters `y_a`. The encoder does not find a 15-character match, and it proceeds as follows: (1) It outputs an escape to indicate that no deterministic match has been found. (2) It invokes the LOE procedure to compute an order. (3) It uses the order to predict the current symbol the way ordinary PPM does. (4) It takes steps to ensure that such a case will not happen again.

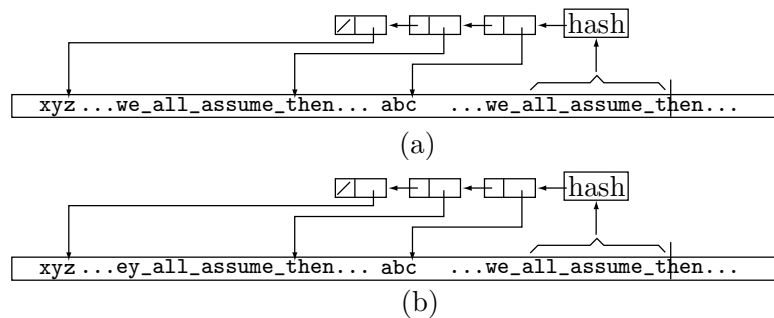


Figure 5.72: Unbounded-Length Deterministic Contexts in PPMZ.

In the first of these steps, the encoder appends a node to the list and sets it to point to the new 12-character context `ll_assume_th`. The second step increments the minimum match length of both contexts by 1 (i.e., to 16 characters). This ensures that these two contexts will be used in the future only when the encoder can match enough characters to distinguish between them.

This complex procedure is executed by the PPMZ encoder to guarantee that all the unbounded-length contexts are deterministic.

Local order estimation is another innovation of PPMZ. Traditional PPM uses the same value for  $N$  (the maximum order length, typically 5–6) for all input streams, but a more sophisticated version should attempt to estimate different values of  $N$  for different input files or even for different contexts within an input file. The LOE computation performed by PPMZ tries to decide which high-order contexts are unreliable. LOE finds a matching context, examines it in each order, and computes a *confidence rating* for each order.

At first, it seems that the best measure of confidence is the entropy of the context, because the entropy estimates the length of the output in bits. In practice, however, this measure turned out to underestimate the reliability of long contexts. The reason mentioned by the method's developer is that a certain symbol  $X$  may be common in an input stream; yet any specific context may include  $X$  only once.

The confidence measure finally selected for LOE is based on the probability  $P$  of the most probable character in the context. Various formulas involving  $P$  were tried, and all resulted in about the same performance. The conclusion was that the best confidence measure for LOE is simply  $P$  itself.

The last important feature of PPMZ is the way it estimates the escape probabilities. This is called secondary escape estimation or SEE. The main idea is to have an adaptive algorithm where not only the counts of the escapes but also the way the counts are computed are determined by the input stream. In each context, the PPMC method of counting the escapes is first applied. This method counts the number of novel characters (i.e., characters found that were not predicted) in the context. This information is then used to construct an escape context which, in turn, is used to look up the escape probability in a table.

The escape context is a number constructed from the four fields listed here, each quantized to a few bits. Both linear and logarithmic quantization were tried. Linear quantization simply truncates the least-significant bits. Logarithmic quantization computes the logarithm of the number to be quantized. This quantizes large numbers more than small ones, with the result that small values remain distinguishable, while large values may become equal. The four components of the escape context are as follows:

1. The PPM order (which is between 0 and 8), quantized to two bits.
2. The escape count, quantized to two bits.
3. The number of successful matches (total count minus the escape count), quantized to three bits.
4. Ten bits from the two characters  $xy$  preceding the current symbol  $S$ . Seven bits are taken from  $x$  and three bits from  $y$ .

This number becomes the order-2 escape context. After deleting some bits from it, PPMZ also creates order-1 and order-0 contexts (15 bits and 7 bits long, respectively). The escape contexts are used to update a table of escape counts. Each entry in this table corresponds to matches coded from past PPM contexts that had the same escape contexts. The information in the table is then used in a complex way to construct the escape probability that is sent to the arithmetic coder to code the escape symbol itself.

The advantage of this complex method is that it combines the statistics gathered from the long (high order) contexts. These contexts provide high compression but are sparse, causing the original PPM to overestimate their escape probabilities.

Applied to the entire (concatenated) Calgary Corpus, PPMZ resulted in an average of 2.119 bpc. This is 10% better than the 2.34 bpc obtained by PPM\* and 17% better than the 2.48 bpc achieved by PPMC.

### 5.14.8 Fast PPM

Fast PPM is a PPM variant developed and implemented by [Howard and Vitter 94b] as a compromise between speed and performance of PPM. Even though it is recognized as one of the best (if not *the* best) statistical compression method, PPM is not very popular because it is slow. Most general-purpose lossless compression software implementations select a dictionary-based method. Fast PPM attempts to isolate those features of PPM that contribute only marginally to compression performance and replace them by approximations. It was hoped that this would speed up execution to make this version competitive with common, commercial lossless compression products.

PPM has two main aspects: modeling and encoding. The modeling part searches the contexts of the current symbol to compute its probability. The fast version simplifies that part by eliminating the explicit use of escape symbols, computing approximate probabilities, and simplifying the exclusion mechanism. The encoding part of PPM uses adaptive arithmetic coding. The fast version speeds up this part by using quasi-arithmetic coding, a method developed by the same researchers [Howard and Vitter 92c] and not discussed here.

The modeling part of fast PPM is illustrated in Table 5.73. We assume that the input stream starts with the string `abcbabdbaeabbabe` and that the current symbol is the second `e` (the last character of the string). Part (a) of the table lists the contexts of this character starting with order 3. The order-3 context of this `e` is `bab`, which has been seen once in the past, but was followed by a `d`, so it cannot be used to predict the current `e`. The encoder therefore skips to order 2, where the context `ab` was seen three times, but never followed by an `e` (notice that the `d` following `ab` has to be excluded). Skipping to order 1, the encoder finds four different symbols following the order-1 context `b`. They are `c`, `a`, `d`, and `b`. Of these, `c`, `d`, and `b` have already been seen, following longer contexts, and are therefore excluded, and the `d` is designated NF (not found), because we are looking for an `e`. Skipping to order 0, the encoder finally finds `e`, following `a`, `b`, `c`, and `d`, which are all excluded. The point is that both the encoder and decoder of fast PPM can easily generate this table with the information available to them. All that the decoder needs in order to decode the `e` is the number of NFs (4 in our example) in the table.

