Text Compression

A paradox of modern computer technology is that despite an almost incomprehensible increase in storage and transmission capacities, more and more effort has been put into using compression to increase the amount of data that can be handled. No matter how much storage space or transmission bandwidth is available, someone always finds something to fill it with. It seems that Parkinson's law applies to space as well as time.

The problem of representing information efficiently is nothing new. People have always been interested in means for storing and communicating information, and methods for compressing text to improve this process predate computers. For example, the Braille code for the blind can include "contractions," which represent common words with two or three characters, and Morse code also "compresses" data by using shorter representations for common characters.

Text compression on a computer involves changing the representation of a file so that it takes less space to store or less time to transmit, yet the original file can be reconstructed exactly from the compressed representation. *Text* compression techniques are distinguished from the more general *data* compression methods because the original file can always be reconstructed exactly. For some types of data other than text, such as sound or images, small changes, or *noise*, in the reconstructed data can be tolerated because it is a digital approximation to an analog waveform anyway. However, with text it must be possible to reproduce the original file exactly.

Many compression methods have been invented and reinvented over the years. These range from numerous ad hoc techniques to more principled methods that can give very good compression. One of the earliest and best-known methods of text compression for computer storage and telecommunications is Huffman coding.[1]

1 David Huffman, then a student at M.I.T., devised his celebrated coding method in response to a challenge from his professor, and as a result managed to avoid having to take the final exam for the course!

This uses the same principle as Morse code: common symbols—conventionally, characters—are coded in just a few bits, while rare ones have longer codewords. First published in the early 1950s, Huffman coding was regarded as one of the best methods of compression for several decades, until two breakthroughs in the late 1970s—Ziv-Lempel compression and arithmetic coding—made higher compression rates possible. Both these ideas achieve their power through the use of *adaptive compression*—a kind of dynamic coding where the input is compressed relative to a model that is constructed from the text that has just been coded. By basing the model on what has been seen so far, adaptive compression methods combine two key virtues: they are able to encode in a single pass through the input file, and they are able to compress a wide variety of inputs effectively rather than being fine-tuned for one particular type of data such as English text.

Ziv-Lempel methods are adaptive compression techniques that give good compression yet are generally very fast and do not require large amounts of memory. The idea behind them was developed by two Israeli researchers, Jacob Ziv and Abraham Lempel, in the late 1970s. *Arithmetic coding* is really an enabling technology that makes a whole class of adaptive compression schemes feasible, rather than a compression method in its own right. Early implementations of character-level Huffman coding were typically able to compress English text to about five bits per character. Ziv-Lempel methods reduced this to fewer than four bits per character—about half the original size. Methods based on arithmetic coding further improved the compression to just over two bits per character. The price paid is slower compression and decompression, and more memory required in the machines that do the processing.

Some of the best compression methods available are variants of a technique called *prediction by partial matching* (PPM), which was developed in the early 1980s. PPM relies on arithmetic coding to obtain good compression performance. Since then, there has been little advance in the amount of compression that can be achieved, other than some fine-tuning of the basic methods, and the development of a new method called *block sorting* that gives similar performance to PPM. On the other hand, many techniques have been discovered that improve the speed or memory requirements of compression methods. Most of these achieve a significant reduction in computing requirements in exchange for a slight loss of compression.

Present compression techniques give compression of about two bits per character for general English text, depending on what you mean by "general English text." Evidence suggests that compression better than one bit per character is not likely to be achieved, and in order to approach this bound, compression methods will have to draw both on the semantic content of the text and external world knowledge. This is discussed further in Section 2.8.

Improvements are still being made in processor and memory utilization during compression, although both of these resources are becoming cheaper and more plentiful. Generally speaking, the amount of compression achieved by the PPM method increases as more memory becomes available. It is not competitive with Ziv-Lempel methods until 100 Kbytes or more are available, and it does not approach its best performance until 500 Kbytes to 1 Mbyte is allocated. Because of this requirement, when PPM was first proposed in the early 1980s it was a laboratory curiosity, requiring a large minicomputer to test it. Now, most PCs have sufficient computing power to execute it quite effectively. Furthermore, processor speed is currently improving at a faster rate than disk speeds and capacities. Since compression decreases the demand on storage devices at the expense of processing, it is becoming more economical to store data in a compressed form than uncompressed.

Most text compression methods can be placed in one of two classes: *symbolwise* methods and *dictionary* methods. Symbolwise methods work by estimating the probabilities of symbols (often characters), coding one symbol at a time, using shorter codewords for the most likely symbols in the same way that Morse code does. Dictionary methods achieve compression by replacing words and other fragments of text with an index to an entry in a "dictionary." The Braille code is a dictionary method since special codes are used to represent whole words.

Symbolwise methods are usually based on either Huffman coding or arithmetic coding, and they differ mainly in how they estimate probabilities for symbols. The more accurately these estimates are made, the greater the compression that can be achieved. To obtain good compression, the probability estimate is usually based on the context in which a symbol occurs. The business of estimating probabilities is called *modeling*, and good modeling is crucial to obtaining good compression. Converting the probabilities into a bitstream for transmission is called *coding*. Coding is well understood and can be performed very effectively using either Huffman coding or arithmetic coding. Modeling is more of an art, and there does not appear to be any single "best" method.

Dictionary methods generally use quite simple representations to code references to entries in the dictionary. They obtain compression by representing several symbols as one output codeword. This contrasts with symbolwise methods, which rely on generating good probability estimates for a symbol, since the length of the output codeword is what determines compression performance; for this reason, symbolwise methods are sometimes referred to as *statistical* methods, since they rely on estimating accurate statistics. The most significant dictionary methods are based on Ziv-Lempel coding, which uses the idea of replacing strings of characters with a reference to a previous occurrence of the string. This approach is adaptive, and it is effective because most characters can be coded as part of a string that has occurred earlier in the text. Compression is achieved if the reference, or *pointer*, is stored in fewer bits than the string it replaces. There are many variations on Ziv-Lempel coding, with different pointer representations and different rules governing which strings can be referenced.

The key distinction between symbolwise and dictionary methods is that symbolwise methods generally base the coding of a symbol on the context in which it occurs, whereas dictionary methods group symbols together, creating a kind of implicit context. Hybrid schemes are possible, in which a group of symbols is coded together and the coding is based on the context in which the group occurs. This does not necessarily provide better compression than symbolwise methods, but it can improve the speed of compression.

The following sections describe in more detail the main compression techniques introduced above. We look at the modeling and coding components separately. First the general idea of modeling is introduced, and then a particularly powerful class of models, adaptive models, is discussed. Before looking at how models are used in practice, we describe the two principal methods of coding used to represent symbols based on the probability distributions generated by models. Each of these descriptions begins with an overview of what the method is and how it works, and each is followed by a much more detailed description of how the coding can be implemented efficiently. These details are included because, although they lead to extremely effective implementations, they are not obvious and are hard to come by in the literature; they should be skipped on a first reading. The two main classes of models, symbolwise and dictionary, are then examined. There is a section that deals with the problem of providing random access to compressed text, which is important for full-text retrieval systems. Finally, the practical performance of various text compression methods is discussed.

2.1 Models

Compression methods obtain high compression by forming good models of the data that is to be coded. The function of a model is to *predict* symbols, which amounts to providing a probability distribution for the next symbol that is to be coded. For example, during the encoding of a text, the "prediction" for the next symbol might include a probability of 2 percent for the letter *u*, based on its relative frequency in a sample of text. The set of all possible symbols is called the *alphabet*, and the probability distribution provides an estimated probability for each symbol in the alphabet.

The model provides this probability distribution to the encoder, which uses it to encode the symbol that actually occurs. The decoder uses an identical model together with the output of the encoder to find out what the encoded symbol was. Figure 2.1 illustrates the whole process.

The number of bits in which a symbol, s, should be coded is called the *information content* of the symbol. The information content, $I(s)$, is directly related to the symbol's predicted probability, $\Pr[s]$, by the function $I(s) = -\log \Pr[s]$ bits.[2] For example, to transmit a symbol representing the fact that the outcome of a fair coin toss was "heads," the best an encoder can do is to use $-\log(1/2) = 1$ bit. The average amount of information per symbol over the whole alphabet is known as the *entropy* of the probability distribution, denoted by H. It is given by

$$H = \sum_s \Pr[s] \cdot I(s) = \sum_s -\Pr[s] \cdot \log \Pr[s].$$

2 All logarithms in this book are to base two unless indicated otherwise.

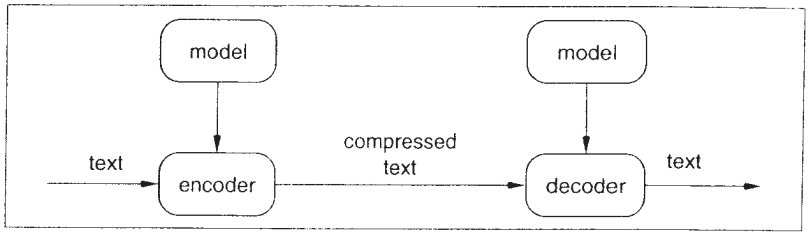

Figure 2.1 Using a model to compress text.

Provided that the symbols appear independently and with the assumed probabilities, H is a *lower* bound on compression, measured in bits per symbol, that can be achieved by *any* coding method. This is the celebrated source coding theorem of Claude Shannon, a Bell Labs scientist who single-handedly developed the field of information theory, and provides a bound that we can strive to attain but can never beat (Shannon 1948).

Huffman coding often achieves compression performance close to the entropy, but can, in some cases, be very inefficient. One such situation is when very good predictions are being made, in which case probabilities close to one are generated. This is exactly when entropy is minimized and "compressibility" is maximized and is what we hope to achieve when designing data compression systems; hence it is unfortunate that Huffman coding is inefficient in this situation. By way of contrast, the more recent method of arithmetic coding comes arbitrarily close to the entropy even when probabilities are close to one and the entropy of the probability distribution is close to zero. These two methods are discussed in Sections 2.3 and 2.4. What is important for the model is to provide a probability distribution that makes the probability of the symbol that actually occurs as high as possible. The above relationship means that a low probability results in a high entropy and vice versa. In the extreme case when $\Pr[s] = 1$, only one symbol s is possible, and $I(s) = 0$ indicates that zero bits are needed to transmit it. This follows intuitively: if a symbol is certain to occur, then it conveys no information and need not be transmitted. To paraphrase a well-worn truism, $I(\text{death}) = 0$ and $I(\text{taxes}) = 0$.

Conversely, I becomes arbitrarily large as $\Pr[s]$ approaches zero, and a symbol with zero probability cannot be coded. In practice, all symbols must be given a nonzero probability because a zero-probability symbol could not be coded if, by unlucky chance, it did occur. Moreover, it is not possible for the encoder to peek at the next symbol and artificially boost its probability just for this step; the encoder and decoder must use the *same* probability distribution, and the decoder clearly cannot look ahead at symbols that have not yet been decoded. Hence the model must take into account all the information available to the decoder and then gamble on what the next symbol will be. The best compression is obtained when the model is backing the symbols that actually occur.

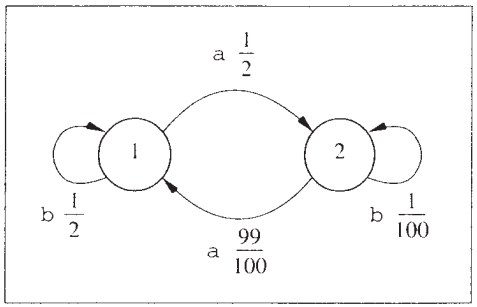

Figure 2.2 A simple finite-state model.

At the beginning of this section the probability of a u was estimated as 2 percent. This corresponds to an information content of 5.6 bits; that is, if it does happen to be the next symbol, u should be transmitted in 5.6 bits. Nothing has been said yet about *how* the probabilities should be estimated. It turns out that predictions can usually be improved by taking account of the previous symbol. If a q has just been encountered, the probability of u may jump to 95 percent, based on how often q is followed by u in a sample of text. This gives a much lower information content for u of 0.074 bits. Of course, other symbols must have lower probabilities and therefore longer codewords to compensate, and if the prediction is incorrect (as in *Iraq* or *Qantas*), the price paid is extra bits in the output. But averaged over many appearances of the context q, the number of bits required to decode each appearance of u can be expected to decrease.

Models that take a few immediately preceding symbols into account to make a prediction are called *finite-context* models of order m, where m is the number of previous symbols used to make the prediction. Such models are effective in a variety of compression applications, and the best text compression methods known are based on this approach.

Other approaches to modeling are possible, and although potentially more powerful, they have not proved as popular as finite-context models. One approach is to use a *finite-state* model, in which each state of a finite-state machine stores a different probability distribution for the next symbol. Figure 2.2 shows such a model. This particular model is for strings in which the symbol a is expected to occur in pairs. Encoding starts in state 1, where a and b are predicted with equal probability, 1/2. Using the formula above, we find that each should be coded in one bit (not surprisingly). If a b is received, the encoder stays in state 1 and uses the same probability distribution for the next symbol. However, if an a is received, it moves to state 2, where the probability of a b is now only 1/100 and requires 6.6 bits to be encoded, as opposed to 0.014 bits to encode an a. This model captures behavior that cannot be represented accurately by a finite-context model because a state model is able to keep track of whether an odd or even number of as have occurred consecutively.

It is important that the decoder works with an identical probability distribution in order to decode symbols correctly. This is achieved by ensuring that it has an identical model to the encoder's and that it starts in the same state as the encoder. The encoder transmits the symbol and then follows a transition; the decoder recovers the symbol and can then follow the same transition, so it is now in the same state as the encoder and will use the same probability distribution for the next symbol. Error-free transmission is assumed, for if any errors were to occur, the encoder and decoder would lose synchronization, with potentially catastrophic results.

If the text being compressed is in a formal language such as C or Java, a grammar can be used to model the language. The text is represented by sending the sequence of *productions*, or rules, that would generate the text from the grammar. By estimating the probability of a particular production occurring, more frequently used productions can be coded in fewer bits, thus achieving good compression. It is hard to obtain a formal grammar for texts written in natural languages, so to date, grammar models have been applied only to artificial languages such as programming languages.

2.2 Adaptive models

There are many ways to estimate the probabilities in a model. We could conceivably guess suitable probabilities when setting up a compression system and use the same distribution for all input texts. However, it is easier, and more accurate, to estimate the probabilities from a sample of the kind of text that is being encoded.

The method that always uses the same model regardless of what text is being coded is called *static* modeling. Clearly, this runs the risk of receiving an input that is quite different from the one for which the model was set up—for example, a model for the English language will probably not perform well with a file of numbers and vice versa. One example of such a mismatch occurs when numeric data is transmitted using Morse code. Because the digits are all relatively rare in normal text, they are assigned long codewords, and so transmission times increase if documents such as financial statements are sent. Another example is shown in Figure 2.3, which is the opening sentence of a rather contorted book—*Gadsby* by E. V. Wright, published in 1939. You may care to try to work out what is unusual about the text before reading on. In fact, the reason that the text reads strangely is that it does not contain a single occurrence of what is usually the most common letter in normal English text—*e*. A static model designed for normal English text would perform poorly in this case.

One solution is to generate a model specifically for each file that is to be compressed. An initial pass is made through the file to estimate symbol probabilities, and these are transmitted to the decoder before transmitting the encoded symbols. This approach is called *semi-static* modeling. (Semi-static modeling has also been referred to as semi-adaptive modeling, but we prefer the term "semi-static" because the implementation of these models has more in common with static models than adaptive ones.) Semi-static modeling has the advantage that the model is invariably

```
If Youth, throughout all history, had had a champion to stand
up for it; to show a doubting world that a child can think;
and, possibly, do it practically; you wouldn't constantly run
across folks today who claim that ''a child don't know anything.''
```

Figure 2.3 The first sentence of an unusual book.

```
"I never heerd a skilful old married feller of twenty
years' standing pipe ''my wife'' in a more used note
than 'a did," said Jacob Smallbury.  "It migh
```

Figure 2.4 Sample text.

better suited to the input than a static one, but the penalty paid is having to transmit the model first, as well as the preliminary pass over the data to accumulate symbol probabilities. In some situations, such as interactive data communications, it may be impractical to make two passes over the data, and for complex models the cost of pretransmitting the model might be a considerable overhead.

Adaptive modeling is an elegant solution to these problems. An adaptive model begins with a bland probability distribution and gradually alters it as more symbols are encountered. As an example, consider an adaptive model that uses the previously encoded part of a string as a sample to estimate probabilities. We will use a model that operates character by character, with no context used to predict the next symbol—in other words, each character of the input is treated as an independent symbol. Technically, this is called a *zero-order* (equivalently, *order-0*) model: in full, an adaptive, zero-order, character-level model. Now consider the text of Figure 2.4, excerpted from Thomas Hardy's book *Far from the Madding Crowd*. It is from the final scene in the book, in which a group is jesting with a newlywed couple. The archaic language in this excerpt is another reminder of the desirability of using adaptive codes.

The zero-order probability that the next character after the excerpt is t is estimated to be $49{,}983/768{,}078 = 6.5$ percent, since in the previous text, 49,983 of the 768,078 characters were ts. Using the same system, an e has probability 9.4 percent, and an x has probability 0.11 percent. The model provides this estimated probability distribution to an encoder such as an arithmetic coder (see Section 2.4). In fact, the next character is a t, which an arithmetic coder represents in about $-\log 0.065 = 3.94$ bits. The decoder is able to generate the same model since it has just decoded all the characters up to (but not including) the t. It makes the same probability estimates as the encoder and so is able to decode the t correctly when it is received. In practice, the encoder and decoder do not extract the statistics from the prior text each time they are needed, but instead keep a running tally of the character counts.

Some details of the adaptive system need to be considered. First, the system must avoid the situation in which a character is predicted with a probability of zero. In the example above, the character Z has never occurred in the text up to this point and would be predicted with zero probability. Such events cannot be coded, yet they might occur; this is referred to as the *zero-frequency problem* (Witten and Bell 1991). The text *It mighZ* is very unlikely, but it can occur. If nowhere else, it has just occurred in this book.

There are several ways to solve the zero-frequency problem. One is to allow one extra count, which is divided evenly among any symbols that have not been observed in the input. In the Hardy example, the total count would be increased by one to 768,079. A total of 82 different characters have been seen so far, so 46 of the 128 ASCII characters have not occurred by this point. Each of these gets $1/46$ of the spare proportion of $1/768{,}079$. Thus, a Z character is given a probability of $1/(46 \times 768{,}079) = 1/35{,}331{,}634$, corresponding to 25.07 bits in the output.

Another possibility is to artificially inflate the count of every character in the alphabet by one, thereby ensuring that none has a zero frequency. This is equivalent to starting the model assuming that we have already processed a stretch of text in which each possible character appeared exactly once. In the above example, allowing for the ASCII alphabet of 128 characters, 128 would be added to the total number of characters seen so far, giving Z a relative frequency of $1/768{,}206 = 0.00013$ percent, corresponding to 19.6 bits in the output.