Part (b) of the table illustrates the situation when the sixth `b` (there are seven `b`'s in all) is the current character. It shows that this character can be identified to the decoder by encoding three NFs and writing them on the compressed stream.

A different way of looking at this part of fast PPM is to imagine that the encoder generates a list of symbols, starting at the highest order and eliminating duplicates. The list for part (a) of the table consists of `dcbae` (four NFs followed by an F), while the list for part (b) is `cdab` (three NFs followed by an F).

Order	Context	Symbol	Count	Action	Order	Context	Symbol	Count	Action
3	bab	d	1	NF, $\rightarrow$ 2	3	eab	—	—	$\rightarrow$ 2
2	ab	c	1	NF	2	ab	c	1	NF
		d	1	exclude			d	1	NF, $\rightarrow$ 1
		b	1	NF, $\rightarrow$ 1	1	b	c	1	exclude
1	b	c	1	exclude			a	2	NF
		a	3	NF			d	1	exclude, $\rightarrow$ 0
		d	1	exclude	0		a	4	exclude
		b	1	exclude, $\rightarrow$ 0			b	5	found
0		a	5	exclude					
		b	7	exclude					
		c	1	exclude					
		d	1	exclude					
		e	1	found					

(a)
(b)

Figure 5.73: Two Examples of Fast PPM For abcbabdbaeabbabe.

Temporal reasoning involves both prediction and explanation. Prediction is projection forwards from causes to effects whilst explanation is projection backwards from effects to causes. That is, prediction is reasoning from events to the properties and events they cause, whilst explanation is reasoning from properties and events to events that may have caused them. Although it is clear that a complete framework for temporal reasoning should provide facilities for solving both prediction and explanation problems, prediction has received far more attention in the temporal reasoning literature than explanation.

—Murray Shanahan, *Proceedings IJCAI 1989*

Thus, fast PPM encodes each character by encoding a sequence of NFs, followed by one F (found). It therefore uses a binary arithmetic coder. For increased speed, quasi-arithmetic coding is used, instead of the more common QM coder of Section 5.11. For even faster operation, the quasi-arithmetic coder is used to encode the NFs only for symbols with highest probability, then use a Rice code (Section 10.9) to encode the symbol's (**e** or **b** in our example) position in the remainder of the list. Variants of fast PPM can eliminate the quasi-arithmetic coder altogether (for maximum speed) or use it all the way (for maximum compression).

The results of applying fast PPM to the Calgary corpus are reported by the developers and seem to justify its development effort. The compression performance is 2.341 bpc (for the version with just quasi-arithmetic coder) and 2.287 bpc (for the version with both quasi-arithmetic coding and Rice code). This is somewhat worse than the 2.074 bpc achieved by PPMC. However, the speed of fast PPM is about 25,000–30,000 characters per second, compared to about 16,000 characters per second for PPMC—a speedup factor of about 2!

## 5.15 PAQ

PAQ is an open-source high-performance compression algorithm and free software that features sophisticated prediction (modeling) combined with adaptive arithmetic encoding. PAQ encodes individual bits from the input data and its main novelty is its adaptive prediction method, which is based on mixing predictions (or models) obtained from several contexts. PAQ is therefore related to PPM (Section 5.14). The main references are [wikiPAQ 08] and [Mahoney 08] (the latter includes four pdf files with many details about PAQ and its varieties).

PAQ has been a surprise to the data compression community. Its algorithm naturally lends itself to all kinds of additions and improvements and this fact, combined with its being open source, inspired many researchers to extend and improve PAQ. In addition to describing the chief features of PAQ, this section discusses the main historical milestones in the rapid development of this algorithm, its versions, subversions, and derivatives. The conclusion at the end of this section is that PAQ may signal a change of paradigm in data compression. In the foreseeable future we may see more extensions of existing algorithms and fewer new, original methods and approaches to compression.

The original idea for PAQ is due to Matt Mahoney. He knew that PPM achieves high compression factors and he wondered whether the main principle of PPM, prediction by examining contexts, could be improved by including several predictors (i.e., models of the data) and combining their results in sophisticated ways. As always, there is a price to pay for improved performance, and in the case of PAQ the price is increased execution time and memory requirements.

Starting in 2002, Mahoney and others have come up with eight main versions of PAQ, each of which acquired several subversions (or variations), some especially tuned for certain types of data. As of late 2008, the latest version, PAQ8, has about 20 subversions.

The PAQ predictor predicts (i.e., computes a probability for) the next input bit by employing several context-based prediction methods (mathematical models of the data). An important feature is that the contexts do not have to be contiguous. Here are the main predictors used by the various versions of PAQ:

- An order- $n$  context predictor, as used by PPM. The last  $n$  bits encoded are examined and the numbers of zeros and 1's are counted to estimate the probabilities that the next bit will be a 0 or a 1.
- Whole word order- $n$  context. The context is the latest  $n$  whole words input. Nonalphabetic characters are ignored and upper- and lowercase letters are considered identical. This type of prediction makes sense for text files.
- Sparse contexts. The context consists of certain noncontiguous bytes preceding the current bit. This may be useful in certain types of binary files.
- A high-order context. The next bit is predicted by examining and counting the bits in the high-order parts of the latest bytes or 16-bit words. This has proved useful for the compression of multimedia files.
- Two-dimensional context. An image is a rectangle of correlated pixels, so we can expect consecutive rows of an image to be very similar. If a row is  $c$  pixels long, then



the next pixel to be predicted should be similar to older pixels at distances of  $c$ ,  $2c$ , and so on. This kind of context also makes sense for tables and spreadsheets.

■ A compressed file generally looks random, but if we know the compression method, we can often see order in the file. Thus, common image compression methods, such as jpeg, tiff, and PNG, produce files with certain structures which can be exploited by specialized predictors.

The PAQ encoder examines the beginning of the input file, trying to determine its type (text, jpeg, binary, spreadsheet, etc.). Based on the result, it decides which predictors to use for the file and how to combine the probabilities that they produce for the next input bit. There are three main ways to combine predictions, as follows:

1. In version 1 through 3 of PAQ, each predictor produces a pair of bit counts  $(n_0, n_1)$ . The pair for each predictor is multiplied by a weight that depends on the length of the context used by the predictor (the weight for an order- $n$  context is  $(n+1)^2$ ). The weighted pairs are then added and the results normalized to the interval  $[0, 1]$ , to become meaningful as probabilities.