Several other solutions to the zero-frequency problem are possible, although in general none offers a particularly significant compression performance advantage over the others. The problem is most acute near the beginning of a text where there are few, if any, samples on which to base estimates; so at face value the choice of method is more critical for small texts than for large ones. The method is also important for models that use very many different contexts because many of the contexts will be used only a few times.

The example above used a zero-order model, in which each character's probability was estimated without regard to context. For a higher-order model, such as a first-order model, the probability is estimated by how often that character has occurred in the current context. For example, the excerpt used above to illustrate a zero-order model was coding the letter t in the context of the phrase *It migh*, but in reality made no use at all of the characters comprising that phrase. On the other hand, a first-order model would use the final h as a context with which to *condition* the probability estimates. The letter h has occurred 37,525 times in the prior text, and 1,133 of these times it was followed by a t. Ignoring for a moment the zero-frequency problem, the probability of a t occurring after an h can be estimated to be $1{,}133/37{,}525 = 3.02$ percent, which would have it coded in 5.05 bits. This is actually worse than the zero-order estimate because the letter t is rare in this context—an h is much more likely to be followed by an e, and so here is an example where use of more information caused inferior compression. On the other hand, a second-order model does substantially better. It uses the relative frequency that the context *gh* is followed by a t, which is 1,129 times out of 1,754, or 64.4 percent, and results in the t being coded in just 0.636 bits.

So far we have suggested how the probabilities in a model can be adapted, but it is also possible—and effective—to adapt a model's *structure*. In a finite-context model, the structure determines which contexts are used; in a finite-state model, the structure is the set of states and transitions available. Adaptation usually involves adding more detail to an area of the model that is heavily used. For example, if the first-order context *h* is being used frequently, it might be worthwhile to add more specific contexts, such as the second-order contexts *th* and *sh*. So long as the encoder and decoder use the same rules for adding contexts, and the decision to add contexts is based on the previously encoded text only, they will remain synchronized.

Adaptive modeling is a powerful tool for compression and is the basis of many successful methods. It is robust, reliable, and flexible. The principal disadvantage is that it is not suitable for random access to files—a text can be decoded only from the beginning, since the model used for coding a particular part of the text is determined from all the preceding text. Hence, adaptive modeling is ideal for general-purpose compression utilities but is not necessarily appropriate for full-text retrieval. This point will be taken up again in Section 2.7.

2.3 Huffman coding

Coding is the task of determining the output representation of a symbol, based on a probability distribution supplied by a model. The general idea is that a coder should output short codewords for likely symbols and long codewords for rare ones. There are theoretical limits on how short the average length of a codeword can be for a given probability distribution, and much effort has been put into finding coders that achieve this limit. Another important consideration is the speed of the coder—a reasonable amount of computation is required to generate near-optimal codes. If speed is important, we might use a coder that sacrifices compression performance to reduce the amount of effort required. For example, if there are 256 possible symbols to be coded, we might use a coder that represents the 15 most probable symbols in 4 bits and the remainder in 12 bits. The extreme of this sort of approximation is just to code all symbols in 8 bits. It gives no compression but is very fast. In fact, many dictionary-based methods use a simple coder like this, with the implicit assumption that the symbols (which in this kind of model are actually groups of characters) are equally likely.

In contrast to dictionary methods, symbolwise schemes depend heavily on a good coder to achieve compression, and most research on coders has been performed with symbolwise methods in mind. This section and the next describe the two main methods of coding: *Huffman coding* and *arithmetic coding*. Huffman coding tends to be faster than arithmetic coding, but arithmetic coding is capable of yielding compression that is close to optimal given the probability distribution supplied by the model. For each of these two types of coder, we first look at the principle by which they achieve compression and then give details of how they are implemented in practice. We begin with Huffman coding (Huffman 1952).

Table 2.1 Codewords and probabilities for a seven-symbol alphabet.

Symbol	Codeword	Probability
a	0000	0.05
b	0001	0.05
c	001	0.1
d	01	0.2
e	10	0.3
f	110	0.2
g	111	0.1

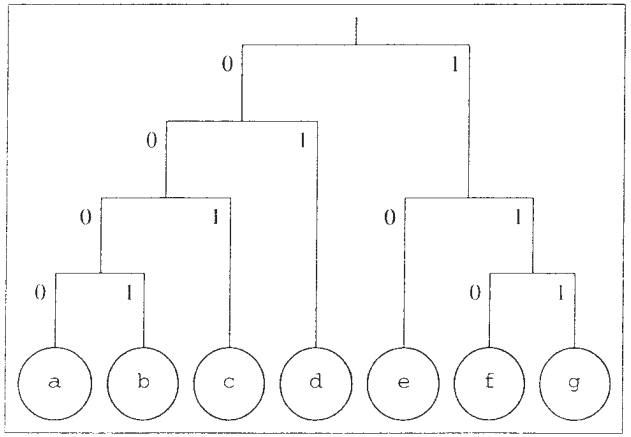

Figure 2.5 A Huffman code tree.

Table 2.1 shows codewords for the seven-symbol alphabet a, b, c, d, e, f, and g. A phrase is coded by replacing each of its symbols with the codeword given by the table. For example, the phrase *eefggfed* is coded as 10101101111111101001. Decoding is performed from left to right. The input to the decoder begins with 10 . . . , and the only codeword that begins with this is the one for e, which is therefore taken as the first symbol. Decoding then proceeds with the remainder of the string, 1011011 . . . .

Figure 2.5 shows a tree that can be used for decoding. The tree is traversed by starting at the root and following the branch corresponding to the next bit in the coded text. The path from the root to each symbol (at a leaf) corresponds to the codewords in Table 2.1. This type of code is called a *prefix* code—or more accurately, a *prefix-free* code—because no codeword is the prefix of another symbol's codeword.

If that were not the case, the decoding tree would have symbols at internal nodes, which leads to ambiguity in decoding.

The code in Table 2.1 was produced by the technique of Huffman coding, which generates codewords for a set of symbols, given some probability distribution for the symbols. The codewords generated yield the best compression possible for a prefix-free code for the given probability distribution.

Huffman's algorithm works by constructing the decoding tree from the bottom up. For the example symbol set, with the probabilities shown in Table 2.1, it starts by creating for each symbol a leaf node containing the symbol and its probability (Figure 2.6a). Then the two nodes with the smallest probabilities become siblings under a parent node, which is given a probability equal to the sum of its two children's probabilities (Figure 2.6b).

The combining operation is repeated, choosing the two nodes with the smallest probabilities and ignoring nodes that are already children. For example, at the next step the new node formed by combining a and b is joined with the node for c to make a new node with probability $p = 0.2$. The process continues until there is only one node without a parent, which becomes the root of the decoding tree (Figure 2.6c). The two branches from every nonleaf node are then labeled 0 and 1 (the order is not important) to form the tree.

Figure 2.7 shows the general algorithm for constructing a Huffman code. The algorithm is expressed in terms of a set T that recursively contains other sets, with each subset corresponding to a node in the tree. When the algorithm terminates, T contains one set, which itself contains two sets—the descriptions of the two subtrees of the root. A more detailed description of how Huffman coding is implemented appears later in this section.

Huffman coding is generally fast for both encoding and decoding, provided that the probability distribution is static. There are also algorithms for adaptive Huffman coding, where localized adjustments are made to the tree to maintain the correct structure as the probabilities change (Gallager 1978; Cormack and Horspool 1984; Knuth 1985; Vitter 1989). However, the better adaptive symbolwise models usually use many different probability distributions at the same time, with the appropriate distribution being chosen depending on the context of the symbol being coded. Huffman coding requires that multiple trees be maintained in this situation, which can become demanding on memory. The alternative is for each tree to be regenerated whenever it is required, but this is slow. Hence, for adaptive compression, arithmetic coding (described in the next section) is usually preferable, as its speed is comparable to that of adaptive Huffman coding, yet it requires less memory and is able to achieve better compression—particularly when high-probability events are being coded.

Nevertheless, Huffman coding turns out to be very useful for some applications. For example, when coupled with a word-based (rather than character-based) model, it gives good compression, and its speed and ease of random access make it more attractive than arithmetic coding. Furthermore, there is a slightly different representation of a Huffman code that decodes very efficiently despite the extremely large models that might arise with a word-based model. This representation is called the

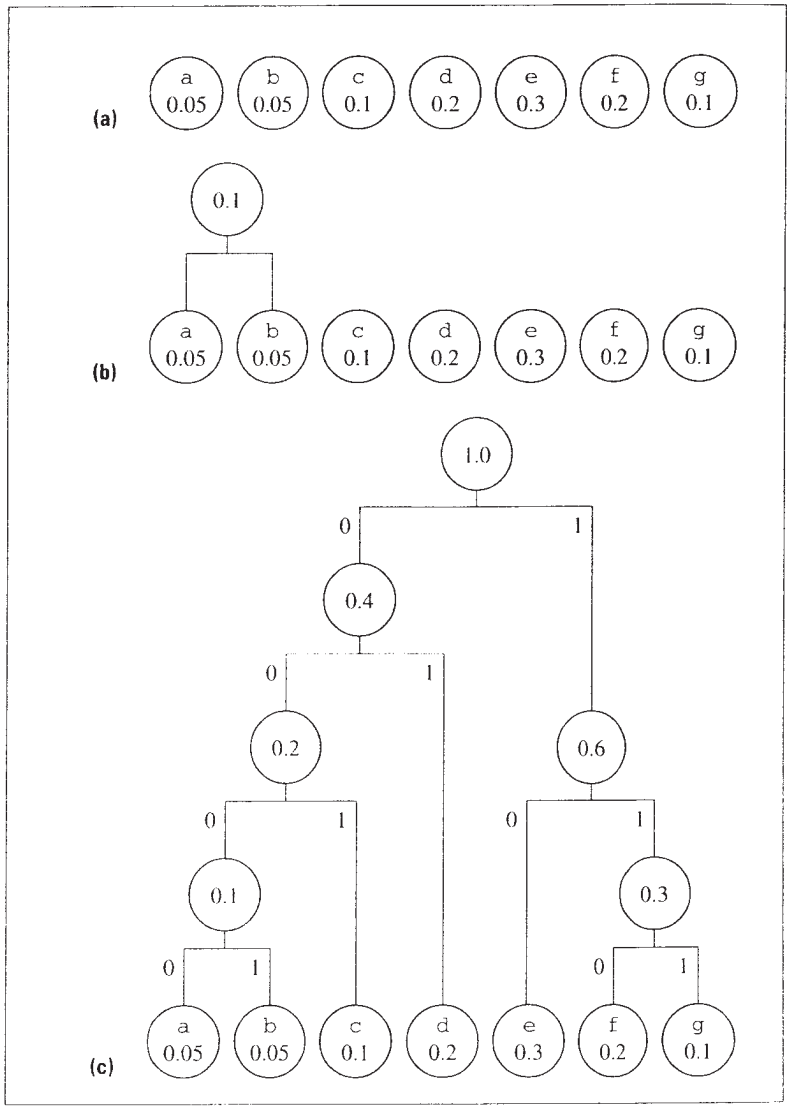

Figure 2.6 Constructing the Huffman tree: (a) leaf nodes; (b) combining nodes; (c) the finished Huffman tree.

To calculate a Huffman code,

1. Set $T \leftarrow$ a set of n singleton sets, each containing one of the n symbols and its probability.
2. Repeat $n - 1$ times
   (a) Set m_1 and $m_2 \leftarrow$ the two subsets of least probability in T.
   (b) Replace m_1 and m_2 with a set $\{m_1, m_2\}$ whose probability is the sum of that of m_1 and m_2.
3. T now contains only one item, which corresponds to the root of a Huffman tree; the length of the codeword for each symbol is given by the number of times it was joined with another set.

Figure 2.7 Assigning a Huffman code.

canonical Huffman code (Hirschberg and Lelewer 1990). It uses the same codeword *lengths* as a Huffman code, but imposes a particular choice on the codeword *bits*.

Table 2.2 shows part of a canonical Huffman code for the Hardy book, where the alphabet has been chosen to be the words that appear in the book. The frequency of each word has been counted, and, as in the conventional Huffman method, the codewords have been chosen to minimize the size of the compressed file for this model. In the terminology introduced on page 28, this is a static zero-order word-level model. The codewords are shown in decreasing order of length, and therefore in increasing order of word frequency—except that within each block of codes of the same length, words are ordered alphabetically rather than by frequency. The list begins with the thousands of words (and numbers) that appear only once. Words that occur only once in a text are called *hapax legomena*, a term that we will meet again on several occasions. Many of the words, such as *yopur* and *youmg*, occur only once because they are typographical errors. The numbers 100, 101, ... come from page numbers that are recorded in the file. (They start at 100 rather than a smaller number because words of the same codeword length are sorted in lexical—not numerical—order, so that 90, 91, ... appear later in the sequence.)

The table shows the codewords sorted from longest to shortest. An important feature of a canonical code like this is that when the codewords are sorted in lexical order—that is, when they are in the sequence they would be in if they were entries in a dictionary—they are also in order from the longest to the shortest codeword. On the other hand the code of Table 2.1 does not exhibit this property: although the codewords are ordered lexicographically, this does not result in them being sorted by length.

The key to using canonical codes efficiently is to notice that a word's encoding can be determined quickly from the length of its codeword, how far through the list it is, and the codeword for the first word of that length. For example, the word *said* is the 10th seven-bit codeword. Given this information and that the first seven-bit

Table 2.2 A canonical Huffman code.

Symbol	Codeword	
	Length	Bits
100	17	00000000000000000
101	17	00000000000000001
102	17	00000000000000010
103	17	00000000000000011
...	...	...
yopur	17	00001101010100100
youmg	17	00001101010100101
youthful	17	00001101010100110
zeed	17	00001101010100111
zephyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
...	...	...
were	8	10100110
which	8	10100111
as	7	1010100
at	7	1010101
for	7	1010110
had	7	1010111
he	7	1011000
her	7	1011001
his	7	1011010
it	7	1011011
s	7	1011100
said	7	1011101
she	7	1011110
that	7	1011111
with	7	1100000
you	7	1100001
I	6	110001
in	6	110010
was	6	110011
a	5	11010
and	5	11011
of	5	11100
to	5	11101
the	4	1111

codeword is 1010100, we can obtain the codeword for *said* by incrementing 1010100 nine times, or, more efficiently, adding nine to its binary representation.

Canonical codes come into their own for decoding because it is not necessary to store a decode tree. All that is required is a list of the symbols ordered according to the lexical order of the codewords, plus an array storing the first codeword of each distinct length. For example, with the code in Table 2.2, if the upcoming bits to be decoded are 1100000101 . . . , then the decoder will quickly determine that the next codeword must come after the first seven-bit word, *as* (1010100), and before the first six-bit word, *I* (110001). Therefore, the next seven bits are read (1100000), and the difference of this binary value from the first seven-bit value is calculated. In this example, the difference is 12, which means that the word is 12 positions after the word *as* in the list, so it must be *with*.

Compared with using a decoding tree, canonical coding is very direct. An explicit decode tree of the kind illustrated in Figure 2.5 requires a lot of space and is accessed randomly. With the canonical representation, only the first codeword of each length is accessed, plus one access to a lookup table to determine what the word is. Storing the first codeword of each length takes negligible space—the example in Table 2.2 has just 14 different lengths, ranging from 4 to 17. The list of symbols in lexical order of their codewords replaces the randomly accessed Huffman tree and is consulted only once for each symbol decoded.

Canonical Huffman codes

We now take a more detailed look at the implementation of a Huffman encoder and decoder pair. These details are quite intricate and can be skipped on the first reading, which is why this section is marked "optional" with a gray bar in the margin.

A canonical code is carefully structured to allow extremely fast decoding, with a memory requirement of only a few bytes per alphabet symbol. The code is called "canonical" (standardized) because much of the nondeterminism of normal Huffman codes is avoided. For example, in normal construction of the Huffman tree, some convention such as using a 0 bit to indicate a left branch and a 1 bit to indicate a right is assumed, and different choices lead to different, but equally valid, codeword assignments.

It is easiest to show this effect with an example. Consider the information shown in Table 2.3. The second column shows the observed symbol frequencies for the symbols in column 1. Three possible prefix-free codes for these symbols are shown in the columns headed *Code 1*, *Code 2*, and *Code 3*. Of course, in word-based codes there would be thousands of symbols instead of the six shown here, with frequencies ranging from 1 for *hapax legomena* to many thousands for common words (like *the*)—just as in Table 2.2.

The derivation of the first two codes in Table 2.3 is exactly as described in Figure 2.6. At each step, the two smallest items are extracted and coalesced, with one of the items having all its codewords prefixed by a 0 bit and the symbols represented by the other item being prefixed by a 1 bit. To obtain Code 1, for example, the sideways tree in Figure 2.8 was used, in which the convention is adopted that the upper

Table 2.3 Possible Huffman codes.

Symbol	Count	Code 1	Code 2	Code 3
a	10	000	111	000
b	11	001	110	001
c	12	100	011	010
d	13	101	010	011
e	22	01	10	10
f	23	11	00	11

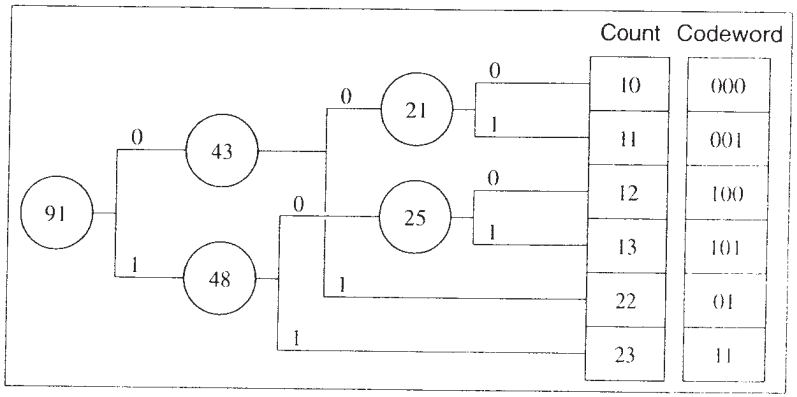

Figure 2.8 Constructing a Huffman code; "upper" edge is assigned a 0 bit.

branches are consistently assigned the 0 bit and the lower branches the 1 bit.

There is no particular requirement for this labeling convention, and Code 2 of Table 2.3 is formed by adopting the opposite rule, that upper edges are assigned 1 bits. In fact, this choice can be made at each of the internal nodes of the tree. A Huffman tree for an alphabet of n symbols has $n - 1$ internal nodes—for example, the tree of Figure 2.8 has five—and so there are 2^{n-1} equivalent, equally optimal, Huffman codes. Table 2.3 lists Code 1 and Code 2; to be exhaustive, it should really list Codes 1 through 32.

Code 3 in Table 2.3 is a little different. It is not just another relabeling of the edges in the Huffman tree. Although it is certainly an optimal prefix-free code for the symbols listed—it must be, because all of the codewords are the same length as they are in Codes 1 and 2—there is in fact no edge relabeling that will derive it. Strictly speaking, Code 3 is not a Huffman code at all because Huffman's algorithm is incapable of generating it, since it will always produce a tree with the shape of the

one in Figure 2.8. Rather than be pedantic, let us sidestep the issue of exactly what Huffman intended and define a Huffman code to be "any prefix-free assignment of codewords where the length of each code is equal to the depth of that symbol in a Huffman tree." That is, Huffman's algorithm should be used to calculate the *length* of each codeword, and then some bit pattern of this length can be assigned as the code.