2. In versions 4 through 6, the weights assigned to the predictors are not fixed but are adjusted adaptively in the direction that would reduce future prediction mistakes. This adjustment tends to favor the more accurate predictors. Specifically, if the predicted bit is  $b$ , then weight  $w_i$  of the  $i$ th predictor is adjusted in the following steps

$$\begin{aligned} n_i &= n_{0i} + n_{1i}, \\ \text{error} &= b - p_1, \\ w_i &\leftarrow w_i + [(S \cdot n_{1i} - S_1 \cdot n_i) / (S_0 \cdot S_1)] \cdot \text{error}, \end{aligned}$$

where  $n_{0i}$  and  $n_{1i}$  are the counts of zeros and 1's produced by the predictor,  $p_1$  is the probability that the next bit will be a 1,  $n_i = n_{0i} + n_{1i}$ , and  $S$ ,  $S_0$ , and  $S_1$  are defined by Equation (5.6).

3. Starting with version 7, each predictor produces a probability rather than a pair of counts. The individual probabilities are then combined with an artificial neural network in the following manner

$$\begin{aligned} x_i &= \text{stretch}(p_{i1}), \\ p_1 &= \text{squash}\left(\sum_i w_i x_i\right), \end{aligned}$$

where  $p_1$  is the probability that the next input bit will be a 1,  $p_{i1}$  is the probability computed by the  $i$ th predictor, and the stretch and squash operations are defined as  $\text{stretch}(x) = \ln(x/(1-x))$  and  $\text{squash}(x) = 1/(1+e^{-x})$ . Notice that they are the inverse of each other.

After each prediction and bit encoding, the  $i$ th predictor is updated by adjusting the weight according to  $w_i \leftarrow w_i + \mu x_i (b - p_1)$ , where  $\mu$  is a constant (typically 0.002 to 0.01) representing the adaptation rate,  $b$  is the predicted bit, and  $(b - p_1)$  is the prediction error.

In PAQ4 and later versions, the weights are updated by

$$w_i \leftarrow \max[0, w_i + (b - p_i)(S n_{1i} - S_1 n_i) / S_0 S_1],$$

where  $n_i = n_{0i} + n_{1i}$ .

PAQ prediction (or modeling) pays special attention to the difference between stationary and nonstationary data. In the former type, the distribution of symbols (the statistics of the data), remains the same throughout the different parts of the input file. In the latter type, different regions of the input file discuss different topics and consequently feature different symbol distributions.

PAQ modeling recognizes that prediction algorithms for nonstationary data must perform poorly for stationary data and vice versa, which is why the PAQ encoder modifies predictions for stationary data to bring them closer to the behavior of nonstationary data.

Modeling stationary binary data is straightforward. The predictor initializes the two counts  $n_0$  and  $n_1$  to zero. If the next bit to be encoded is  $b$ , then count  $n_b$  is incremented by 1. At any point, the probabilities for a 0 and a 1 are  $p_0 = n_0/(n_0 + n_1)$  and  $p_1 = n_1/(n_0 + n_1)$ . Modeling nonstationary data is more complex and can be done in various ways. The approach considered by PAQ is perhaps the simplest. If the next bit to be encoded is  $b$ , then increment count  $n_b$  by 1 and clear the other counter. Thus, a sequence of consecutive zeros will result in higher and higher values of  $n_0$  and in  $n_1 = 0$ . Essentially, this is the same as predicting that the last input will repeat. Once the next bit of 1 appears in the input,  $n_0$  is cleared,  $n_1$  is set to 1, and we expect (or predict) a sequence of consecutive 1's.

In general, the input data may be stationary or nonstationary, so PAQ needs to combine the two approaches above; it needs a semi-stationary rule to update the two counters, a rule that will prove itself experimentally over a wide range of input files. The solution? Rather than keep all the counts (as in the stationary model above) or discard all the counts that disagree with the last input (as in the nonstationary model), the semi-stationary model discards about half the counts. Specifically, it keeps at least two counts and discards half of the remaining counts. This modeling has the effect of starting off as a stationary model and becoming nonstationary as more data is input and encoded and the counts grow. The rule is: if the next bit to be encoded is  $b$ , then increment  $n_b$  by 1. If the other counter,  $n_{1-b}$  is greater than 2, set it to  $\lfloor 1 + n_{1-b}/2 \rfloor$ .

The modified counts of the  $i$ th predictor are denoted by  $n_{0i}$  and  $n_{1i}$ . The counts from the predictors are combined as follows

$$\begin{aligned} S_0 &= \epsilon + \sum_i w_i n_{0i}, & S_1 &= \epsilon + \sum_i w_i n_{1i}, \\ S &= S_0 + S_1, & p_0 &= S_0/S, & p_1 &= S_1/S, \end{aligned} \quad (5.6)$$

where  $w_i$  is the weight assigned to the  $i$ th predictor,  $\epsilon$  is a small positive parameter (whose value is determined experimentally) to make sure that  $S_0$  and  $S_1$  are never zeros, and the division by  $S$  normalizes the counts to probabilities.

The predicted probabilities  $p_0$  and  $p_1$  from the equation above are sent to the arithmetic encoder together with the bit to be encoded.

### History of PAQ

The first version, PAQ1, appeared in early 2002. It was developed and implemented by Matt Mahoney and it employed the following contexts:

- Eight contexts of length zero to seven bytes. These served as general-purpose models. Recall that PAQ operates on individual bits, not on bytes, so each of these models also includes those bits of the current byte that precede the bit currently being predicted and encoded (there may be between zero and seven such bits). The weight of such a context of length  $n$  is  $(n + 1)^2$ .
- Two word-oriented contexts of zero or one whole words preceding the currently predicted word. (A word consists of just the 26 letters and is case insensitive.) This constitutes the unigram and bigram models.
- Two fixed-length record models. These are specialized models for two-dimensional data such as tables and still images. This predictor starts with the byte preceding the current byte and looks for an identical byte in older data. When such a byte is located at a distance of, say,  $c$ , the model checks the bytes at distances  $2c$  and  $3c$ . If all four bytes are identical, the model assumes that  $c$  is the row length of the table, image, or spreadsheet.
- One match context. This predictor starts with the eight bytes preceding the current bit. It locates the nearest group of eight (or more) consecutive bytes, examines the bit  $b$  that follows this context, and predicts that the current bit will be  $b$ .