Consider again Code 3 in Table 2.3. All the three-bit codewords form a neat sequence and the two-bit codewords likewise. Interpreted as integers, the three-bit codes are 0, 1, 2, and 3, and the two-bit codes are 2 and 3. More important, the first two bits of the three-bit codewords are all 00 or 01, that is, integer 0 or 1. This pattern makes decoding very easy. First, two bits from the input stream are read and interpreted as an integer. If their value is 2 or 3, a codeword has been identified, and the corresponding symbol can be output. If their value is 0 or 1, a third bit is appended, and the three bits are again interpreted as an integer and used to index a table to identify the correct symbol.

On such a small example this may not seem much of an improvement over the storage of a detailed code tree with left and right pointers and so on. However, an explicit decode tree raises two problems. First, it can consume a great deal of space. A Huffman tree for n symbols requires n leaf nodes and $n - 1$ internal nodes. Each leaf stores a pointer to a symbol and the information that it is in fact a leaf, and each internal node must store two pointers. In total, this structure requires around $4n$ words, and for an alphabet of one million symbols, 16 Mbytes of memory might be consumed—not even including the strings that are the actual symbols.

The second problem is that traversing a tree from root to leaf involves a lot of pointer chasing through memory, and the nodes accessed show little locality of reference. Each bit of compressed data that is decoded will require a new page of memory to be referenced, causing either page faults or—at best—numerous cache misses.

On the other hand, use of the canonical code means that decoding can be accomplished in a little over n words of memory, and with one random access per symbol rather than one random access per bit. This means that decompression is faster and requires substantially less memory than if an explicit code tree were used.

Let us now describe the use of a canonical code in detail. First, Huffman's algorithm is used to calculate, for each symbol i in the alphabet, the desired length l_i of the corresponding codeword. Exactly how this computation should be performed is described in the following subsection, "Computing Huffman code lengths." Next, the number of codewords of each possible length from 1 to *maxlength* is counted by passing over the set l_i and counting the frequency of each value. This allows the set of possible codes to be partitioned into groups, where each code within the group has the same length and the codewords form consecutive integers. For example, suppose there are to be four codewords of five bits, one of three bits, and three of two bits. At the completion of the partitioning step the five-bit codes will be 00000 through to 00011, the three-bit code will be 001, and the two-bit codes will be 01, 10, and 11. Finally, once the starting codeword for each length has been decided, it is straightforward to process the symbols one by one, simply assigning the next

To assign a canonical Huffman code to a set of symbols, supposing that symbol i is to be assigned a code of l_i bits, that no codeword is longer than *maxlength*, and that there are n distinct symbols,

1. For $l \leftarrow 1$ to *maxlength* do
   Set $numl[l] \leftarrow 0$.
   For $i \leftarrow 1$ to n do
   Set $numl[l_i] \leftarrow numl[l_i] + 1$.
   Number of codes of length l is stored in $numl[l]$.
2. Set $firstcode[maxlength] \leftarrow 0$.
   For $l \leftarrow maxlength - 1$ downto 1 do
   Set $firstcode[l] \leftarrow (firstcode[l + 1] + numl[l + 1])/2$.
   Integer for first code of length l is stored in $firstcode[l]$.
3. For $l \leftarrow 1$ to *maxlength* do
   Set $nextcode[l] \leftarrow firstcode[l]$.
4. For $i \leftarrow 1$ to n do
   (a) Set $codeword[i] \leftarrow nextcode[l_i]$.
   (b) Set $symbol[l_i, nextcode[l_i] - firstcode[l_i]] \leftarrow i$.
   (c) Set $nextcode[l_i] \leftarrow nextcode[l_i] + 1$.

   The rightmost l_i bits of the integer $codeword[i]$ are the code for symbol i.

Figure 2.9 Assigning a canonical Huffman code.

codeword of that length. This process is described in detail by the algorithm of Figure 2.9, in which $nextcode[l]$ is an integer storing the next codeword of length l that should be assigned. The array *symbol* is used during the decoding process, described below.

An example of the application of this algorithm is shown for a small collection of symbols in Table 2.4. The last two rows show the values calculated for the arrays *numl*, the number of codewords of each possible length, and *firstcode*, the integer value corresponding to the first codeword of each length. These values are used to assign the integer codes shown in the third column. The fourth column shows the corresponding bit patterns when the rightmost l_i bits of $codeword[i]$ are taken. These are the prefix-free codes that are generated.

The right-hand part of Table 2.4 shows the integer values that result when the first l bits of each codeword are extracted and considered as an integer, for $1 \leq l \leq 5 = maxlength$. Note that all the two-bit prefixes for codewords longer than two bits are less than $firstcode[2]$, all the three-bit prefixes for codewords longer than three bits are less than $firstcode[3]$, and all the four-bit prefixes for codewords longer than four bits are less than $firstcode[4]$. Indeed, all the five-bit prefixes for codewords

Table 2.4 Construction of a canonical Huffman code.

Symbol	Code length	*codeword*[*i*]	Bit pattern	*l*-bit prefix				
i	l_i			1	2	3	4	5
1	2	1	01	0	1			
2	5	0	00000	0	0	0	0	0
3	5	1	00001	0	0	0	0	1
4	3	1	001	0	0	1		
5	2	2	10	1	2			
6	5	2	00010	0	0	0	1	2
7	5	3	00011	0	0	0	1	3
8	2	3	11	1	3			
			numl[*l*]	0	3	1	0	4
			firstcode[*l*]	2	1	1	2	0

To decode a symbol represented in a canonical Huffman code,

1. Set $v \leftarrow$ *nextinputbit*().
   Set $l \leftarrow 1$.
2. While $v <$ *firstcode*[l] do
   (a) Set $v \leftarrow 2 * v +$ *nextinputbit*().
   (b) Set $l \leftarrow l + 1$.
   Integer v is now a legitimate code of l bits.
3. Return *symbol*[$l, v -$ *firstcode*[l]].
   This is the index of the decoded symbol.

Figure 2.10 Decoding using a canonical Huffman code.

longer than five bits are less than *firstcode*[5] too. (There are none of these in the example because no codewords are longer than five bits.)

This observation is the heart of the decoding process, and it is the reason why canonical codes are of such interest. Figure 2.10 shows how the decoding algorithm makes use of these relationships. The function *nextinputbit*() returns an integer 0 or 1, depending on the next input bit from the compressed stream that is being decoded. The array *symbol* is the mapping established by the code construction process of Figure 2.9. The space required by this array is discussed below.

For example, suppose the codewords of Table 2.4 are being used and that the bit-stream 00110 . . . is to be decoded. Variable v in Figure 2.10 is initialized to 0, which

as an integer is less than *firstcode*[1] = 2, so a second bit is appended to give v the value 00. This in turn is less than *firstcode*[2], and a third bit is added. This gives v the value 001 or integer 1, and now the test guarding the *while* loop (step 2) fails, since *firstcode*[3] = 1. Now a code has been parsed, and *symbol*[3, 0] is output—the correct symbol number 4. At the next decoding step v is initialized to 1, and the process repeats to form the code 10, which is identified as being *symbol*[2, 1], or symbol 5.

How much memory space is required during decoding? Surprisingly, the only large structure is the array *symbol*. The three other arrays—only one of which is required for decoding—require one word of memory for each possible code length, 1 . . . *maxlength*. For most implementations, a convenient value is *maxlength* = 32, so that integers can be used to store the codewords. (The restrictions imposed by this decision are discussed in Section 9.1, as is a mechanism for guaranteeing that no codeword exceeds such a limit.) Thus, only $3 \times 32 = 96$ words are required by the three small arrays together, irrespective of the size of the alphabet. Figures 2.9 and 2.10 use the array *symbol* as though it were two-dimensional. However, it can be implemented as a single one-dimensional array, in which the lth component is exactly *numl*[l] words long. The amount of space required for this array is $\sum_{l=1}^{maxlength} numl[l] = n$, which is the number of symbols. Thus, the mapping stored in *symbol* can be achieved in n words. In total, about $n + 100$ words of storage are required during decoding in addition to the space required to describe the n symbols. Note that it is not even necessary to store the length of symbol i since it is implicit in the sequential arrangement of codewords.

The implementation described in Figure 2.10 is also fast. One array lookup, one addition, and one integer comparison occur for each input bit, and the array lookup is to consecutive elements in a small array of just 32 words. If the processor is using memory caching, there will be no cache misses during the execution of this loop, and the logic is almost identical to that required to decode binary numbers of arbitrary length. Indeed, if the minimum codeword length is greater than one, there is no need to commence the linear search (step 2 of Figure 2.10) at one—it can instead be started at the minimum codeword length, with variable v assigned that number of input bits in a single operation. In this way binary decoding can be seen to be just a special case of canonical decoding.

Once a length and offset pair have been decoded, the array *symbol* is consulted, and a symbol number located. This might cause a cache miss. However, it takes place at most once per output symbol, and since the array *symbol* is organized with the frequent symbols (those with short codes) close together, it will on average occur even less often than that. Furthermore, in a word-level model each "symbol" represents several characters, so the time per character of output is relatively small.

Computing Huffman code lengths

An important area that has not been addressed so far in our discussion is determining the lengths for a Huffman code, which is required for canonical Huffman coding. In this section we will look at what this problem involves and then

describe the heap data structure, which is a useful data structure for Huffman coding. Next, we will describe a memory-efficient three-phase algorithm that uses a heap to determine the code lengths. Finally, two improvements are examined that can improve the speed of the algorithm and take advantage of the distribution of symbol frequencies found in practice.

The efficiency of computing Huffman code lengths affects only the cost of encoding, so it is not usually crucial, but elegant and efficient solutions are available and can make a big difference if the alphabet is large. The tree algorithm described earlier could be used to determine the lengths, and it has, over the years, proved to be an area rich in programming assignments for computer science students. However, for canonical coding there is no need to calculate the actual codewords, but just the code lengths. Also, in a word-based model, the alphabet is likely to contain thousands or even millions of symbols, and memory space for an explicit tree data structure might be a problem.

For these two reasons it is interesting to consider other techniques for calculating the code lengths of an optimal prefix-free coding. The algorithm presented in this section assumes that a file of n integers is supplied, where the ith integer is the number of times symbol i has appeared in the text. That is, if c_i is the ith value in the file, a code is to be built in which the probability of symbol i is $c_i/(\sum_{i=1}^{n} c_i)$. The output of the calculation is a second file, also of length n, in which the ith integer is the length in bits l_i of the code to be assigned to symbol i.

Since all the frequencies must be read into memory before any calculation can be performed, and it seems similarly impossible to write any of the lengths until all have been generated, a minimum requirement of $2n$ nodes with three four-byte fields (that is, $24n$ bytes) appears inevitable to store an explicit tree. On an alphabet of one million symbols, this corresponds to 24 Mbytes of memory. The process described in this section is substantially more economical and requires just $2n$ four-byte words (that is, $8n$ bytes), or 8 Mbytes for a model of one million symbols. The process is also efficient in terms of time.

The efficiency of the process comes from using a heap data structure, which can repeatedly find the smallest frequency very quickly with practically no memory overhead for storing the data structure. A *heap* is an implicit binary tree, with values stored at all leaves and internal nodes, and an ordering rule that requires values to be nonincreasing along each path from a leaf to the root. Figure 2.11a shows, in tree form, an example heap of 10 items. The immediate consequence of the ordering is that the smallest value is stored at the root of the tree. Although visualized as a binary tree, a heap is actually stored in an array, as shown in Figure 2.11b. The mapping of a heap to storage in memory is particularly elegant: the root is stored in location 1 of an array; the left child of the node stored in location i is stored in location $2i$; and the right child of location i is stored in location $2i + 1$. These rules mean that the parent of the node in position i occupies location $\lfloor i/2 \rfloor$.

A heap is very good for repeatedly finding and removing the smallest item. The trick at each stage is to remove the smallest item, replace it with another, and reinstate the heap order, all without spending too much effort. Suppose in the example that the smallest item, 2, is removed from location 1, and the last item in the

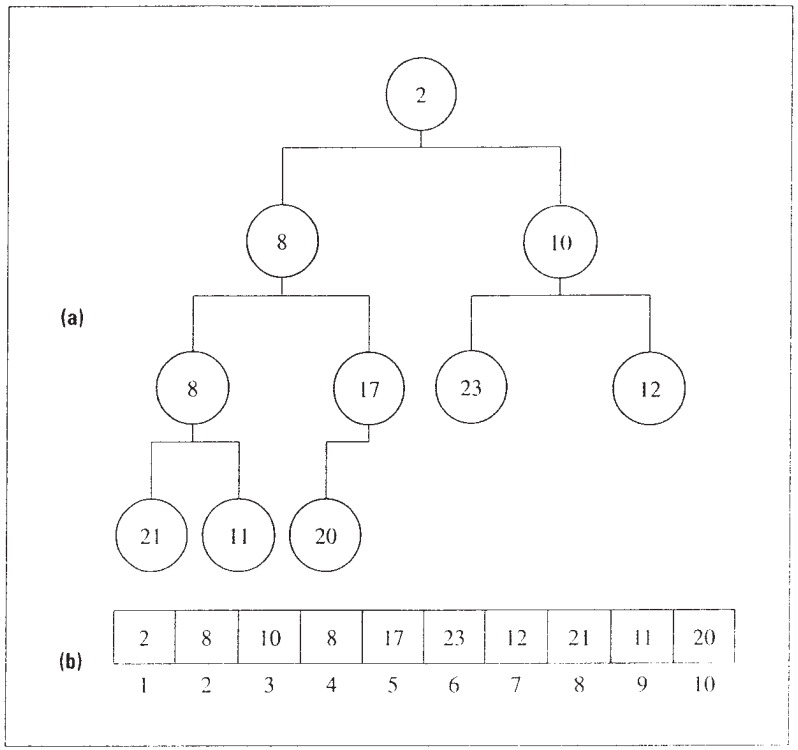

Figure 2.11 Heap data structure for finding the smallest value: (a) tree with heap ordering property; (b) implicit storage of tree in array.

array—item 20 in location 10—replaces it, as shown in Figure 2.12a for the heap of Figure 2.11a. To rebuild the ordering at the top level of the tree, item 8—the smaller of the two children of the root—must be swapped with 20. This localizes the order violation to the left subtree, and the process is repeated until heap order is restored. Figure 2.12b shows the reestablished heap after that item has been sifted back down the tree. The worst that can happen is that the item just promoted from the bottom leaf level returns all the way back to the bottom.

Using a heap, the smallest of n items can be found with one of these sifting operations. Since the depth of the tree is $\lceil \log n \rceil$, and just two comparisons are required at each level as an item is sifted down the tree, the total cost of finding the next smallest item is no more than $2\lceil \log n \rceil$ comparisons. Alterations to the weight associated with any item are handled in exactly the same way, and an item weight can be reassigned at a cost of at most $2\lceil \log n \rceil$ comparisons.

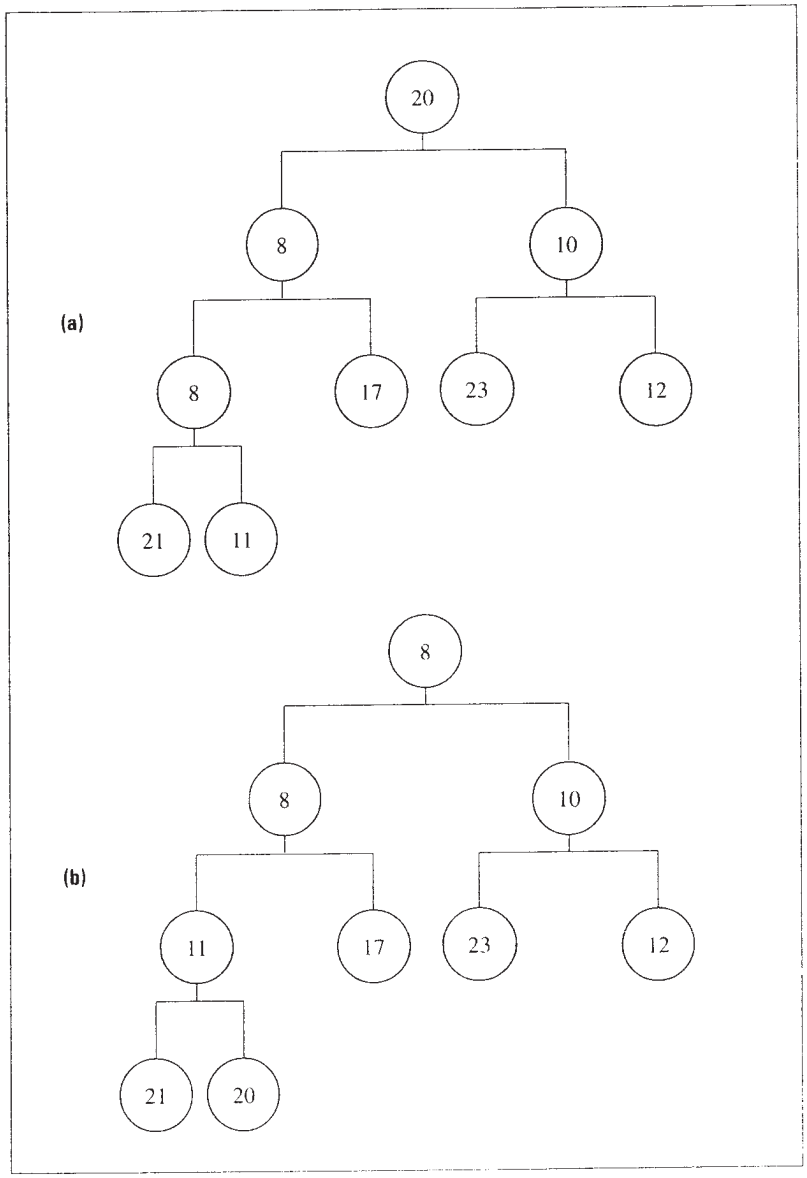

Figure 2.12 Finding the next smallest: (a) after 20 is swapped into the root; (b) after 20 is sifted down the tree.

Not included so far is the cost of constructing the initial heap from an unsorted list of n items. The algorithm for doing so makes use of the same sift operation, applying it to the root of every subtree in the heap from right to left and bottom to top. This stage requires approximately $2n$ comparisons.

The heap is used to calculate Huffman code lengths in the algorithm described in Figure 2.13. It uses an array of $2n$ entries to calculate the lengths for an alphabet of n symbols. Figure 2.14 illustrates how the array is used. Initially (Figure 2.14a), the frequencies of the symbols occupy the last n entries in the array. The first half of the array stores a heap that is used to efficiently locate the symbol with the lowest frequency for the node-combining step of Huffman's algorithm. As entries are removed from the heap, the space vacated is reused to store pointers that correspond to branches in the Huffman tree (Figure 2.14b). The frequency counts are also overwritten to store these pointers. At the end of processing, the heap contains only one item, and the rest of the array stores the shape of a Huffman tree. This is the arrangement shown in Figure 2.14c.

The details of the three main stages are as follows. In the first, the file of frequencies is read into locations $n + 1 \ldots 2n$ of an array A of $2n$ words. Each of the first n words in locations $1 \ldots n$ of A is set to point at the corresponding frequency in location $n + 1 \ldots 2n$. Next, the bottom half of A is turned into a heap using the method described above. The values used to drive the ordering in the heap are the frequencies pointed at by the items compared, not the items themselves. That is, the heap construction process must ensure that $A[A[i]] \leq A[A[2i]]$ and $A[A[i]] \leq A[A[2i + 1]]$ for all i in $1 \leq i \leq n/2$. Once the heap is constructed, $A[1]$ is the index in the range $n + 1 \leq A[1] \leq 2n$ of the smallest frequency in the second half of the array.

Now the second phase begins. As described earlier, the symbols, or aggregates of symbols, are considered in pairs, in each case taking the two smallest remaining items. In Figure 2.13, each of the items yet to be considered is represented in $A[1 \ldots h]$ by a pointer to its value, with the smallest always at the root of the heap. Thus, to find the two smallest, the root is taken from the heap, the leaf at $A[h]$ is moved into the root to fill the gap, h is decreased by one to indicate that position $A[h]$ is no longer included in the heap, and the heap is sifted. This brings the second smallest to the root of the heap. It is combined with the previously noted smallest, the weight of the combination is recorded at the empty location $A[h + 1]$, and $A[1]$ is set to point at this new aggregate value. To record the combination that took place, the two individual frequency counts for the two smallest items, $m1$ and $m2$—neither of which is required anymore—are changed into tree pointers that indicate the logical parent of these two nodes. This is the statement $A[m1] \leftarrow A[m2] \leftarrow h + 1$. Finally, the heap is sifted again to reestablish the invariant that the root of the heap indicates the smallest frequency count.

Figure 2.15 shows an example of this process at several stages. In Figure 2.15a, the last item in the heap is $A[h]$, and $A[m1]$ and $A[m2]$ are the two smallest entries in the heap, with values 4 and 5, respectively. Figure 2.15b shows the heap after the smallest item has been removed and the heap has been sifted. Now $m2$ is at the root of the heap, h has been decremented, and location $h + 1$ is empty. Next, the two

To calculate codeword lengths for a Huffman code,

1. /* Phase One */
   Create an array A of $2n$ words.
2. For $i \leftarrow 1$ to n do
   Read c_i, the ith integer in the input file,
   Set $A[n+i] \leftarrow c_i$ and $A[i] \leftarrow n+i$.
   $A[n+i]$ is now the frequency of symbol i.
   $A[i]$ points at $A[n+i]$.
3. Set $h \leftarrow n$.
   Build a min-heap out of $A[1 \ldots h]$.
   $A[1]$ stores $m1$ such that $A[m1] = \min\{A[n+1 \ldots 2n]\}$.
4. /* Phase Two */
   While $h > 1$ do
   (a) Set $m1 \leftarrow A[1]$, $A[1] \leftarrow A[h]$, and $h \leftarrow h-1$.
   $A[m1]$ is the current smallest.
   (b) Sift the heap $A[1 \ldots h]$ down from $A[1]$,
   Set $m2 \leftarrow A[1]$.
   $A[m2]$ is the second smallest.
   (c) Set $A[h+1] \leftarrow A[m1] + A[m2]$, $A[1] \leftarrow h+1$, and
   $A[m1] \leftarrow A[m2] \leftarrow h+1$.
   $A[h+1]$ now represents $(m1 + m2)$.
   (d) Sift the heap $A[1 \ldots h]$ down from $A[1]$.
   For $3 \leq i \leq 2n$, the value of $A[i]$ represents the parent of i in the Huffman tree, the leaves are in $A[n+1 \ldots 2n]$.
5. /* Phase Three */
   For $n+1 \leq i \leq 2n$ do
   (a) Set $d \leftarrow 0$ and $r \leftarrow i$.
   (b) While $r > 2$ do
   Set $d \leftarrow d+1$ and $r \leftarrow A[r]$.
   (c) Set $A[i] \leftarrow d$.
6. For $i \leftarrow 1$ to n do
   Write $A[n+i]$ as l_i, the length of the code for symbol i.

Figure 2.13 Calculating Huffman code lengths.

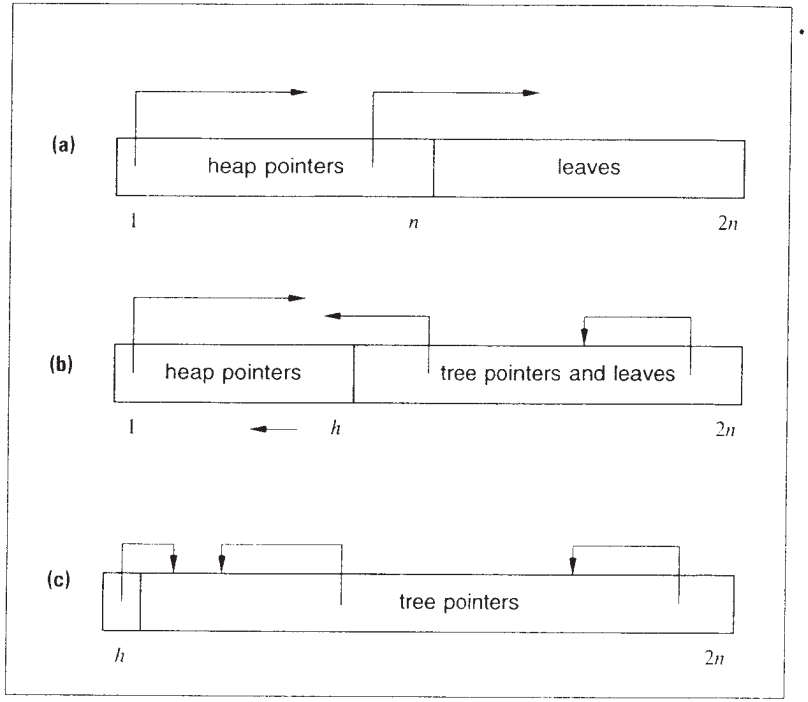

Figure 2.14 Use of an array to generate Huffman code lengths: (a) two sections, heap on left, frequencies on right; (b) heap shrinking, and pointers replacing frequencies; (c) one item remaining in heap.

smallest items are combined. Their total frequency is 9, which is stored at location $h+1$. To get this new aggregate into the heap, $A[1]$ is made to point at location $h+1$, and, to note the composition of the aggregate, both $A[m1]$ and $A[m2]$ also point at $h+1$—the arrangement shown in Figure 2.15c. Finally, the heap is sifted, and the next smallest item moves up to position $A[1]$. In Figure 2.15d, this is the last item, with frequency 7. It will be the first element in the next combination and may, perhaps, be merged with the aggregate just formed.

At the completion of each such step, two items have been combined into one. As a result, the heap contains one less item, and one item has been added to the superstructure of internal tree nodes that records what combinations took place. After $n-1$ iterations, a single aggregate remains in the heap. The frequency of this item is stored in $A[2]$, and $A[1]$ must contain 2 since there is just one item in the heap. All the other $2n-2$ values in A contain parent pointers. To find the depth in the tree of any particular leaf, we can now simply start at that leaf and count how

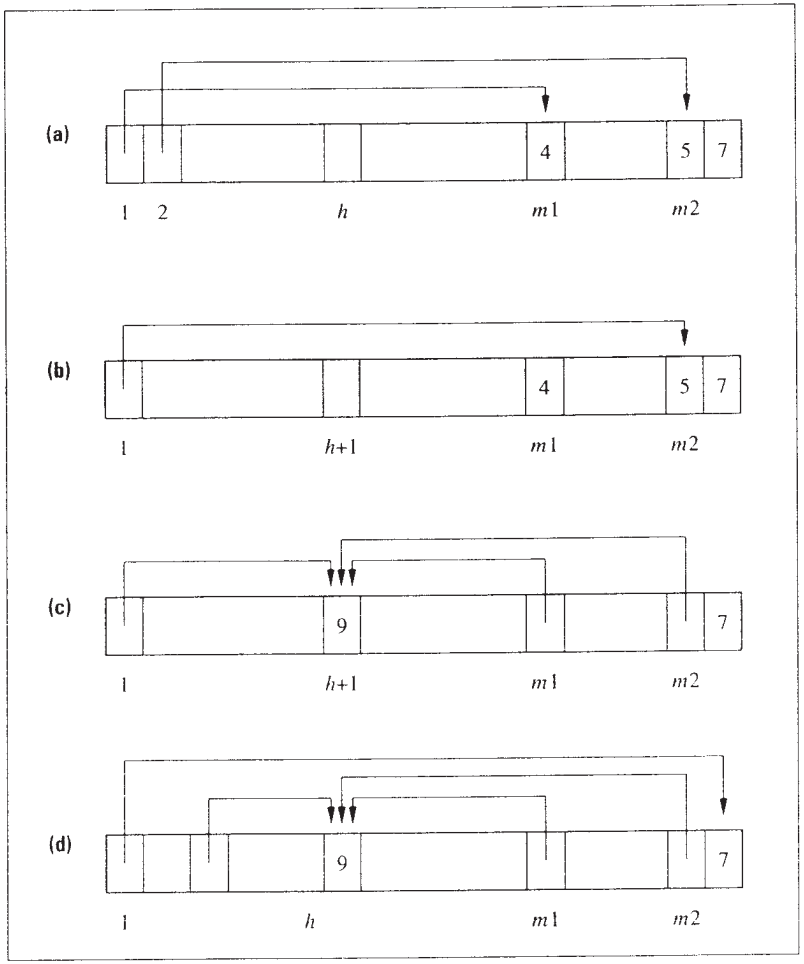

Figure 2.15 Building a Huffman code: (a) before smallest is removed from heap; (b) before second smallest is removed from heap; (c) after smallest and second smallest have been removed and combined; (d) after heap has been sifted.

many parent pointers must be followed to get to location 2, the root of the entire code tree.

This is the third phase of the algorithm. Starting from each leaf in turn—recall that the symbols of the alphabet are indicated by positions $n + 1 \ldots 2n$—the parent pointers are followed and the depth of that particular leaf stored in place of the

5′. /* Revised Phase Three */
Set $A[2] \leftarrow 0$.
For $i \leftarrow 3$ to $2n$ do
Set $A[i] \leftarrow A[A[i]] + 1$.

Figure 2.16 Improved counting of leaf depths.

parent pointer. For example, in Figure 2.15, the paths from both $m1$ and $m2$ will pass through $h + 1$, and one bit is counted toward their code lengths for that part of their path. Thereafter, both traversals follow the same path to the root.

When this process is finished, the array $A[n + 1 \ldots 2n]$ stores the desired Huffman code lengths. These can be written to disk or used to construct a code. If it were more convenient, the values could be written as they are calculated in phase three.

Now consider the running time of the algorithm in Figure 2.13. The first phase is linear and takes n steps. Even when $n = 1{,}000{,}000$, this represents just a few seconds of computation. In the second phase, a heap of at most n items is sifted about $2n$ times. Each sift operation takes $\lceil \log h \rceil \leq \lceil \log n \rceil$ iterations, and each iteration takes a constant number of steps. In total, the second phase will take $kn \log n$ steps, where k is a small constant. Typical workstations execute about one million steps per second, and so with $n = 1{,}000{,}000$ the time taken by the second phase is at most a few tens of seconds.

Analysis of the third phase is a little more problematic. It would appear that one loop iteration is required for each bit of code length for each symbol, so we should calculate how many bits there could be. In the fastest case, the distribution of symbol frequencies will be uniform, and all codes will be approximately $\log n$ bits long. The total counting time will thus be $n \log n$ steps. Another way to analyze this is simply to assume in the implementation that no code will be longer than, say, 32 bits—this automatically provides an upper bound on the number of loop iterations.

In the worst case, however, it is possible for a pathological distribution of frequencies to drive the code for the ith symbol to i bits. This in turn means that the cost of computing the code lengths is proportional to $\sum_{i=1}^{n} i \approx n^2/2$, and this dominates the time required during phases one and two. (In fact, if the ith symbol received a code i bits long in any nontrivial application, there would probably be serious repercussions elsewhere in the system.) To avoid this blow-out in computation time during the third phase of the algorithm, a further refinement is necessary. Figure 2.16, which replaces step 5 of Figure 2.13, describes such an alternative. The modified phase three is based on the observation that all the pointers in the array produced by phase two point from right to left, so a labeling process from left to right will label each parent node before either of its children is encountered. Hence, if the root in $A[2]$ is labeled with a depth of 0, each following item can, in turn, be labeled with depth 1 greater than the depth of its parent—$A[A[i]] + 1$ in Figure 2.16. The resulting algorithm takes some effort to understand but is very simple

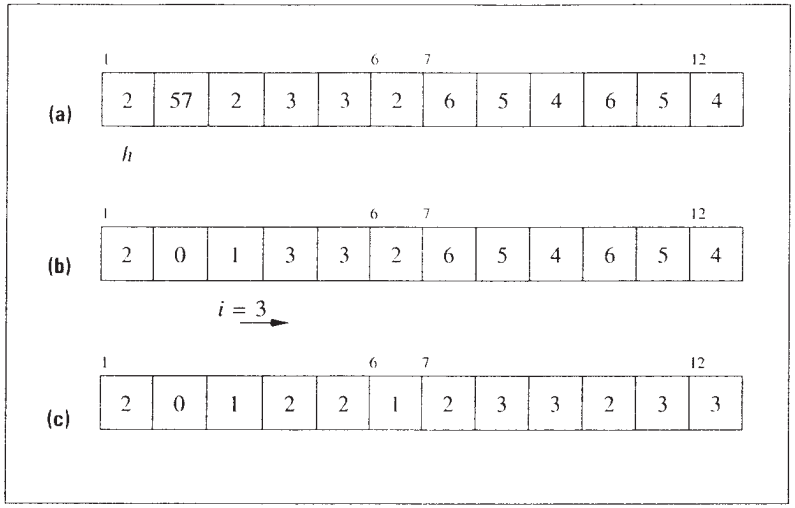

Figure 2.17 Traversing the tree: (a) before traversal; (b) after processing $i = 3$; (c) after the traversal is complete.

to code and executes extremely quickly. It takes time proportional to the number of symbols in the alphabet and even for an alphabet of one million symbols requires just a few seconds.

Figure 2.17 shows an example of the processing carried out by the revised depth-counting mechanism. Figure 2.17a shows one possible state of the array A immediately prior to step 5. The total count of all symbols is recorded in $A[2]$ and is 57. The combination $h = 1$ and $A[1] = 2$ indicates that this is the only item still in the heap. The other 10 locations indicate parent nodes for the six symbols represented in the leaves in locations 7...12. Figure 2.17b shows the situation when $i = 3$ has been completed, and Figure 2.17c shows the arrangement when the loop of Figure 2.16 has terminated. In Figure 2.17c, $A[i + 6]$ for $1 \leq i \leq 6$ records the number of bits to be allocated to the Huffman code of symbol i.

There is a further way that the code calculation process can be improved. For some alphabets, particularly very large ones such as the one shown in Table 2.2, it is common—indeed, all but inevitable—for many symbols to have the same frequency. In particular, *hapax legomena* typically account for as many as 50 percent of the distinct words, and in this case half of the frequency counts will be 1. If 50 percent of the symbols have a count of 1, then half the Huffman aggregation operations will involve joining two symbols with a count of 1. The repetition of counts can be exploited using an algorithm that represents the list of symbol frequencies using run-length coding, where a sequence of identical counts are represented by just two numbers, one giving the count and the other giving the number of repetitions (Moffat and Turpin 1998). The combining rule—which requires aggregating the smallest two frequencies—becomes rather more complex, but a substantial reduction in time and space is possible, provided only that the list of input symbol frequencies is already in sorted order. Run-length coding can also be applied to the output (such as the lengths shown in Table 2.2), which also inevitably contains long runs of identical code lengths. Analysis of the run-length method for calculating Huffman codes shows that the time taken is proportional to $r + r \log(n/r)$ for an alphabet of n symbols in which there are r distinct symbol frequencies (Moffat and Turpin 1998). Furthermore, the space required is proprtional to the running time. For typical large-alphabet values n = 1,000,000 and r = 10,000, these bounds represent a considerable saving in resources, and the mechanism is very useful in situations when the $n \log n$ cost of sorting the symbol frequencies can be either avoided or amortized over more than one calculation.

Summary

Now, at last, we have seen how to generate and use Huffman codes efficiently for alphabets containing thousands of symbols. This is an essential prerequisite for virtually all large-scale information retrieval systems that use compression. Although Huffman codes are standard fodder for undergraduate courses in computer science, the methods described in the usual textbooks do not scale up effectively. Considerable savings can be made in both the time and the main memory required for decoding by using canonical Huffman codes instead of the standard decoding tree, and decoding is one of the principal ongoing operations in compressed information retrieval systems. Although encoding is done far less often—just once every time the database is reconstructed—it is still necessary to accomplish it within reasonable resource constraints, and again substantial savings in both time and space can be reaped by using the nonstandard, and by no means obvious, techniques that we have described.

2.4 Arithmetic coding

Arithmetic coding is a technique that has made it possible to obtain excellent compression using sophisticated models. Its principal strength is that it can code arbitrarily close to the entropy. It has been shown that it is not possible to code better than the entropy on average (Shannon 1948), so in this sense arithmetic coding is optimal.

To compare arithmetic coding with Huffman coding, suppose symbols from a binary alphabet are to be coded, where the symbols have probabilities of 0.99 and 0.01. The information content of a symbol s with probability $\Pr[s]$ is $-\log \Pr[s]$ bits, so the symbol with probability 99 percent can be represented using arithmetic coding in just under 0.015 bits. In contrast, a Huffman coder must use at least one bit per symbol. The only way to prevent this situation with Huffman coding is to "block" several symbols together at a time. A block is treated as the symbol to be coded, so that the per-symbol inefficiency is now spread over the whole block.

Blocking is difficult to implement because there must be a block for every possible combination of symbols, so the number of blocks increases exponentially with their length; this effect is exacerbated if consecutive symbols come from different alphabets, as is the case in the high-performance schemes that are described in the next few sections.

The compression advantage of arithmetic coding is most apparent in situations like the previous example, in which one symbol has a particularly high probability. More generally, it has been shown that the inefficiency of Huffman coding is bounded above by

$$\Pr[s_1] + \log \frac{2 \log e}{e} \approx \Pr[s_1] + 0.086$$

bits per symbol, where s_1 is the most likely symbol (Gallager 1978). For the extreme example described above, the inefficiency is thus 1.076 bits per symbol at most (a bound that can trivially be reduced to one bit per symbol), which is many times higher than the entropy of the distribution. On the other hand, for English text compressed using a zero-order character-level model, the entropy is about five bits per character, and the most common character is usually the space character, with a probability of about 0.18. With such a model the inefficiency is less than $0.266/5 \approx 5.3$ percent. In many compression applications $\Pr[s_1]$ is even smaller, and the inefficiency becomes almost negligible. Moreover, this is an *upper* bound, and in practice the inefficiency is often significantly less than the bound might indicate. On the other hand, when images are being compressed, it is common to deal with two-symbol alphabets and highly skewed probabilities, and in these cases arithmetic coding is essential unless the symbol alphabet is altered using a technique like blocking.