All models except the last one (match) employ the semi-stationary update rule described earlier.

PAQ2 was written by Serge Osnach and released in mid 2003. This version adds an SSE (secondary symbol estimation) stage between the predictor and encoder of PAQ1. SSE inputs a short, 10-bit context and the current prediction and outputs a new prediction from a table. The table entry is then updated (in proportion to the prediction error) to reflect the actual bit value.

PAQ3 has two subversions, released in late 2003 by Matt Mahoney and Serge Osnach. It includes an improved SSE and three additional sparse models. These employ two-byte contexts that skip over intermediate bytes.

PAQ4, again by Matt Mahoney, came out also in late 2003. It used adaptive weighting of 18 contexts, where the weights are updated by

$$w_i \leftarrow \max[0, w_i + (b - p_i)(S_{n_{1i}} - S_1 n_i)/S_0 S_1],$$

where  $n_i = n_{0i} + n_{1i}$ .

PAQ5 (December 2003) and PAQ6 (the same month) represent minor improvements, including a new analog model. With these versions, PAQ became competitive with the best PPM implementations and began to attract the attention of researchers and users. The immediate result was a large number (about 30) of small, incremental improvements and subversions. The main contributors were Berto Destasio, Johan de Bock, Fabio Buffoni, Jason Schmidt, and David A. Scott. These versions added a number of features as follows:

- A second mixer whose weights were selected by a 4-bit context (consisting of the two most-significant bits of the last two bytes).
- Six new models designed for audio (8 and 16-bit mono and stereo audio samples), 24-bit color images, and 8-bit data.

- Nonstationary, run-length models (in addition to the older, semi-stationary model).
- A special model for executable files with machine code for Intel x86 processors.
- Many more contexts, including general-purpose, match, record, sparse, analog, and word contexts.

In May, 2004, Alexander Ratushnyak threw his hat into the ring and entered the world of PAQ and incremental compression improvements. He named his design PAQAR and started issuing versions in swift succession (seven versions and subversions were issued very quickly). PAQAR extends PAQ by adding many new models, multiple mixers with weights selected by context, an SSE stage to each mixer output, and a preprocessor to improve the compression of Intel executable files (relative `CALL` and `JMP` operands are replaced by absolute addresses). The result was significant improvements in compression, at the price of slow execution and larger memory space.

Another spinoff of PAQ is PAsQDa, four versions of which were issued by Przemysław Skibiński in early 2005. This algorithm is based on PAQ6 and PAQAR, with the addition of an English dictionary preprocessor (termed a word reducing transform, or WRT, by its developer). WRT replaces English words with a code (up to three bytes long) from a dictionary. It also performs other transforms to model punctuations, capitalizations, and end-of-lines. (It should be noted that several specialized subversions of PAQ8 also employ text preprocessing to achieve minor improvements in the compression of certain text files.) PAsQDa achieved the top ranking on the Calgary corpus but not on most other benchmarks.

Here are a few words about one of the terms above. For the purposes of context prediction, the end-of-line (EOL) character is artificial in the sense that it intervenes between symbols that are correlated and spoils their correlation (or similarities). Thus, it makes sense to replace the EOL (which is ASCII character CR, LF, or both) with a space. This idea is due to Malcolm Taylor.

Another significant byproduct of PAQ6 is RK (or WinRK), by Malcolm Taylor. This variant of PAQ is based on a mixing algorithm named PWCM (PAQ weighted context mixing) and has achieved first place, surpassing PAQAR and PAsQDa, in many benchmarks.

In late 2007, Matt Mahoney released PAQ7, a major rewrite of PAQ6, based on experience gained with PAQAR and PAsQDa. The major improvement is not the compression itself but execution speed, which is up to three times faster than PAQAR. PAQ7 lacks the WRT dictionary and the specialized facilities to deal with Intel x86 executable code, but it does include predictors for jpg, bmp, and tiff image files. As a result, it does not compress executables and text files as well as PAsQDa, but it performs better on data that has text with embedded images, such as MS-excel, MS-Word and PDF. PAQ7 also employs a neural network to combine predictions.

PAQ8 was born in early 2006. Currently (early 2009), there are about 20 subversions, many issued in series, that achieve incremental improvements in performance by adding, deleting, and updating many of the features and facilities mentioned above. These subversions were developed and implemented by many of the past players in the PAQ arena, as well as a few new ones. The interested reader is referred to [wikiPAQ 08] for the details of these subversions. Only one series of PAQ subversions, namely the eight algorithms PAQ8HP1 through PAQ8HP8, will be mentioned here. They were re-

leased by Alexander Ratushnyak from August, 2006 through January, 2007 specifically to claim improvement steps for the Hutter Prize (Section 5.13).

I wrote an experimental PAQ9A in Dec. 2007 based on a chain of 2-input mixers rather than a single mixer as in PAQ8, and an LZP preprocessor for long string matches. The goal was better speed at the cost of some compression, and to allow files larger than 2 GB. It does not compress as well as PAQ8, however, because it has fewer models, just order- $n$ , word, and sparse. Also, the LZP interferes with compression to some extent.

I wrote LPAQ1 in July 2007 based on PAQ8 but with fewer models for better speed. (Similar goal to PAQ9A, but a file compressor rather than an archiver). Alexander Ratushnyak wrote several later versions specialized for text which are now #3 on the large text benchmark.

—Matt Mahoney to D. Salomon, Dec. 2008

## A Paradigm Change

Many of the algorithms, techniques, and approaches described in this book have originated in the 1980s and 1990s, which is why these two decades are sometimes referred to as the golden age of data compression. However, the unexpected popularity of PAQ with its many versions, subversions, derivatives, and spinoffs (several are briefly mentioned below) seems to signal a change of paradigm. Instead of new, original algorithms, may we expect more and more small, incremental improvements of existing methods and approaches to compression? No one has a perfect crystal ball, and it is possible that there are clever people out there who have novel ideas for new, complex, and magical compression algorithms and are only waiting for another order of magnitude increase in processor speed and memory size to implement and test their ideas. We'll have to wait and see.