Another advantage of arithmetic coding over Huffman coding is that arithmetic coding calculates the representation on the fly, so less primary memory is required for operation and adaptation is more readily accommodated. Canonical Huffman codes are also fast, but they are suitable only if static or semi-static modeling is being used. Arithmetic coding is particularly suitable as the coder for high-performance adaptive models, where very high probabilities (confident predictions) are occurring, where many different probability distributions are stored in the model, and where each symbol is coded as the culmination of a sequence of lower-level decisions.

One disadvantage of arithmetic coding is that it is slower than Huffman coding, especially in static or semi-static applications. Also, the nature of the output means that it is not easy to start decoding in the middle of a compressed stream. This contrasts with Huffman coding, in which it is possible to index "starting points" in a compressed text if the model is static. In full-text retrieval, both speed and random access are important, so arithmetic coding is not likely to be appropriate. Furthermore, for the types of models used to compress full-text systems, Huffman coding gives compression that is practically as good as arithmetic coding. Thus, in large collections of text and images, Huffman coding is likely to be used for the text and arithmetic coding for the images.

How arithmetic coding works

Arithmetic coding can be somewhat difficult to grasp. Though it is interesting, understanding it is not essential. Source code is readily available for efficient arithmetic coders (see *www.cs.mu.oz.au/mg/*), which can be plugged into a compression system to perform the coding part. Only the interface to the coder needs to be understood in order to use it.

The output of an arithmetic coder, like that of any other coder, is a stream of bits. However, it is easier to describe arithmetic coding if we prefix the stream of bits with 0. and think of the output as a fractional binary number between 0 and 1. In the following example, the output stream is 1010001111, but it will be treated as the binary fraction 0.1010001111. In fact, for the sake of readability, the number will be shown as a decimal value (0.64) rather than as binary, and some possible efficiencies will be overlooked initially.

As an example we will compress the string *bccb* from the ternary alphabet $\{a, b, c\}$. We will use an adaptive zero-order model and deal with the zero-frequency problem by initializing all character counts to one.

When the first b is to be coded, all three symbols have an estimated probability of $1/3$. An arithmetic coder stores two numbers, *low* and *high*, which represent a subinterval of the range 0 to 1. Initially, *low* = 0 and *high* = 1. The range between *low* and *high* is divided according to the probability distribution about to be coded. Figure 2.18a shows the initial interval, from 0 to 1, with a third of it allocated to each symbol. The arithmetic coding step simply involves narrowing the interval to the one corresponding to the character to be coded. Thus, because the first symbol is a b, the new values are *low* = 0.3333 and *high* = 0.6667 (working to four decimal places). Before coding the second character, c, the probability distribution adapts because of the b that has already been seen, so $\Pr[a] = 1/4, \Pr[b] = 2/4$, and $\Pr[c] = 1/4$. The new interval is now divided up in these proportions, as shown in Figure 2.18b, and coding of the c involves changing the interval so that it is from *low* = 0.5834 to *high* = 0.6667. Coding continues as shown in Figure 2.18c, and at the end the interval extends from *low* = 0.6390 to *high* = 0.6501.

At this point, the encoder transmits the code by sending any value in the range from *low* to *high*. In the example, the value 0.64 would be suitable. The decoder simulates what the encoder must have been doing. It begins with *low* = 0 and *high* = 1 and divides the interval as shown in Figure 2.18a. The transmitted number, 0.64, falls in the part of the range corresponding to the symbol b, so b must have been the first input symbol. The decoder then calculates that the range should be changed to *low* = 0.3333 and *high* = 0.6667, and, because the first symbol is now known to be b, the new probability allocation where $\Pr[b] = 2/4$ (shown in Figure 2.18b) can be calculated. Decoding proceeds along these lines until the entire string has been reconstructed.

A general algorithm for calculating the range during encoding is shown in Figure 2.19, and Figure 2.20 shows how a symbol is decoded and the range is updated afterward. Since arithmetic coding deals with ranges of probabilities, it is usual for a model to supply *cumulative probabilities* to the encoder and decoder; this makes

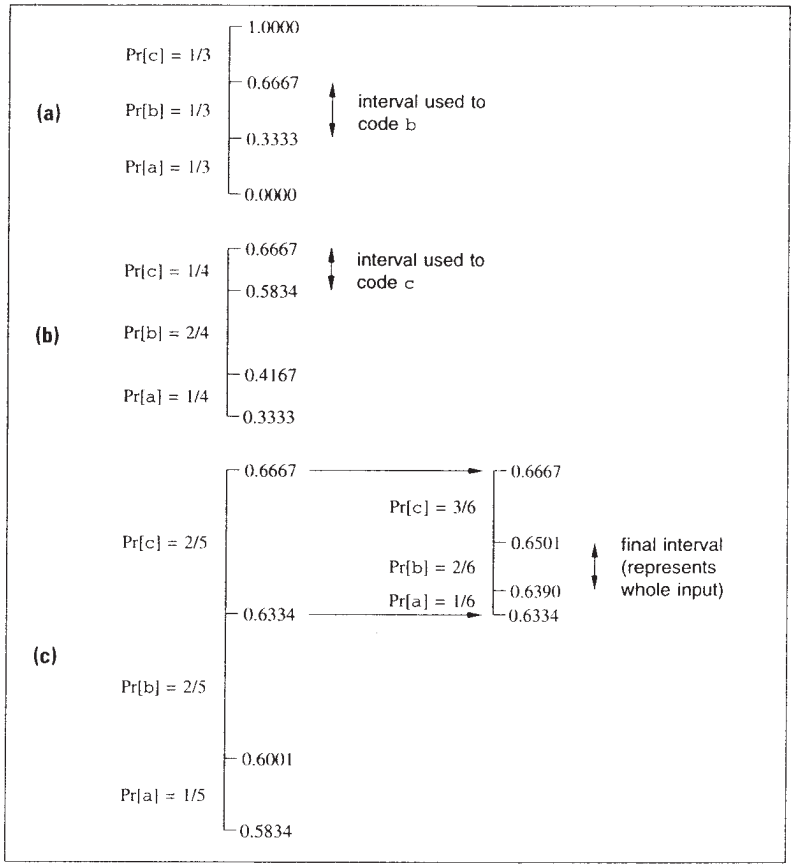

Figure 2.18 Arithmetic coding example for the string *bccb*: (a) first symbol; (b) second symbol; (c) third and fourth symbols.

the first steps of each algorithm easy to implement. For static and semi-static coding, the cumulative probabilities can be stored in an array. The encoder accesses the array by symbol number, and the decoder accesses it by binary search or through the use of a lookup table. Adaptive coding requires a more sophisticated structure so that cumulative probabilities can be adjusted on the fly. More commonly, frequency counts are kept for each symbol, and these are normalized to probabilities as an integral part of the arithmetic coding process.

Informally, compression is achieved because high-probability events do not decrease the interval from *low* to *high* very much, while low-probability events result in a much smaller next interval. A small final interval requires many digits (bits)

To code symbol s, where it is assumed that symbols are numbered from 1 to n, and that symbol i has probability $\Pr[i]$,

1. Set $low\_bound \leftarrow \sum_{i=1}^{s-1} \Pr[i]$.
2. Set $high\_bound \leftarrow \sum_{i=1}^{s} \Pr[i]$.
3. Set $range \leftarrow high - low$.
4. Set $high \leftarrow low + range \times high\_bound$.
5. Set $low \leftarrow low + range \times low\_bound$.

Figure 2.19 The arithmetic encoding step.

To decode a symbol, assuming that the symbols are numbered from 1 to n, that symbol i has probability $\Pr[i]$, and that *value* is the arithmetic code to be processed,

1. Find s such that
$$\sum_{i=1}^{s-1} \Pr[i] \leq (value - low)/(high - low) < \sum_{i=1}^{s} \Pr[i].$$
2. Perform the range-narrowing steps described in Figure 2.19, exactly as if symbol s is being *en*coded.
3. Return symbol s.

Figure 2.20 The arithmetic decoding step.

to specify a number that is guaranteed to be within the interval—for example, at least six decimal digits are needed to specify a number that is between 0.378232 and 0.378238. In contrast, a large interval requires few digits—for example, any number beginning with 0.4 is between 0.378232 and 0.578238, so the decoder only needs to know that the number begins with 0.4. The number of digits necessary is proportional to the negative logarithm of the size of the interval, just as the number of digits needed to represent numbers on the other side of the decimal point is proportional to the positive logarithm of the number. The size of the final interval is the product of the probabilities of the symbols coded, and so the logarithm of this quantity is the same as the sum of the logs of the individual probabilities. Therefore, a symbol s of probability $\Pr[s]$ contributes $-\log \Pr[s]$ bits to the output, which is equal to the symbol's information content and results in a code that is identical to the bound given by the entropy formula. This is why arithmetic coding produces a near-optimal number of output bits and is, in effect, capable of coding

high-probability symbols in just a fraction of a bit. In practice, arithmetic coding is not exactly optimal because of the use of limited precision arithmetic and because a whole number of bits (or even bytes) must eventually be transmitted, but it is extremely close.

As we have described arithmetic coding so far, nothing appears in the output until all the encoding has been completed. In practice, it is possible to output bits *during* encoding, which avoids having to work with higher and higher precision numbers. The trick is to observe that when *low* and *high* are close in value, they have a common prefix. For example, after the third character has been coded (Figure 2.18c), the range is *low* = 0.6334 and *high* = 0.6667. Both are prefixed with 0.6, so no matter what happens to the range from there on, the first symbol transmitted must be a 6 (or would be, if the encoder were working in decimal). Thus, an output symbol can be transmitted at this point and then removed from *low* and *high* without any effect on the remaining calculations. In the example, the first decimal digit (which is 6) can be removed from *low* and *high*, changing them to 0.334 and 0.667, respectively. Working with these new values, the top of the *a* interval is now calculated to be 0.390 instead of 0.6390. The same output is being constructed, but the need for increasing precision has been avoided, and the output is generated incrementally, rather than having to wait until the end of coding.

As mentioned earlier, the final output value is really transmitted as a binary fractional number, and the 0. on the front is unnecessary because the decoder knows that it will appear. The values *low* and *high* are stored in binary; in fact, with suitable scaling they can be stored as integers rather than as real numbers. Working with finite precision causes compression to be a little worse than the entropy bound, but 16- or 32-bit precision usually degrades compression by an insignificant amount. Witten, Neal, and Cleary (1987), in a seminal paper, describe a general-purpose arithmetic coder based upon the use of integer arithmetic, and they also give analysis to bound the inefficiency arising from calculation errors. So although the example used unlimited-precision floating-point decimal numbers, in practice arithmetic coding can be implemented using fixed-precision integers, and the output is a stream of bits. Each symbol coded requires just a few arithmetic operations in the arithmetic coder. In the following section, we look more carefully at exactly how many operations are necessary.

Implementing arithmetic coding

This section describes a practical implementation of arithmetic coding. As well as examining how the interface between the model and coder works in practice, we will look at the special cases where the alphabet is very small (perhaps just two symbols) or very large (thousands or millions of symbols).

The interface between an arithmetic coder and a model is a little unusual because of the way that arithmetic coding works. To divide up the range, the coder needs to know which part of it corresponds to the symbol to be coded. We now describe three routines that are found in the mg source code: *arithmetic_encode*, *arithmetic_decode_target*, and *arithmetic_decode*.

The interface to the encoder is through the single routine *arithmetic_encode*. Instead of using probabilities, the coder is passed integers related to the counts of symbols. The procedure *arithmetic_encode* has three parameters: *low_count*, *high_count*, and *total*. The *total* parameter is the sum of all the counts for all possible symbols. In the example in Figure 2.18, for the coding of the fourth symbol, the total would be 6. The *low_count* parameter is the sum of the counts of all symbols prior (in the alphabet ordering) to the one to be encoded, so that the cumulative probability *low_bound* in Figure 2.19 is then easily calculated as *low_count*/*total*. In the example, the fourth character coded was *b*, and *a* is the only one that comes before it in the alphabet, so *low_count* = 1 (the count for *a*). The *high_count* parameter is the top of the range corresponding to *b*—that is, *low_count* plus the count of *b*. The example would have *high_count* = 3. Thus the *b* would be coded by calling *arithmetic_encode*(1, 3, 6). If the fourth character had been *a* or *c*, then it would have been coded by calling *arithmetic_encode*(0, 1, 6) or *arithmetic_encode*(3, 6, 6), respectively.

Two routines are required to interface to the arithmetic decoder. The first, function *arithmetic_decode_target*, has *total* as a parameter, obtained from the decoder model. It returns a number corresponding to the encoded symbol, in the range from 0 to *total* − 1. The range is closed below and open at the top, so the values "in the range" of 1 to 3 (for the example symbol *b*) are 1 and 2. If either of these targets is received, then a *b* is decoded. Similarly, *a* has a range of 1, so it has only one target value, which is 0; a target of 3, 4, or 5 indicates that a *c* is to be decoded. The decoder must use the model to determine to which symbol the target value belongs, and then the full range must be adjusted by a call to *arithmetic_decode*. The parameters for *arithmetic_decode* should be identical to the ones that were given to *arithmetic_encode*; in the example, the call would be *arithmetic_decode*(1, 3, 6). Use of the identical parameters enables the decoder to narrow the range correctly.

There are two important issues surrounding implementation of arithmetic coding. One is how the cumulative counts can be maintained efficiently, since it would usually be too slow to perform a summation every time one is needed. We return to this in the following subsection, "Maintaining cumulative counts."

The other issue, which we address immediately, is exactly how the arithmetic should be performed to make the coder work as fast as possible.

The key operations performed during arithmetic coding are the operations to adjust the range as sketched in Figure 2.19. In practice, the cumulative probabilities are represented using integers (*low_count*, *high_count*, and *total*), so, for example, *low_bound* is actually *low_count*/*total*. This means that step 5 of Figure 2.19 is now

$$low \leftarrow low + range \times low\_count/total.$$

To perform this calculation accurately, double-precision integer arithmetic is needed for the intermediate results of the multiplication/division component. It turns out that by choosing the ordering of the calculations carefully, and allowing a little loss of precision, it is possible to simultaneously avoid the use of high-precision arithmetic and reduce the number of operations required. The exact details of the trade-off will depend on the architecture of the machine; if high-precision integer

To *arithmetic_encode*(*low_count*, *high_count*, *total*) using low-precision arithmetic and assuming that the state variables *low* and *range* are to be modified to reflect the new range,

1. Set $r \leftarrow$ *range* div *total*.
2. Set *low* $\leftarrow$ *low* + $r \times$ *low_count*.
3. If *high_count* $<$ *total* then
   Set *range* $\leftarrow r \times$ (*high_count* $-$ *low_count*)
   else
   Set *range* $\leftarrow$ *range* $- r \times$ *low_count*.
4. Renormalize *low* and *range*.

To *arithmetic_decode_target*(*total*), returning an integer in the range [0, *total*), where D buffers the compressed bits being decoded and corresponds to *value* $-$ *low* in Figure 2.20,

1. Set $r \leftarrow$ *range* div *total*.
2. Return $\min\{total - 1, D \text{ div } r\}$.

To *arithmetic_decode*(*low_count*, *high_count*, *total*) using low-precision arithmetic, assuming that r has been set by *arithmetic_decode_target* and that the state variables D and *range* are to be modified to reflect the new range,

1. Set $D \leftarrow D - r \times$ *low_count*.
2. If *high_count* $<$ *total* then
   Set *range* $\leftarrow r \times$ (*high_count* $-$ *low_count*)
   else
   Set *range* $\leftarrow$ *range* $- r \times$ *low_count*.
3. Renormalize *range* and load new bits into D to match the renormalization.

Figure 2.21 An efficient arithmetic coding implementation.

multiplications are fast, then it may not be so beneficial. The cost of using approximations will be a small loss in compression performance, but for the method shown below, this generally amounts to a fraction of a percent in practice and is a worthwhile optimization.

An effective algorithm that achieves this is shown in Figure 2.21. A significant difference from the version described in Figures 2.19 and 2.20 is that instead of representing the current range using its boundaries *low* and *high*, it is represented using

the lower boundary *low* and *range*, the size of the current interval. The speed efficiency is gained by using the temporary variable r, which can be stored to relatively low precision. In fact, because r is typically a small integer, it can be more efficient to perform the multiplicative operations in which it is involved using shifting and adding, which results in a further speed gain on some architectures.

Notice that a special case is made when *high_count* is at the top of the range. This serves two purposes. First, the separate calculation ensures that the new high point is the same as before; if it were rounded down because of the truncation when r is calculated, some compression inefficiency would be introduced unnecessarily. Second, the calculation $r \times$ *low_count* has already been performed in the second step and need not be repeated here, saving one multiplication. This can happen relatively often if the most probable symbol is allocated the top part of the range.

The renormalization step involves transmitting bits from the left-hand side of *low*, and shifting *range* left, until excessive leading 0 bits at the start of *range* have been eliminated.

The decoder mirrors the state of the encoder, except that instead of storing *low*, it stores the difference D between *low* and the value in the input. This saves some operations in the decoder by simplifying the renormalization step—only D needs to be renormalized instead of both *low* and the incoming value.

This approximate method is from Moffat, Neal, and Witten (1998), who evaluate a number of improvements to arithmetic coding. Source code for their methods is available by ftp from *ftp://munnari.oz.au/pub/arith_coder/*; the original 1987 implementation of Witten, Neal, and Cleary is available at *ftp://ftp.cpsc.ucalgary.ca/pub/projects/ar.cod/cacm-87.shar*. Other enhancements improve the speed by using shift/add operations instead of multiplications and divisions (Rissanen and Mohiuddin 1989; Chevion, Karnin, and Walach 1991; Feygin, Gulak, and Chow 1994; Stuiver and Moffat 1998) or looking up approximate arithmetic calculations in a precomputed table (Howard and Vitter 1993a). Approximating the calculations leads to higher compression throughput but is at the cost of degraded compression effectiveness because less care is taken with the accuracy of the arithmetic.

One special case where approximate arithmetic coding has been used very effectively is for binary input alphabets. In this case there is only one point between *high* and *low* to be calculated, and the order of the two symbols in the range can easily be swapped so that the most probable one is at the top. The split point is therefore guaranteed to be below halfway. If it is approximated with a power of two, the multiplication can be replaced with a single shift operation, and because the split point is less than 0.5 (and often a lot less than this), the loss of compression efficiency can be bounded and is usually very small. This kind of approximation is exploited by the IBM Q-coder, which is used for various applications including the JBIG and JPEG image compression standards described in Chapter 6.

Maintaining cumulative counts

For nonbinary alphabets, the speed of arithmetic coding is strongly affected by how quickly the cumulative counts can be calculated, so it is important to use a data

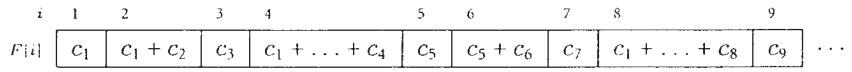

Figure 2.22 Array storing implicit tree for cumulative count calculation.

structure that makes it easy to calculate the cumulative count for a symbol and easy to search the cumulative counts for a target. If the probability distribution is very skew, so that only a small subset of the possible symbols are being regularly used (or if the alphabet contains in total only a few symbols), then a move-to-front list is suitable. This uses a linear search, but the list being searched is reorganized by moving an element to the head of the list when it is accessed. If only a few symbols are being coded, they will quickly end up near the front of the list, and most accesses will only require a short search. This is the case for a simple zero-order character-level model, where for typical English text only about 10 elements in a move-to-front list are probed on average for each symbol coded.

If the alphabet is used sparsely (which occurs in high-order contexts for coders such as the PPM method, described in Section 2.5), even a simple linked list may be suitable since only a few counts will be kept for the few symbols that are observed in each context. For binary alphabets—one of the main applications of arithmetic coding—the problem of calculating cumulative counts is trivial.

For larger alphabets, such as the set of words in a document, the probability distribution will be less skew and a better structure is needed. A particularly elegant structure for this situation has been developed that uses an implicit tree, where each node contains the cumulative sum of a specific range of counts (Fenwick 1994). The tree can be traversed from the root to a symbol's node in such a way that the selection of ranges encountered on the path will exactly cover the entire range up to the symbol for which a cumulative count is required. For an alphabet of n symbols, the maximum depth of the tree is $\log_2 n$, so the cumulative count can be determined in logarithmic time. The tree is implicit and is actually stored without pointers in an array. The fast Huffman implementation described earlier also uses an implicit tree (a heap) but with different rules for the structure. Like the heap, the implicit tree used for cumulative counts is very efficient; only n integers are stored to represent n counts, so there are no storage overheads.

Figure 2.22 shows the beginning of the array F that is used to store the implicit tree. Each entry $F[i]$ in the array contains the sum of the counts for a contiguous group of symbols including c_i, the frequency of symbol i; from these the required cumulative sums can be readily calculated.

To calculate the cumulative count for symbol i, we start at entry $F[i]$ and work backward toward $F[1]$, adding a selected subset of the array entries. For example, for $i = 11$ the entries summed are $F[11]$, $F[10]$, and $F[8]$. The set of entries to visit is easily determined from the binary representation of i; the next entry to visit is found by setting the rightmost one in the binary representation to a zero. In the previous example, the index 11 is represented as 1011 in binary. Setting the rightmost bit to zero gives 1010 and then 1000, which in decimal are 10 and 8, respectively.

To increment the count for symbol i, the appropriate entries beyond the one for i must be incremented. For example, if symbol 3 is to have its count incremented, then the array entries at $F[3]$, $F[4]$, and $F[8]$ must be incremented. This sequence is also easily determined from the binary representation of the numbers.

This implicit tree technique calculates cumulative sums in logarithmic time. Another mechanism exists that uses a tree in which the shape depends on the counts, and as a result the time taken is linear in number of input and output bits that would be used if the counts are used for an arithmetic coder (Moffat 1990b). Although asymptotically optimal, in practical applications this structure is no faster than the Fenwick tree described above, and it uses more memory.


```
"I never heerd a skilful old married feller of twenty
years' standing pipe ''my wife'' in a more used note
than 'a did," said Jacob Smallbury.  "It might have been
a little more true to nater if't had been spoke a little
chillie
```

Figure 2.23 Sample text.

2.5 Symbolwise models

Now we look at symbolwise models that can be combined with the coders described in the two previous sections. Compression methods that work in a symbolwise manner and make use of adaptively generated statistics give excellent compression—in fact, they include the best-known methods—and so are well worth studying. Four main approaches will be discussed in detail: PPM, which makes predictions based on previous characters; block sorting, which transforms the text to bring similar contexts together; DMC, which uses a finite-state machine for the model; and word-based methods, which use words rather than letters as the atomic symbols for compression. Each of these methods generates predictions for the input symbols. The predictions take the form of probability distributions that are provided to a coder—usually either an arithmetic or Huffman coder.

Prediction by partial matching

Prediction by partial matching (PPM) uses finite-context models of characters (Cleary and Witten 1984b). Rather than being restricted to one context length, it uses different sizes, depending on what contexts have been observed in the previously coded text—hence the term "partial matching" in the name.

Suppose Hardy's book is being coded by PPM and the encoder is up to the passage shown in Figure 2.23. The characters *chillie* have just been encoded, and the character *r* is about to be coded. PPM starts with a reasonably large context,

typically three or four characters. For illustration, suppose a context of five characters (*illie*) is used to try to make a prediction. This string has never occurred before in the prior text, so the encoder switches to a context of four characters (*llie*). The decoder is able to do likewise, since it has seen the same prior text as the encoder and also recognizes that the context has not occurred before. This smaller four-character context *has* occurred previously. However, it has only appeared once, and it was followed by the character *s*. This is a zero-frequency situation—the character *r* is to be coded, but it has a count of zero. Rather than code it explicitly using one of the zero-frequency methods described above, PPM sends a special "escape" symbol—available in every context—that tells the decoder that the symbol cannot be coded in the current context and that the next smaller context should be tried. Assigning a probability to the escape symbol is really just another form of the zero-frequency problem. One effective method is to allocate a count of one to the escape symbol. This is sometimes referred to as *escape method A*, and the version of PPM that uses it is referred to as *PPMA*. In the example, the escape symbol has a probability of 1/2 (and the character *s* has the remaining 1/2). The escape symbol is transmitted, and both encoder and decoder shift down to a context of three symbols. One bit has been transmitted so far and one arithmetic coding step completed.

The three-symbol context *lie* has occurred 201 times in the prior text, and 19 of those were followed by an *r*. Allowing one count for the escape symbol, the *r* can now be coded with a probability of 19/202, which requires 3.4 bits. In total, the *r* requires two encoding steps and is represented in 4.4 bits.

If the *r* character had not occurred in the order-3 context, another escape would have been used to get to the order-2, and then, if necessary, order-1 or order-0 contexts. If a symbol has never occurred in the prior text, even the order-0 model cannot be used, and a further escape symbol is transmitted to revert to a simple model in which all characters are coded with equal probability. In this case, six arithmetic encoding steps would be required. This may seem extreme, but actually it is very rare—during the early parts of the text, while the model is still learning, it is unlikely that fifth-order contexts will be repeated, and once the model is up to speed it is unlikely that any of the low-order contexts will be required. As a result, compression using the PPM method is only a little slower than using a straightforward zero-order character-based model that uses exactly one arithmetic coding step per symbol.

The probability estimates used above can be improved a little using a technique called *exclusions*. In the example, although the three-symbol context *lie* had occurred 201 times, 22 of these occurrences were followed by the letter *s*, yet an *s* will not be coded in that context since it was available to be coded in the four-symbol context. Therefore *s* can be excluded from the count. This means that only 179 occurrences of the context will be used as a sample, and an *r* will have an estimated probability of 19/180 instead of 19/202. Performing exclusion takes a little extra time but gives a reasonable payback in terms of extra compression (and an improvement is guaranteed since all nonexcluded characters have their probabilities increased). There are several other variations that offer different trade-offs between speed and compression (Lelewer and Hirschberg 1991; Howard and Vitter 1993a; Willems, Shtarkov, and Tjalkens 1995, 1996; Åberg, Shtarkov, and Smeets 1997; Bunton 1997b).

PPM is very effective. Figure 2.24 shows the number of bits used to code an excerpt extracted from the end of the Hardy book, assuming that the prior part of the book has already been processed to establish the contexts and frequency counts. The number of bits for each character includes any escape characters necessary to get the decoder to the correct context; for the example in this figure, the coder attempts to code each character first in a third-order context. Notice that many characters are very predictable. For example, the letter *d* in the word *old* in the first line is coded in just 0.19 bits; a *d* is very common in this context. The space character after the *d* is coded in 0.86 bits; it is not quite so predictable because the letter *e* is also considered likely since the prior text contains words like *older* and *beholder*. The unpredictable characters stand out. For example, longer representations are generated for the last three characters of the word *heerd* in the first line, and the closing parenthesis in the penultimate line takes 11.06 bits because, although that context occurs very frequently in the text, this is only the second time that *now* is followed by a parenthesis. Despite these exceptions, the majority of the symbols are predicted with high probabilities and so are coded in two to three bits each. Because PPM generates so many high probabilities, it is best to code its output with an arithmetic coder.

The amount of compression achieved by PPM is affected by the method used to estimate escape probabilities. One of the better methods, referred to as *method C*, estimates the probability of an escape to be $r/(n + r)$, where n is the total number of symbols seen previously in the current context and r is the number of them that were *distinct* (Moffat 1990a). A character with a count of c_i in the context would have a probability of $c_i/(n + r)$. Using method C, the probability of an escape character increases as the relative frequency of novel characters increases, but also decreases as the total number of times the context has occurred increases. Using PPMC (PPM with escape method C) combined with arithmetic coding, Hardy's book can be coded in an average of 2.5 bits per character; that is, it is compressed to 31 percent of its original size.

A number of improvements to the PPM method can shave a little more off the size of compressed files. For example, the PPMD method (which is consistently slightly better than PPMC) only gives half the weight to novel events, so the probability of an escape character is $r/(2n)$, and the probability of a symbol that has been observed c_i times in the current context is $(2c_i - 1)/(2n)$ (Howard 1993). Another way of calculating escape probabilities is method X, proposed by Witten and Bell (1991). In method X, the number of *hapax legomena* (symbols of frequency one) is used as an estimate of the number of symbols of frequency zero. That is, if t_1 is the number of symbols i for which $c_i = 1$, then the escape probability is calculated as $t_1/(n + t_1)$, and the probability of symbols with $c_i \geq 1$ as $c_i/(n + t_1)$. To avoid the problems that arise when $t_1 = 0$, and no symbols have appeared only once, further approximations can be used: $(t_1 + 1)/(n + t_1 + 1)$ and $c_i/(n + t_1 + 1)$, respectively.

As well as refinements to the escape calculation strategy, researchers have explored using arbitrarily large contexts instead of a fixed size for the starting point

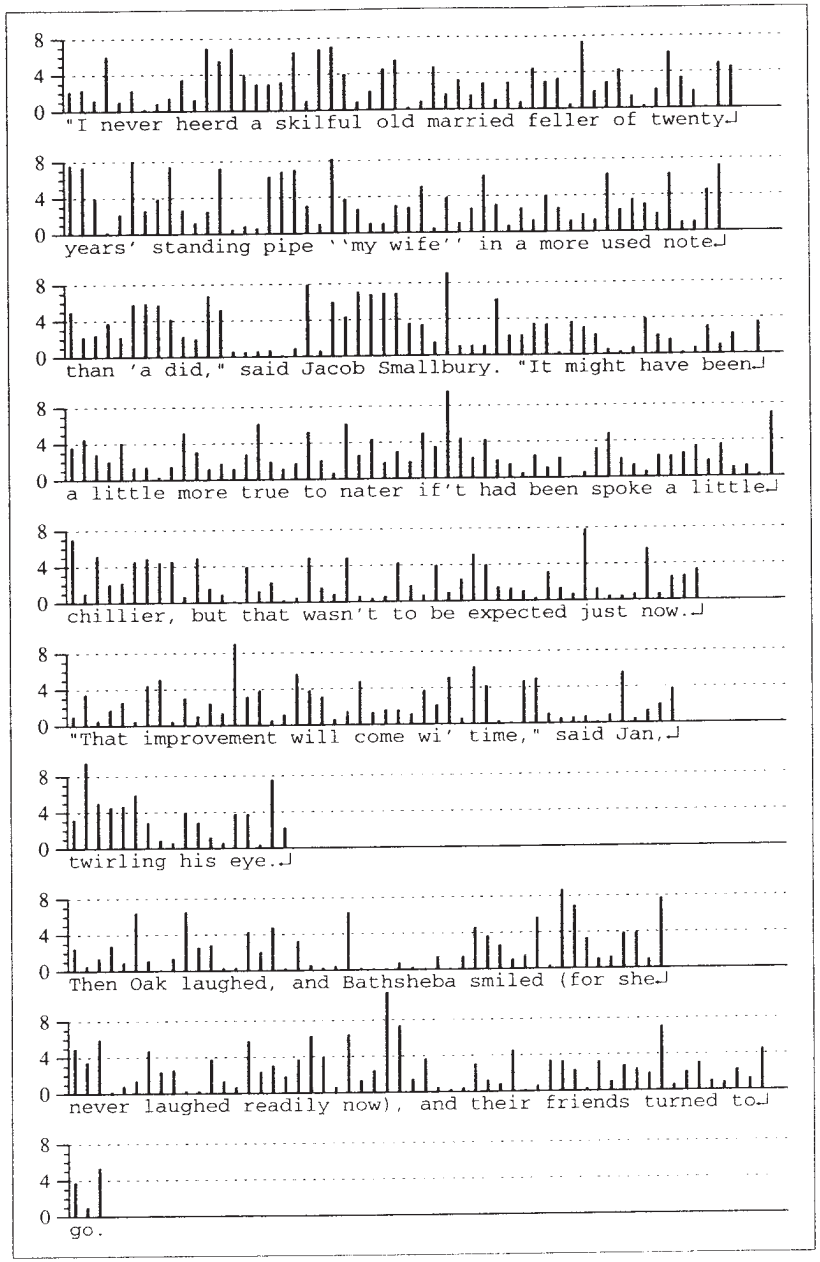

Figure 2.24 Information content of PPM coding of individual characters (bits per character).

(Cleary, Teahan, and Witten 1995). This general technique is referred to as PPM*, and one implementation, PPMZ, can compress *Far from the Madding Crowd* to about 2.2 bits per character. This places PPM-based methods among the best known for achieving good compression.

Some generalized PPM systems have been developed for experimenting with different forms of the basic algorithm. One is known as the Swiss Army Knife Data Compression (SAKDC) method (Williams 1991a). As its name suggests, SAKDC is actually many variations rolled into one, with some 25 parameters to control how the model is used. The parameters include the maximum context, how memory is managed, how probabilities are estimated, and how new contexts are added to the model. It includes several other techniques as special cases. Like its namesake, the large number of parameters makes it unsuitable for regular use, but it has been valuable in exploring the bounds of how much compression can be achieved with PPM-style models.

In her 1997 doctoral dissertation, Suzanne Bunton described a similar heavily parameterized program. Indeed, her "executable taxonomy" also includes the DMC model (described on page 69) as a variant, as well as many other possibilities.

Block-sorting compression

Block-sorting compression is an intriguing approach that was first published in 1994 (Burrows and Wheeler 1994). It is unusual because it works by transforming the text into a form that is more amenable to compression. The transformed text is compressed and transmitted to the decompressor, which reverses the compression and then applies the inverse transform. This is analogous to the discrete cosine transform, used for compressing images, or the Fourier transform, which converts signals from the time domain to the frequency domain. One disadvantage of block sorting is that the input must be broken up into blocks that are processed one at a time, rather than the process being continuously adaptive as characters arrive.

The transformation used for block-sorting compression is sometimes called the Burrows-Wheeler transform, after its inventors. It permutes the characters in a text so that those occurring in similar contexts end up near each other. The permuted text is the same size as the original, but it is easily compressed using simple techniques because only a limited selection of characters tends to occur in similar contexts. The decompressor must rearrange it back to its original order. It is remarkable that this reverse permutation can be done with little computational effort (in fact, it is remarkable that it can be done at all!).

The transformation is performed by sorting each character in the text, using its context as the sort key. Figure 2.25 shows some of the sorted characters for the Hardy book, preceded by their contexts. Notice that the contexts are compared for sorting by working from right to left. The comparisons go back as far as necessary to order any two contexts; if the beginning of the file is reached in a comparison, then it wraps around to the end of the file, although, as Figure 2.25 shows, most of the time only a few characters are needed to distinguish contexts. The transformed text is simply the characters in the order of their sorted contexts, so the permuted text

```
nly thrown into greater relie  f
n. Nevertheless, he␣was relie  v
eba, feeling a nameless relie  f
rise, experienced great relie  f
thsheba was momentarily relie  v
P 398>␣foreheads, quite␣relie  v
t such times is a great␣relie  f
e droning of␣blue-bottle flie  s
and the reasonable probabilie  s
tions, pinks, picotees, lilie  s
 her head and feet,␣the lilie  s
eads, all␣about their familie  s
e as common among the␣familie  s
d been spoke a little␣chillie  r
no absurd sides to the follie  s
lways be your␣friend,' replie  d
s I've got no chance,' replie  d
␣'O no -- not at all,' replie  d
 'tis my only doctor,' replie  d
```

Figure 2.25 Sorted contexts for the Burrows-Wheeler transform.

for the Hardy book will contain the sequence *fvffvvfsssssrsdddd* from the right-hand column of Figure 2.25.

The two main issues that arise at this point is how the permuted text should be coded and how the original text can be reconstructed from the permuted one. We will deal with the coding first.

Notice that characters occur in clusters in the permuted text. For example, in the context *relie* in Figure 2.25, only the letters f and v occur. This pattern is conveniently coded using a *move-to-front* coder, which maintains a list of characters, moving them to the front of the list each time they are coded. Characters near the front of the list are assigned shorter codes. In the example, f and v will be in the first two characters of the list for a while, and then will be displaced by s and, later, d.

But how can the original source text be reconstructed from a permutation like *fvffvvfsssssrsdddd*? The permuted text gives us the last column of characters from Figure 2.25. One important observation is that the second to last column (containing only the symbol e in Figure 2.25) can be constructed by sorting the permuted text, so the decoder has access to both of these columns.

The shorter text *mississippi* will be used as an example to describe the reverse transformation. Figure 2.26 shows the encoding of this text. In Figure 2.26a, each character is shown with its context, where wraparound is used to get a full con-

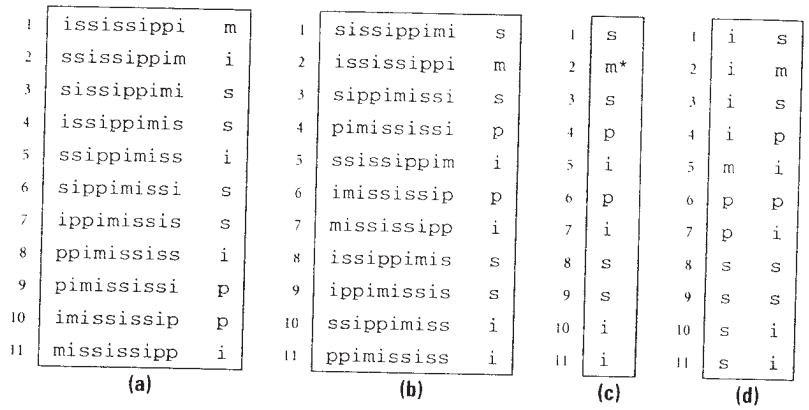

Figure 2.26 Burrows-Wheeler transform of the string *mississippi*: (a) rotations of the string; (b) sorted by contexts; (c) permuted string (transmitted), with * indicating the starting character; (d) sorted string (last character of context) and permuted string.

text. In Figure 2.26b, the contexts have been sorted into order, and the permuted string is transmitted (Figure 2.26c). The number of the context corresponding to the first character, which is 2 in this case, must also be transmitted; this is indicated by an asterisk in Figure 2.26c. The last character of the contexts is reconstructed in Figure 2.26d by sorting the permuted string.