## PAQ Derivatives

Another reason for the popularity of PAQ is its being free. The algorithm is not encumbered by a patent and existing implementations are free. This has encouraged several researchers and coders to extend PAQ in several directions. The following is a short list of the most notable of these derivatives.

- WinUDA 0.291, is based on PAQ6 but is faster [UDA 08].
- UDA 0.301, is based on the PAQ8I algorithm [UDA 08].
- KGB is essentially PAQ6v2 with a GUI (beta version supports PAQ7 compression algorithms). See [KGB 08].
- Emilcont, an algorithm based on PAQ6 [Emilcont 08].
- Peazip. This is a GUI frontend (for Windows and Linux) for LPAQ and various PAQ8\* algorithms [PeaZip 08].
- PWCM (PAQ weighted context mixing) is an independently developed closed source implementation of the PAQ algorithm. It is used (as a plugin) in WinRK to win first place in several compression benchmarks.

## 5.16 Context-Tree Weighting

Whatever the input stream is, text, pixels, sound, or anything else, it can be considered a binary string. Ideal compression (i.e., compression at or very near the entropy of the string) would be achieved if we could use the bits that have been input so far in order to predict with certainty (i.e., with probability 1) the value of the next bit. In practice, the best we can hope for is to use history to estimate the probability that the next bit will be 1. The context-tree weighting (CTW) method [Willems et al. 95] starts with a given bit-string  $b_1^t = b_1 b_2 \dots b_t$  and the  $d$  bits that precede it  $c_d = b_{-d} \dots b_{-2} b_{-1}$  (the context of  $b_1^t$ ). The two strings  $c_d$  and  $b_1^t$  constitute the input stream. The method uses a simple algorithm to construct a tree of depth  $d$  based on the context, where each node corresponds to a substring of  $c_d$ . The first bit  $b_1$  is then input and examined. If it is 1, the tree is updated to include the substrings of  $c_d b_1$  and is then used to calculate (or estimate) the probability that  $b_1$  will be 1 given context  $c_d$ . If  $b_1$  is zero, the tree is updated differently and the algorithm again calculates (or estimates) the probability that  $b_1$  will be zero given the same context. Bit  $b_1$  and its probability are then sent to an arithmetic encoder, and the process continues with  $b_2$ . The context bits themselves are written on the compressed stream in raw format.

The depth  $d$  of the context tree is fixed during the entire compression process, and it should depend on the expected correlation among the input bits. If the bits are expected to be highly correlated, a small  $d$  may be enough to get good probability predictions and thus good compression.

In thinking of the input as a binary string, it is customary to use the term “source.” We think of the bits of the inputs as coming from a certain information source. The source can be *memoryless* or it can have memory. In the former case, each bit is independent of its predecessors. In the latter case, each bit depends on some of its predecessors (and, perhaps, also on its successors, but these cannot be used because they are not available to the decoder), so they are correlated.

We start by looking at a memoryless source where each bit has probability  $P_a(1)$  of being a 1 and probability  $P_a(0)$  of being a 0. We set  $\theta = P_a(1)$ , so  $P_a(0) = 1 - \theta$  (the subscript  $a$  stands for “actual” probability). The probability of a particular string  $b_1^t$  being generated by the source is denoted by  $P_a(b_1^t)$ , and it equals the product

$$P_a(b_1^t) = \prod_{i=1}^t P_a(b_i).$$

If string  $b_1^t$  contains  $a$  zeros and  $b$  ones, then  $P_a(b_1^t) = (1 - \theta)^a \theta^b$ .

Example: Let  $t = 5$ ,  $a = 2$ , and  $b = 3$ . The probability of generating a 5-bit binary string with two zeros and three ones is  $P_a(b_1^5) = (1 - \theta)^2 \theta^3$ . Table 5.74 lists the values of  $P_a(b_1^5)$  for seven values of  $\theta$  from 0 to 1. It is easy to see that the maximum is obtained when  $\theta = 3/5$ . To understand these values intuitively we examine all 32 5-bit numbers. Ten of them consist of two zeros and three ones, so the probability of generating such a string is  $10/32 = 0.3125$  and the probability of generating a particular string out of these 10 is 0.03125. This number is obtained for  $\theta = 1/2$ .

In real-life situations we don’t know the value of  $\theta$ , so we have to estimate it based on what has been input in the past. Assuming that the immediate past string  $b_1^t$  consists

$\theta$ :	0	1/5	2/5	1/2	3/5	4/5	5/5
$P_a(2, 3)$ :	0	0.00512	0.02304	0.03125	0.03456	0.02048	0

Table 5.74: Seven Values of  $P_a(2, 3)$ .

of  $a$  zeros and  $b$  ones, it makes sense to estimate the probability of the next bit being 1 by

$$P_e(b_{t+1} = 1 | b_1^t) = \frac{b}{a + b},$$

where the subscript  $e$  stands for “estimate” (the expression above is read “the estimated probability that the next bit  $b_{t+1}$  will be a 1 given that we have seen string  $b_1^t$  is. . .”). The estimate above is intuitive and cannot handle the case  $a = b = 0$ . A better estimate, due to Krichevsky and Trofimov [Krichevsky 81], is called KT and is given by

$$P_e(b_{t+1} = 1 | b_1^t) = \frac{b + 1/2}{a + b + 1}.$$

The KT estimate, like the intuitive estimate, predicts a probability of  $1/2$  for any case where  $a = b$ . Unlike the intuitive estimate, however, it also works for the case  $a = b = 0$ .

### The KT Boundary

All over the globe is a dark clay-like layer that was deposited around 65 million years ago. This layer is enriched with the rare element iridium. Older fossils found under this KT layer include many dinosaur species. Above this layer (younger fossils) there are no dinosaur species found. This suggests that something that happened around the same time as the KT boundary was formed killed the dinosaurs. Iridium is found in meteorites, so it is possible that a large iridium-enriched meteorite hit the earth, kicking up much iridium dust into the stratosphere. This dust then spread around the earth via air currents and was deposited on the ground very slowly, later forming the KT boundary.

This event has been called the “KT Impact” because it marks the end of the Cretaceous Period and the beginning of the Tertiary. The letter “K” is used because “C” represents the Carboniferous Period, which ended 215 million years earlier.

- ◇ **Exercise 5.43:** Use the KT estimate to calculate the probability that the next bit will be a zero given string  $b_1^t$  as the context.