Reversing the transformation relies on the key observation that the order in which the corresponding characters appear in the two columns are the same. For example, the letter s appears four times in each column. Looking at Figure 2.26b, in each column, the first appearance corresponds to the first s in *mississippi* (lines 1 and 8), the second appearance corresponds to the third s in the word (lines 3 and 9), the third appearance corresponds to the second (lines 8 and 10), and the fourth to the fourth (lines 9 and 11). This relationship can be used as follows to reconstruct the original string.

Using the information in Figure 2.26d, the decoder starts with the indicated first row (line 2), which gives the first character, m. Looking down the left-hand column, the context ending with m only occurs once (line 5), so this gives the next character, i. The context i has occurred four times. However, the one that was just decoded is the first i in the second column, so we should use the first i in the first column as the next context (line 1), giving the next character, s. This is the first s in the permuted string, so the next context is the first s (line 8), which is followed by another s. This latest s is the third one, so the next context is the third s (line 10), which is followed by i. The rest of the text is decoded by going through contexts 3, 9, 11, 4, 6, 7, and 2, at which point the whole text has been decoded.

To encode using the Burrows-Wheeler transform, and produce a permuted version of the input that is amenable to symbol-ranking coding,

1. Sort the N input characters using the preceding characters as the sort key, creating a permuted array $P[1 \ldots N]$.
2. Output the position in P that contains the first character from the compressed file.
3. Output the permuted array P.

To decode using the Burrows-Wheeler transform,

4. Set $p \leftarrow$ the position of the first input character (from the encoder).
5. Set $P[1 \ldots N] \leftarrow$ the permuted symbols (from the encoder).
6. Set $K[s] \leftarrow$ the number of times symbol s occurs in P.
7. Set the array $M[s]$ to be the position of the first occurrence of s in the array that would be obtained by sorting P:
   (a) Set $M[\text{first symbol in lexical order}] \leftarrow 1$.
   (b) For each symbol s (in lexical order)
   Set $M[s] \leftarrow M[s-1] + K[s-1]$.
8. For each input symbol position i from 1 to N
   (a) Set $s \leftarrow P[i]$.
   (b) Set $L[i] \leftarrow M[s]$.
   (c) Set $M[s] \leftarrow M[s] + 1$,

   Array L now stores the links with which to traverse the permuted string.
9. Traverse the link array to reconstruct the original string:
   (a) Set $i \leftarrow p$ (initial position).
   (b) Repeat N times (until $i = p$ a second time)
   Output $P[i]$,
   Set $i \leftarrow L[i]$.

Figure 2.27 Encoding and decoding using the Burrows-Wheeler transform.

In practice the block sorting can be implemented quite efficiently. For example, there is no need to create all the substrings as shown in Figure 2.26a since they can be represented by an array of pointers to the input text. The inverse transform can be performed efficiently by making a single pass through the strings in Figure 2.26d, storing links to the next context, which avoids searching through the contexts for each character.

Algorithms for encoding and decoding using the Burrows-Wheeler transform are shown in Figure 2.27. Encoding simply involves permuting the input symbols using each symbol's prior context as a sort key. The entire prior context is not usually needed for sorting; most of the time only a few previous characters will be required to determine the lexical order of the context. The sorted characters a.e coded using a coder that exploits the locality of reference in the permuted string.

The decoder recovers the permuted string into the array P. Although the example above showed a sorted version of P used to determine the permuted order (the first column of Figure 2.26d), it is not necessary for this sorted array to be constructed explicitly. Instead, two arrays (K and M) store it implicitly. Each different symbol in the sorted array will occur in one contiguous group. The array K stores how long each group is for each symbol, and from this M is contructed, which stores the position of the start of each group. These are easily constructed from P.

The decoding is facilitated by a link array, L, which stores the order in which characters should be taken from P. The link array is constructed from one pass through P using M to keep track of which occurrence of a symbol is being used. Finally, P is traversed using the order given by L, to produce the original uncompressed file.

The method used to code the permuted string is crucial to the compression performance of the system. The permuted string has quite different characteristics than normal text, and a specialized coding method is called for. A context-based coder is not appropriate because almost all of the contextual information has been removed by the permutation. As mentioned earlier, one suitable method is to use a move-to-front coder, which assigns a higher probability to characters that have occurred recently in the input. For text files, the position numbers in the move-to-front list usually follow an inverse square frequency distribution, which the coder must exploit. A variety of codes have been proposed for this, including zero-order arithmetic or Huffman coding or even fixed codes that approximate a typical distribution.

Figure 2.25 indicates that the sorted contexts correspond quite closely to the PPM method. For example, the context *chillie* appears adjacent to the context *follie*, which was the longest matching PPM context (order-4) and predicts an *s* after *llie*. This came up in coding the same character in the example of PPM coding (page 62). In fact, block sorting is very closely related to the PPM* method, which is a variant of PPM that allows arbitrary-length contexts. Not surprisingly, in practice the compression performance of block sorting is similar to that of PPM-based methods.

Dynamic Markov compression

Dynamic Markov compression (DMC) is a modeling technique based on a finite-state model (Cormack and Horspool 1987). Such a model is often called a Markov model after the Russian mathematician A. A. Markov (1856–1922). DMC is capable of achieving compression comparable to that of PPM, making it one of the better compression methods currently known. It is also quite easy to implement a working version of DMC, although it tends to be slow unless some effort is put into making it efficient.

DMC is adaptive. Both the probabilities and the structure of the finite-state machine change as coding proceeds. Figure 2.28 shows a model created by DMC. The

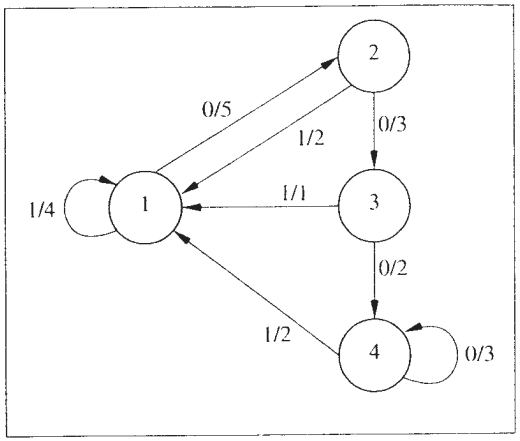

Figure 2.28 A model generated by the DMC method.

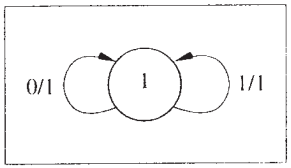

Figure 2.29 A simple initial model for DMC.

input alphabet comprises single bits, rather than bytes. Each transition out of a state records how often it has been traversed, and these counts are used to estimate the probabilities of the transitions. For example, the transition out of state 1 labeled 0/5 indicates that a 0 bit in state 1 has occurred five times. The decoder constructs an identical model using the symbols decoded and keeps track of which state the encoder is in. In practice, the values on the transitions are not necessarily integers because counts are divided up when states are cloned, as explained later.

The zero-frequency problem is avoided by initializing unused transitions to a count of one. Probabilities are estimated using the relative frequencies of the two transitions out of the current state. For example, if an encoder using the model in Figure 2.28 is in state 2 and the next input bit is a 0, then that input is coded with a probability of $3/(2 + 3) = 0.6$, corresponding to 0.737 bits. The count on the 0 transition is then incremented to 4, and the transition is followed so that state 3 is now the current state. The next input bit is coded from this new state.

The adaptation of the *structure* of a DMC model is achieved by a heuristic called *cloning*. In its simplest form, DMC starts with the elementary model shown in Figure 2.29. When a transition appears to be heavily used, the state that it leads to is

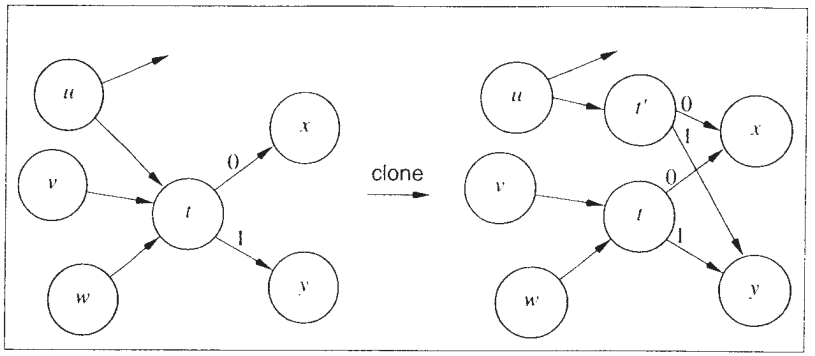

Figure 2.30 The DMC cloning operation.

cloned into two states. Figure 2.30 shows the cloning of state t. The heavy use of the transition into it from state u has triggered the cloning, and so a new state, t', is created. The transition from state u now goes to t', while transitions from other states (v and w) to t remain the same. The transitions out of state t' are set to be the same as those out of state t. The counts on the transitions out of t' are in the same ratio as the counts out of t, and the counts for the transitions out of t and t' are adjusted so that their sum is equal to the sum of counts on transitions coming in. This is a logical equivalent of Kirchoff's law for electrical circuits, which states that the algebraic sum of the currents flowing into any node must be zero. The extended structure means that a new state is now available to record independent probabilities for symbols occurring after a transition from state u to state t. Previously, these probabilities were confused with those of states v and w.

The encoding algorithm for DMC is shown in Figure 2.31. The structure of the finite-state machine is stored using the array T. Entry $T[t][e]$ in the array stores the state reached on transition e out of state t. The array $C[t][e]$ stores the number of times the input has traversed transition e out of state t, with one added to avoid the zero-frequency problem. Initially there is just one state, number 1, with both transitions going to state 1 (although more sophisticated initial models could be used). Linking in a new cloned state requires just three assignments to the entries in T, plus some work to redistribute the counts. The pseudocode shown here contains most of the details for implementing a DMC model—just a little over a dozen lines of code are required, with a simple data structure. The decoder is similar.

Being a finite-*state* model rather than a finite-*context* model, DMC is potentially more powerful than, for example, PPM. However, it has been shown that the choice of initial models and the nature of the cloning heuristic mean that only finite *contexts* are generated (Bell and Moffat 1989). This is borne out in the compression performance of DMC, which is similar to that of PPM.

The use of bits rather than bytes simplifies the implementation of DMC, although it has the disadvantage that every bit in the input must be coded individ-

To encode using the DMC method,

1. Set $s \leftarrow 1$. /* the current number of states */
2. Set $t \leftarrow 1$. /* the current state */
3. Set $T[1][0] \leftarrow 1$ and $T[1][1] \leftarrow 1$. /* initial model */
4. Set $C[1][0] \leftarrow 1$ and $C[1][1] \leftarrow 1$. /* set counts to 1 to avoid zero-frequency problem */
5. For each input bit e do
   (a) Set $u \leftarrow t$.
   (b) Set $t \leftarrow T[u][e]$. /* follow the transition */
   (c) Code e with probability $C[u][e]/(C[u][0] + C[u][1])$.
   (d) Set $C[u][e] \leftarrow C[u][e] + 1$.
   (e) If cloning thresholds are exceeded then
       Set $s \leftarrow s + 1$ /* the new state (t') */,
       Set $T[u][e] \leftarrow s$,
       Set $T[s][0] \leftarrow T[t][0]$,
       Set $T[s][1] \leftarrow T[t][1]$, and
       Move some of the counts from $C[t]$ to $C[s]$.

Figure 2.31 Encoding using DMC.

ually, and this can be slow. Bitwise DMC is probably one of the easiest models to implement—only a few lines of code are needed to perform the counting and cloning—yet DMC is one of the best compression methods in terms of compression performance. Its speed can be improved by adapting it to work with bytes rather than bits, but this requires more sophisticated data structures to avoid excessive memory usage. In this case, the implementation effort becomes comparable to that of other high-performance methods, such as PPM, and the difference between the two methods—both in implementation effort and compression efficiency—becomes small.

Word-based compression

Word-based compression methods parse a document into "words" (typically, contiguous alphanumeric characters) and "nonwords" (typically, punctuation and white-space characters) between the words. The words and nonwords become the symbols to be compressed. There are various ways to compress them. Generally, the most effective approach is to form a zero-order model for words and another for nonwords. It is assumed that the text consists of strictly alternating words and nonwords (the parsing method needs to ensure this), and so the two models are used alternately. If the models are adaptive, a means of transmitting previously unseen

words and nonwords is required. Usually, some escape symbol is transmitted, and then the novel word is spelled out character by character. The explicit characters can be compressed using a simple model, typically a zero-order model of the characters.

Although this approach seems to be specific to textual documents, it does not perform too badly on other types of data. This is because if few "words" are found, then the model effectively reverts to the simple zero-order model of characters, and for nontextual data such as images, this sort of model is reasonably appropriate. A well-tuned word-based compressor can achieve compression performance close to that of PPM, and it has the potential to be substantially faster because it codes several characters at a time.

There are many different ways to break English text into words and the intervening nonwords.[3] One scheme is to treat any string of contiguous alphabetic characters as a word and anything else as a nonword. More sophisticated schemes could take into account punctuation that is part of a word, such as apostrophes and hyphens, and even accommodate some likely sequences, such as a capital letter following a period. This kind of improvement does not have much effect on compression but may make the resulting list of words more useful for indexing purposes in a full-text retrieval system.

One aspect of parsing that deserves attention is the processing of numbers. If digits are treated in the same way as letters, a sequence of digits will be parsed as a word. This can cause problems if a document contains many numbers—such as tables of financial figures. The same situation occurs, and can easily be overlooked, when a large document contains page numbers—with 100,000 pages, the page numbers will generate 100,000 "words," each of which occurs only once. Such a host of unique words can have a serious impact on operation: in an adaptive system, each one must be spelled out explicitly, and in a static system, each will be stored in the compression model. In both cases this is grossly inefficient because the frequency distribution of these numbers is quite different from the frequency distribution of normal words for which the system is designed. One solution is to limit the length of numbers to just a few digits. Longer numbers are broken up into shorter ones, with a null punctuation marker in between. This moderates the vocabulary generated by the numbers and can result in considerable savings in decode-time memory requirements.

Word-based schemes can generate a large number of symbols since there is a different symbol for each word and word variant that appears in the text being coded. This means that special attention must be given to efficient data structures for storing the model. In a static or semi-static situation, a canonical Huffman code is

3 Note, however, that not all languages are so cooperative. Most Asian languages are written and stored without any equivalent of the white-space characters that are taken for granted in English, and they are very difficult to segment into words. For compression purposes, this is not a serious handicap, as other methods can be used. But for information retrieval purposes (see Chapter 4), where discrete words are employed as the terms used to search for documents relevant to a query, segmentation is an important problem indeed.

ideal. It provides efficient decoding, and because even the most common words occur with a relatively low frequency, coding inefficiency is small. Details of exactly such a scheme are given in Section 9.1.

This section has described a relatively small selection of symbolwise models for compression. Many other models have been proposed. They are generally based on the principles discussed above and differ chiefly in how they compromise between speed, memory requirements, and compression performance. The "Further reading" section at the end of this chapter cites books and surveys that discuss some of these methods in more detail.

2.6 Dictionary models

Dictionary-based compression methods use the principle of replacing substrings in a text with a codeword that identifies that substring in a *dictionary*, or *codebook*. The dictionary contains a list of substrings and a codeword for each substring. This type of substitution is used naturally in everyday life, for example, in the substitution of the number 12 for the word *December*, or representing "the chord of B minor with the seventh added" as *Bm7*. Unlike symbolwise methods, dictionary methods often use fixed codewords rather than explicit probability distributions because reasonable compression can be obtained even if little attention is paid to the coding component.

The simplest dictionary compression methods use small codebooks. For example, in *digram coding*, selected pairs of letters are replaced with codewords. A codebook for the ASCII character set might contain the 128 ASCII characters, as well as 128 common letter pairs. The output codewords are eight bits each, and the presence of the full ASCII character set in the codebook ensures that any input can be represented. At best, every pair of characters is replaced with a codeword, reducing the input from seven bits per character to four bits per character. At worst, each seven-bit character will be expanded to eight bits. Furthermore, a straightforward extension caters to files that might contain some non-ASCII bytes—one codeword is reserved as an escape, to indicate that the next byte should be interpreted as a single eight-bit character rather than as a codeword for a pair of ASCII characters. Of course, a file consisting of mainly binary data will be expanded significantly by this approach; this is the inevitable price that must be paid for use of a static model.

Another natural extension of this system is to put even larger entries in the codebook—perhaps common words, like *and* and *the*, or common components of words, such as *pre* and *tion*. Strings like these that appear in the dictionary are sometimes called *phrases*. A phrase may sometimes be as short as one or two characters, or it may include several words. Unfortunately, having a dictionary with a predetermined set of phrases does not give very good compression because the entries must usually be quite short if input independence is to be achieved. In fact, the more suitable the dictionary is for one sort of text, the less suitable it is for others. For example, if this book were to be compressed, then we would do well if the codebook contained phrases like *compress*, *dictionary*, and even *arithmetic coding*, but such a codebook would be unsuitable for a text on, say, business management.

One way to avoid the problem of the dictionary being unsuitable for the text at hand is to use a semi-static dictionary scheme, constructing a new codebook for each text that is to be compressed. However, the overhead of transmitting or storing the dictionary is significant, and deciding which phrases should be put in the codebook to maximize compression is a surprisingly difficult problem.

The elegant solution to this problem is to use an adaptive dictionary scheme. Practically all adaptive dictionary compression methods are based on one of just two related methods developed by Jacob Ziv and Abraham Lempel in the 1970s (Ziv and Lempel 1977, 1978). We label the methods LZ77 and LZ78, respectively, after the years in which they were published; some authors refer to the two methods as LZ1 and LZ2, respectively. They are only rarely referred to as being "ZL" methods, even though this is the ordering of the authorship of the seminal papers in which they are described. These methods are the basis for many schemes that are widely used in utilities for compressing and archiving, although they have undergone much fine-tuning since their invention.

Both methods use a simple principle to achieve adaptivity: a substring of text is replaced with a pointer to where it has occurred previously. Thus, the codebook is essentially all the text prior to the current position, and the codewords are represented by pointers. The prior text makes a very good dictionary since it is usually in the same style and language as upcoming text; furthermore, the dictionary is transmitted implicitly at no cost because the decoder has access to all previously encoded text. The many variants of Ziv-Lempel coding differ primarily in how pointers are represented and in the limitations they impose on what the pointers are able to refer to.

The LZ77 family of adaptive dictionary coders

One of the key features of LZ77 (Ziv and Lempel 1977) and its successors is that it is relatively easy to implement, and decoding can be performed extremely quickly using only a small amount of memory. For these reasons it is particularly suitable when the resources required for decoding must be minimized, such as when data is distributed or broadcast from a central source to a number of small computers.