Example: We use the KT estimate to calculate the probability of the 5-bit string 01110. The probability of the first bit being zero is (since there is no context)

$$P_e(0 | \text{null}) = P_e(0 | a=b=0) = \left(1 - \frac{0 + 1/2}{0 + 0 + 1}\right) = 1/2.$$



The probability of the entire string is the product

$$\begin{aligned}
 P_e(01110) &= P_e(2, 3) \\
 &= P_e(0|\text{null})P_e(1|0)P_e(1|01)P_e(1|011)P_e(0|0111) \\
 &= P_e(0|_{a=b=0})P_e(1|_{a=1,b=0})P_e(1|_{a=b=1})P_e(1|_{a=1,b=2})P_e(0|_{a=1,b=3}) \\
 &= \left(1 - \frac{0+1/2}{0+0+1}\right) \cdot \frac{0+1/2}{1+0+1} \cdot \frac{1+1/2}{1+1+1} \cdot \frac{2+1/2}{1+2+1} \cdot \left(1 - \frac{3+1/2}{1+3+1}\right) \\
 &= \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{3}{6} \cdot \frac{5}{8} \cdot \frac{3}{10} = \frac{3}{256} \approx 0.01172.
 \end{aligned}$$

In general, the KT estimated probability of a string with  $a$  zeros and  $b$  ones is

$$P_e(a, b) = \frac{1/2 \cdot 3/2 \cdots (a-1/2) \cdot 1/2 \cdot 3/2 \cdots (b-1/2)}{1 \cdot 2 \cdot 3 \cdots (a+b)}. \quad (5.7)$$

Table 5.75 lists some values of  $P_e(a, b)$  calculated by Equation (5.7). Notice that  $P_e(a, b) = P_e(b, a)$ , so the table is symmetric.

	0	1	2	3	4	5
0	-	1/2	3/8	5/16	35/128	63/256
1	1/2	1/8	1/16	5/128	7/256	21/1024
2	3/8	1/16	3/128	3/256	7/1024	9/2048
3	5/8	5/128	3/256	5/1024	5/2048	45/32768
4	35/128	7/256	7/1024	5/2048	35/32768	35/65536
5	63/256	21/1024	9/2048	45/32768	35/65536	63/262144

Table 5.75: KT Estimates for Some  $P_e(a, b)$ .

Up until now we have assumed a memoryless source. In such a source the probability  $\theta$  that the next bit will be a 1 is fixed. Any binary string, including random ones, is generated by such a source with equal probability. Binary strings that have to be compressed in real situations are generally not random and are generated by a non-memoryless source. In such a source  $\theta$  is not fixed. It varies from bit to bit, and it depends on the past context of the bit. Since a context is a binary string, all the possible past contexts of a bit can be represented by a binary tree. Since a context can be very long, the tree can include just some of the last bits of the context, to be called the *suffix*. As an example consider the 42-bit string

$$S = 000101100111010110001101001011110010101100.$$

Let's assume that we are interested in suffixes of length 3. The first 3 bits of  $S$  don't have long enough suffixes, so they are written raw on the compressed stream. Next we examine the 3-bit suffix of each of the last 39 bits of  $S$  and count how many times each suffix is followed by a 1 and how many times by a 0. Suffix 001, for example, is followed



twice by a 1 and three times by a 0. Figure 5.76a shows the entire suffix tree of depth 3 for this case (in general, this is not a complete binary tree). The suffixes are read from the leaves to the root, and each leaf is labeled with the probability of that suffix being followed by a 1-bit. Whenever the three most recently read bits are 001, the encoder starts at the root of the tree and follows the edges for 1, 0, and 0. It finds  $2/5$  at the leaf, so it should predict a probability of  $2/5$  that the next bit will be a 1, and a probability of  $1 - 2/5$  that it will be a 0. The encoder then inputs the next bit, examines it, and sends it, with the proper probability, to be arithmetically encoded.

Figure 5.76b shows another simple tree of depth 2 that corresponds to the set of suffixes 00, 10, and 1. Each suffix (i.e., each leaf of the tree) is labeled with a probability  $\theta$ . Thus, for example, the probability that a bit of 1 will follow the suffix ... 10 is 0.3. The tree is the *model* of the source, and the probabilities are the *parameters*. In practice, neither the model nor the parameters are known, so the CTW algorithm has to estimate them.

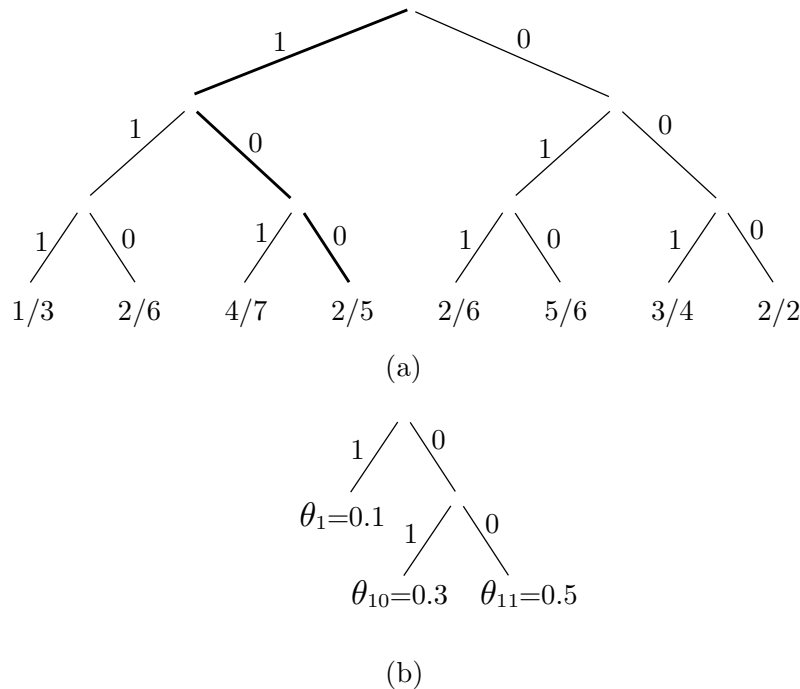


Figure 5.76: Two Suffix Trees.

Next we get one step closer to real-life situations. We assume that the model is known and the parameters are unknown, and we use the KT estimator to estimate the parameters. As an example we use the model of Figure 5.76b but without the probabilities. We use the string  $10|0100110 = 10|b_1b_2b_3b_4b_5b_6b_7$ , where the first two bits are the suffix, to illustrate how the probabilities are estimated with the KT estimator. Bits  $b_1$  and  $b_4$  have suffix 10, so the probability for leaf 10 of the tree is estimated as the KT probability of substring  $b_1b_4 = 00$ , which is  $P_e(2,0) = 3/8$  (two zeros and no ones) from Table 5.75. Bits  $b_2$  and  $b_5$  have suffix 00, so the probability for leaf 00 of the

tree is estimated as the KT probability of substring  $b_2b_5 = 11$ , which is  $P_e(0, 2) = 3/8$  (no zeros and two ones) from Table 5.75. Bits  $b_3 = 0$ ,  $b_6 = 1$ , and  $b_7 = 0$  have suffix 1, so the probability for leaf 1 of the tree is estimated as  $P_e(2, 1) = 1/16$  (two zeros and a single one) from Table 5.75. The probability of the entire string 0100110 given the suffix 10 is thus the product

$$\frac{3}{8} \cdot \frac{3}{8} \cdot \frac{1}{16} = \frac{9}{1024} \approx .0088.$$

- ◇ **Exercise 5.44:** Use this example to estimate the probabilities of the five strings 0, 00, 000, 0000, and 00000, assuming that each is preceded by the suffix 00.

In the last step we assume that the model, as well as the parameters, are unknown. We construct a binary tree of depth  $d$ . The root corresponds to the null context, and each node  $s$  corresponds to the substring of bits that were input following context  $s$ . Each node thus splits up the string. Figure 5.77a shows an example of a context tree for the string  $10|0100110 = 10|b_1b_2b_3b_4b_5b_6b_7$ .

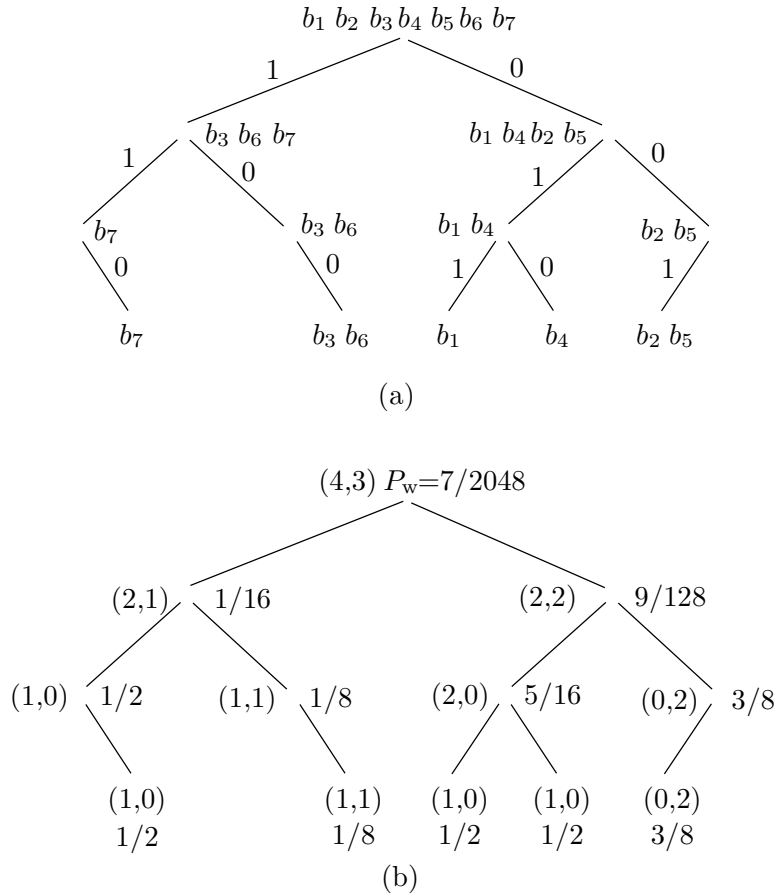


Figure 5.77: (a) A Context Tree. (b) A Weighted Context Tree.

Figure 5.77b shows how each node  $s$  contains the pair  $(a_s, b_s)$ , the number of zeros and ones in the string associated with  $s$ . The root, for example, is associated with the entire string, so it contains the pair  $(4, 3)$ . We still have to calculate or estimate a weighted probability  $P_w^s$  for each node  $s$ , the probability that should be sent to the arithmetic encoder to encode the string associated with  $s$ . This calculation is, in fact, the central part of the CTW algorithm. We start at the leaves because the only thing available in a leaf is the pair  $(a_s, b_s)$ ; there is no suffix. The best assumption that can therefore be made is that the substring consisting of  $a_s$  zeros and  $b_s$  ones that's associated with leaf  $s$  is memoryless, and the best weighted probability that can be defined for the node is the KT estimated probability  $P_e(a_s, b_s)$ . We therefore define

$$P_w^s \stackrel{\text{def}}{=} P_e(a_s, b_s) \quad \text{if } \text{depth}(s) = d. \quad (5.8)$$

Using the weighted probabilities for the leaves, we work our way recursively up the tree and calculate weighted probabilities for the internal nodes. For an internal node  $s$  we know a little more than for a leaf, since such a node has one or two children. The children, which are denoted by  $s_0$  and  $s_1$ , have already been assigned weighted probabilities. We consider two cases. If the substring associated with suffix  $s$  is memoryless, then  $P_e(a_s, b_s)$  is a good weighted probability for it. Otherwise the CTW method claims that the product  $P_w^{s_0} P_w^{s_1}$  of the weighted probabilities of the child nodes is a good coding probability (a missing child node is considered, in such a case, to have weighted probability 1).

Since we don't know which of the above cases is true for a given internal node  $s$ , the best that we can do is to assign  $s$  a weighted probability that's the average of the two cases above, i.e.,

$$P_w^s \stackrel{\text{def}}{=} \frac{P_e(a_s, b_s) + P_w^{s_0} P_w^{s_1}}{2} \quad \text{if } \text{depth}(s) < d. \quad (5.9)$$

The last step that needs to be described is the way the context tree is updated when the next bit is input. Suppose that we have already input and encoded the string  $b_1 b_2 \dots b_{t-1}$ . Thus, we have already constructed a context tree of depth  $d$  for this string, we have used Equations (5.8) and (5.9) to calculate weighted probabilities for the entire tree, and the root of the tree already contains a certain weighted probability. We now input the next bit  $b_t$  and examine it. Depending on what it is, we need to update the context tree for the string  $b_1 b_2 \dots b_{t-1} b_t$ . The weighted probability at the root of the new tree will then be sent to the arithmetic encoder, together with bit  $b_t$ , and will be used to encode  $b_t$ .

If  $b_t = 0$ , then updating the tree is done by (1) incrementing the  $a_s$  counts for all nodes  $s$ , (2) updating the estimated probabilities  $P_e(a_s, b_s)$  for all the nodes, and (3) updating the weighted probabilities  $P_w(a_s, b_s)$  for all the nodes. If  $b_t = 1$ , then all the  $b_s$  should be incremented, followed by updating the estimated and weighted probabilities as above. Figure 5.78a shows how the context tree of Figure 5.77b is updated when  $b_t = 0$ .

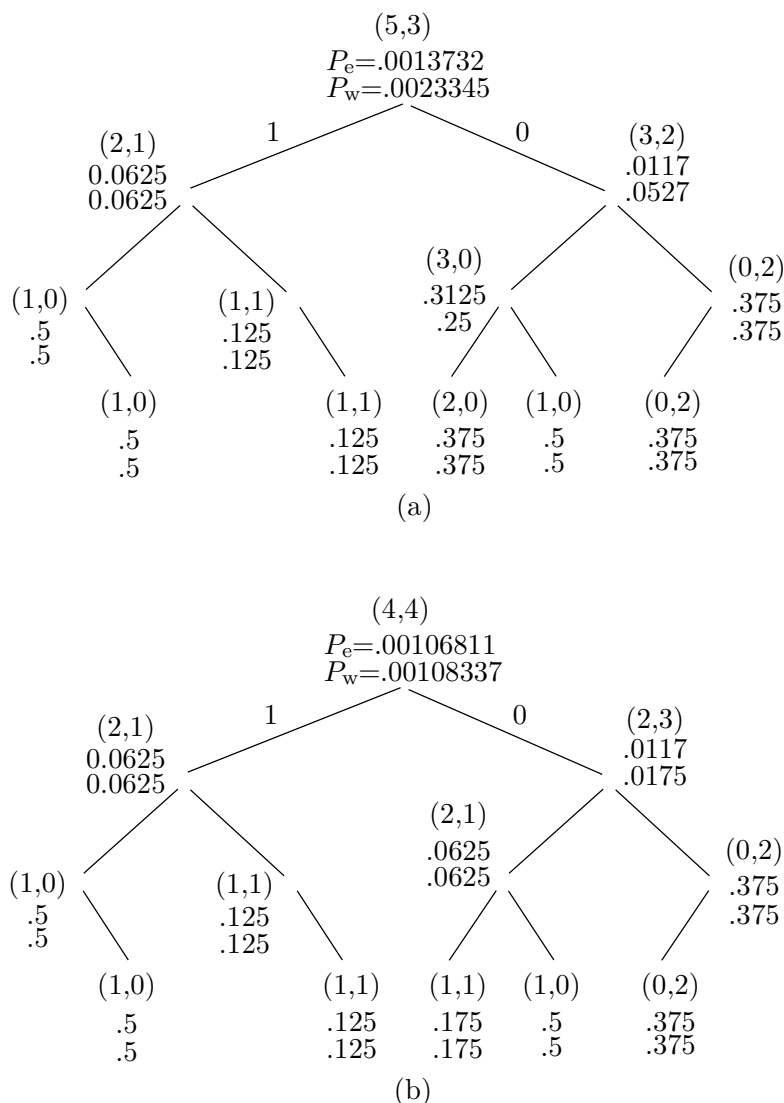
Figure 5.78: Context Trees for  $b_t = 0, 1$ .

Figure 5.78b shows how the context tree of Figure 5.77b is updated when  $b_t = 1$ .

- ◇ **Exercise 5.45:** Construct the context trees with depth 3 for the strings 000|0, 000|00, 000|1, and 000|11.

The depth  $d$  of the context tree is selected by the user (or is built into both encoder and decoder) and does not change during the compression job. The tree has to be updated for each input bit processed, but this requires updating at most  $d + 1$  nodes. The number of operations needed to process  $n$  input bits is thus linear in  $n$ .

I hoped that the contents of his pockets might help me to form a conclusion.

—Arthur Conan Doyle, *Memoires of Sherlock Holmes*

### 5.16.1 CTW for Text Compression

The CTW method has so far been developed for compressing binary strings. In practice, we are normally interested in compressing text, image, and sound streams, and this section discusses one approach to applying CTW to text compression.

Each ASCII character consists of seven bits, and all 128 7-bit combinations are used. However, some combinations (such as E and T) are more common than others (such as Z, <, and certain control characters). Also, certain character pairs and triplets (such as TH and THE) appear more often than others. We therefore claim that if  $b_t$  is a bit in a certain ASCII character  $X$ , then the  $t - 1$  bits  $b_1 b_2 \dots b_{t-1}$  preceding it can act as context (even if some of them are not even parts of  $X$  but belong to characters preceding  $X$ ). Experience shows that good results are obtained (1) with contexts of size 12, (2) when seven context trees are used, each to construct a model for one of the seven bits, and (3) if the original KT estimate is modified to the *zero-redundancy* estimate, defined by

$$P_e^z(a, b) \stackrel{\text{def}}{=} \frac{1}{2} P_e(a, b) + \frac{1}{4} \vartheta(a = 0) + \frac{1}{4} \vartheta(b = 0),$$

where  $\vartheta(\text{true}) \stackrel{\text{def}}{=} 1$  and  $\vartheta(\text{false}) \stackrel{\text{def}}{=} 0$ .

Another experimental feature is a change in the definition of the weighted probabilities. The original definition, Equation (5.9), is used for the two trees on the ASCII borders (i.e., the ones for bits 1 and 7 of each ASCII code). The weighted probabilities for the five context trees for bits 2–6 are defined by  $P_s^w = P_w^{s_0} P_w^{s_1}$ .

This produces typical compression of 1.8 to 2.3 bits/character on the (concatenated) documents of the Calgary Corpus.

Paul Volf [Volf 97] has proposed other approaches to CTW text compression.

Statistics are like bikinis. What they reveal  
is suggestive, but what they conceal is vital.

—Aaron Levenstein