Like many compression methods, LZ77 is most easily explained in terms of its decoding. Figure 2.32 shows some output from an LZ77 encoder, supposing for the purposes of the example that the input alphabet consists of just *a*s and *b*s. The output consists of a series of triples. The first component of a triple indicates how far back to look in the previous (decoded) text to find the next phrase, the second component records how long the phrase is, and the third gives the next character from the input. The first two items constitute a pointer back into the text. Strictly, the third is necessary only when the character to be coded does not occur anywhere in the previous input; it is included in every triple for the sake of simplicity.

In Figure 2.32, the characters *abaabab* have already been decoded, and the next characters to be decoded are represented by the triple $\langle 5, 3, b \rangle$. Thus the decoder

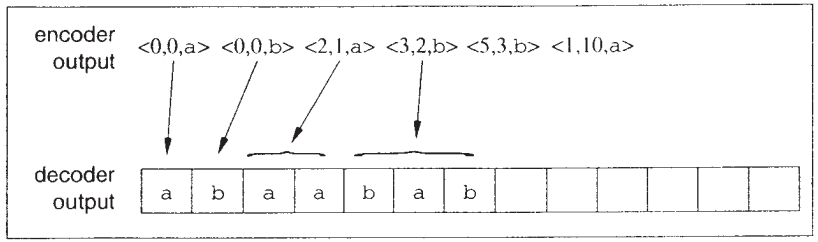

Figure 2.32 Example of LZ77 compression.

goes back five characters in the decoded text (to the third one from the start) and copies three characters, yielding the phrase *aab*. The third item in the triple, b, is then added to the output. The next triple, $\langle 1, 10, a\rangle$, is a recursive reference. To decode it, the decoder starts copying from one character back (the b) and copies the next 10 characters. Despite the recursive reference, each of the characters will be available before it is needed, yielding 10 consecutive bs. In this way, a kind of run-length coding is achieved.

LZ77 places limitations on how far back a pointer can refer and the maximum size of the string referred to. For English text there is little advantage in allowing the reach of pointers to exceed a window of a few thousand characters. For example, if the window is limited to 8,192 characters, then the amount of text it holds is equivalent to several book pages, and the first component of the triple can be represented in 13 bits. Extending the reach of the pointer beyond this makes more text available for referencing, but the gain is generally offset by the extra cost of storing the pointer. The second component of the triple, the length of the phrase, is also limited, typically to about 16 characters. Again, matches longer than this are rare and do not justify allocating extra space to the number representing the length of the phrase.

Algorithms for LZ77 encoding and decoding are shown in Figure 2.33. The search for a match may return a length of zero, in which case the position of the match is not relevant. Notice that the decoder is simply a small loop that copies from an array. In practice the array can be a circular buffer of W characters, and characters are written to the output as they are decoded.

The LZ77 method has gradually been refined into systems that have fast implementations and give good compression. There are some straightforward ways to improve the method described above. It is common to use different representations for the pointers. For the first component (the offset), it can be effective to use shorter codewords for recent matches and longer codewords for matches further back in the window, because recent matches are more common than distant ones. The second component of a pointer (the match length) can be represented more efficiently with variable-length codes that use fewer bits to represent smaller numbers. Also, in many schemes the third element of the triple, the character, is included only when necessary. For example, a one-bit flag can be used to indicate whether the next

To encode the text $S[1 \ldots N]$ using the LZ77 method, with a sliding window of W characters,

1. Set $p \leftarrow 1$. /* the next character of S to be coded */
2. While there is text remaining to be coded do
   (a) Search for the longest match for $S[p \ldots]$ in $S[p - W \ldots p - 1]$. Suppose that the match occurs at position m, with length l.
   (b) Output the triple $\langle p - m, l, S[p + l]\rangle$.
   (c) Set $p \leftarrow p + l + 1$.

To decode the text $S[1 \ldots N]$ using the LZ77 method, with a sliding window of W characters,

1. Set $p \leftarrow 1$. /* the next character of S to be decoded */
2. For each triple $\langle f, l, c\rangle$ in the input do
   (a) Set $S[p \ldots p + l - 1] \leftarrow S[p - f \ldots p - f + l - 1]$.
   (b) Set $S[p + l] \leftarrow c$.
   (c) Set $p \leftarrow p + l + 1$.

Figure 2.33 Encoding and decoding using LZ77.

```
"I never heerd a skilful old married
feller of twenty␣years' standing
pipe ''my wife'' in a more used not
e␣than 'a did," said Jacob Smallbury. "
It might have been␣a little more true
to nater if't had been spoke a little␣
chillier, but that wasn't to be
 expected just now.␣"That improvement
will come wi' time," said Jan,␣
twirling his eye.␣Then Oak laughed
, and Bathsheba smiled (for she␣never
laughed readily now), and their friends
 turned to␣go.␣
```

Figure 2.34 Coding of Hardy's book using an LZ77-type method.

item in the output is a pointer (offset and match length) or a character. Figure 2.34 shows how part of the Hardy book is coded using an LZ77-based method. Boxes are drawn around characters that can be coded as pointers. The rest of the characters could not be coded economically as part of a pointer and were transmitted as "raw"

characters. It is interesting to compare this coding with the one shown in Figure 2.24 on page 64, which shows the PPM algorithm: both methods tend to have difficulty with the same parts of the text.

Encoding for LZ77 involves searching the window of prior text for the longest match with the upcoming phrase. A naive linear search is very time-consuming and can be accelerated by indexing the prior text with a suitable data structure, such as a trie,[4] hash table, or binary search tree. A simple but effective method of searching is to use an index of pairs of characters in the window. Each entry in the index is the head of a linked list that points to each occurrence of the given pair of characters in the window. This can greatly decrease the number of matches that need to be evaluated. The index could be a two-dimensional array, a hash table, or a structure that stores just the more popular character pairs. Speed can be guaranteed at the expense of slight compression loss by limiting how much of the linked list is searched. The characters in the window can be stored in a circular buffer, so very little data movement is required to maintain the window.

Decoding for LZ77-type methods is very fast because each character decoded requires just one array lookup; moreover, the array is small compared to the cache size of current computers, so the lookup is likely to be very fast. The decoding program is very simple, so it can be included with the data at very little cost—in fact, the compressed data is stored as part of the decoder program, so the user sees just one file. This makes the data self-expanding, in that the original compression software is not needed to read it. For example, a compressed file could be downloaded from a network and expanded without needing any extra software. It is common to distribute files using this technique. When executed, this "program" generates the original file or files, greatly simplifying the distribution and installation of software and data.

The *gzip* variant of LZ77

One of the higher-performance compression methods based on LZ77 is *gzip*, distributed by the Gnu Free Software Foundation in Cambridge, Massachusetts (Gailly 1993). It contains many worthwhile refinements and so is described in a little more detail here.

Gzip uses a hash table to locate previous occurrences of strings. The next three characters to be coded are hashed, and the resulting value is used to look up a table entry. This is the head of a linked list that contains places where the three characters have occurred in the window. At the expense of a small loss in compression, the length of the linked list is restricted to prevent search time from growing too much. The size of the limit on the list length can be chosen at encoding time, so that the speed/compression trade-off is made by the user. Having a limit is particularly important if the input contains long runs of the same character because this results in a very long list of references. Long lists are time-consuming to maintain and are unnecessary because the first few items in the list will usually be more than sufficient to find a good match. Compression can be improved slightly by storing recent occurrences at the beginning of the linked list, in order to favor recent matches. Matches are represented in the encoded file by a pointer consisting of an offset and a length. If no suitable previous occurrence can be found, a raw character is transmitted. The offset of a pointer is represented using a Huffman code so that more frequent offsets (usually recent ones) are coded in fewer bits. The match length is represented by another Huffman code, and the same code is also used for raw characters. It may seem contradictory to combine the match lengths and raw characters into one code, but in fact this gives better compression because if they were separated, an extra bit would need to be transmitted to indicate whether the next input is a match length or a character; the combined code, however, can use less than one bit on average. The match length is sent before the offset of a pointer so that the decoder can tell whether a pointer or a raw character is being transmitted.

As described so far, the matching algorithm for *gzip* is "greedy"—it codes the upcoming characters as a pointer if at all possible. Sometimes, long-term compression is actually better if a raw character is transmitted, even though a pointer could be used. This occurs when the use of a raw character gives a better match for the characters immediately following the one about to be coded. If the user specifies that compression is more important than speed, *gzip* checks for this situation and transmits a raw character if that yields better compression in the long run.

The Huffman codes for *gzip* are generated semi-statically. Blocks of up to 64 Kbytes from the input file are processed at a time. The appropriate canonical Huffman codes are generated for the pointers and raw characters, and a code table is placed at the beginning of the compressed form of the block. This means that *gzip* is not really a single-pass method. However, the blocks are small enough to be held in memory. As a result, the input file need be read only once, and the program behaves as if it operates in one pass.

Because of the fast searching algorithm and compact output representation based upon Huffman codes, *gzip* outperforms most other Ziv-Lempel methods in terms of both speed and compression effectiveness. One implementation that is faster is the LZRW1 method, which achieves extremely fast encoding and decoding by limiting the search of previous text to just one candidate phrase (Williams 1991b). As with *gzip*, a hash table is used to locate previous occurrences of triples of characters. However, the table has just one entry for each hashed value, so a collision results in the previous phrase at that hash location being lost. Further time is saved by updating the hash table only with triples that begin a coded phrase; that is, it is updated only once per phrase coded, rather than once per byte in the input. Simple byte-aligned binary codes are used for all output components. This ruthless housekeeping results in considerable speed, at the price of compression performance. Performance measurements for these methods are given in Section 2.8.

The LZ78 family of adaptive dictionary coders

In contrast to the LZ77 method, in which pointers can refer to any substring in the window of prior text, the LZ78 method (Ziv and Lempel 1978) places restrictions

4 A trie is a multiway tree with a path for each string inserted in it, which allows rapid location of strings and substrings, and is discussed in more detail on page 80.

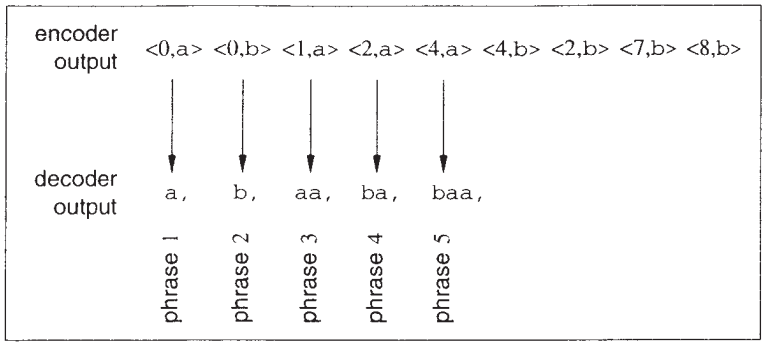

Figure 2.35 A string compressed by LZ78.

on which substrings can be referenced. However, it does not have a window to limit how far back substrings can be referenced. By restricting the set of strings that can be referenced, LZ78 avoids the inefficiency of having more than one coded representation for the same string, which occurs frequently in LZ77 methods since the window will often contain many repeated substrings.

Figure 2.35 shows a string being decoded by LZ78. The text prior to the current coding position has been parsed into substrings, and only these parsed phrases can be referenced. They are numbered in sequence, so phrase number 1 is *a*, number 2 is *b*, number 3 is *aa*, and so on. The characters about to be encoded are represented by the number of the longest parsed substring that the characters match, followed by an explicit character. The next part of the encoder output to be decoded is the pair $\langle 4, b \rangle$, which represents phrase 4 (*ba*) followed by a *b*. The characters just decoded (*bab*) are added to the dictionary as a new phrase, number 6. The remaining pairs represent a run of consecutive *b*s. The code $\langle 2, b \rangle$ represents *bb*, and in the remainder of the example the phrases used to code the run gradually increase in length, one character at a time.

Phrase 0 is the empty string, so if there is no match with a previous phrase then the next character, say, *c*, can always be coded as $\langle 0, c \rangle$.

The parsing strategy can be implemented efficiently by storing the phrases in a trie data structure (sometimes also known as a digital search tree). Figure 2.36 shows the trie for the parsed phrases of Figure 2.35. The characters of each phrase specify a path from the root of the trie to the node that contains the number of the phrase. The characters that are about to be encoded are used to traverse the trie until the path is blocked, either because there is no onward path for the indicated character or because a leaf is reached. The node at which the block occurs gives the phrase number to output. The next character from the input is then used to add a new node below this one, and this is how new phrases are added to the codebook. In the example, when the phrase *bab* was being encoded, the first two characters (*ba*) were used to traverse the trie to node 4, and then the extra branch from this node was

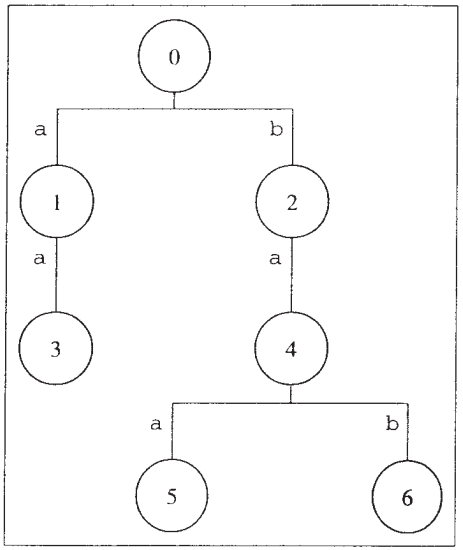

Figure 2.36 Trie data structure for LZ78 coding.

added to create node 6 (shown shaded in Figure 2.36). Thus, the longest previous phrase (phrase 4) is found, and the structure is updated with the new phrase at the same time. In practice, the multiway branches of a trie are tricky to implement efficiently because most nodes have relatively few children if the input alphabet is large, requiring an efficient representation for the sparse array of pointers. It can be faster and simpler to use a hash table in which the current node number and the next input character are hashed to determine where the next node can be found.

The data structure for an LZ78 compressor continues to grow throughout coding, and eventually growth must be stopped to avoid using too much memory. Several strategies can be used when memory is full. The trie can be removed and reinitialized; it can be used as it is, without further updates; or it can be partially rebuilt using the last few hundred bytes of coded text, thereby avoiding much of the learning penalty of starting again from scratch.

Although encoding for LZ78 can be faster than for LZ77, decoding is slower because the decoder must also store the parsed phrases. Nevertheless, the scheme is attractive, and one of its variants, LZW, forms the basis of several widely used compression systems.

The LZW variant of LZ78

LZW (Welch 1984) is one of the more popular variants of Ziv-Lempel coding, partly because the paper describing it is more accessible than Ziv and Lempel's 1978 one in

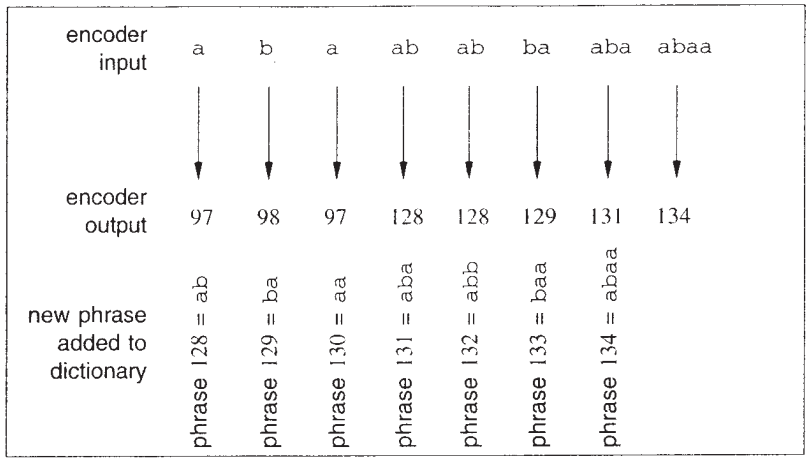

Figure 2.37 Example of LZW coding.

which the original idea was proposed. It has been used as the basis of several popular programs, including the Unix *compress* program and some personal computer archiving systems.

The main difference between LZW and LZ78 is that LZW encodes only the phrase numbers and does not have explicit characters in the output. This is made possible by initializing the list of phrases to include all characters in the input alphabet. A new phrase is constructed from a coded one by appending the first character of the *next* phrase to it. Figure 2.37 shows a string that has been partially coded. The phrases are numbered from 128 because 0 to 127 are used to represent the 128 characters of the ASCII alphabet. Each phrase is coded as a single number that identifies a previously parsed phrase. For example, the eighth and ninth characters, *ba*, are represented by transmitting the number 129 to identify a phrase that was constructed earlier. The new phrase to be added after the *ba* is received is constructed by adding the next character, *a* (not yet coded), to the phrase, creating the new phrase *baa*, which is number 133. Next, the phrase *aba* is coded as the number 131. It is only at this point that the decoder can construct phrase 133 because it needs to use the first character of phrase 134 (*abaa*).

This lag in construction of phrases is not a problem unless a new phrase is used by the encoder immediately after it is constructed. The last phrase in Figure 2.37 shows how this situation can arise for the decoder. The first seven phrases have been decoded, and the phrase *abaa* (number 134) has just been added to the encoder's dictionary. The decoder does not yet know what the last character of the phrase is, so it cannot be added to the decoder's dictionary. The next input codeword, 134, now requires the use of the new phrase. Fortunately, the decoder knows the beginning of the new phrase—it is *aba*—and the last (currently unknown) character

To encode the text $S[1 \ldots N]$ using the LZW method,

1. Set $p \leftarrow 1$. /* the next character of S to be coded */
2. For each character $d \in 0 \ldots q-1$ in the alphabet do /* initial dictionary */
   Set $D[d] \leftarrow$ character d.
3. Set $d \leftarrow q-1$. /* d points to the last entry in the dictionary */
4. While there is text remaining to be coded do
   (a) Search for the longest match for $S[p \ldots]$ in D.
   Suppose that the match occurs at entry c, with length l.
   (b) Output the code c.
   (c) Set $d \leftarrow d+1$. /* add an entry to the dictionary */
   (d) Set $p \leftarrow p+l$.
   (e) Set $D[d] \leftarrow D[c]$ ++ $S[p]$. /* concatenation */

To decode using the LZW method,

1. Set $p \leftarrow 1$. /* the next character of S to be decoded */
2. For each character $d \in 0 \ldots q-1$ in the alphabet do
   Set $D[d] \leftarrow$ character d.
3. Set $d \leftarrow q-1$. /* d points to the last entry in the dictionary */
4. For each code c in the input do
   (a) If $d \neq (q-1)$ then /* first time is an exception */
   Set last character of $D[d] \leftarrow$ first character of $D[c]$.
   (b) Output $D[c]$.
   (c) Set $d \leftarrow d+1$. /* add an entry to the dictionary */
   (d) Set $D[d] \leftarrow D[c]$ ++ ?. /* last character is currently unknown */

Figure 2.38 Encoding and decoding using LZW.

of the phrase is the first character of the next phrase (number 135). Since phrase 135 will be constructed by appending one character to phrase 134, phrase 135 must begin with the same character as phrase 134, an *a*. Thus, phrase 134 must be *abaa*, and decoding can proceed. In fact, whenever a phrase is referenced as soon as the encoder has constructed it, the last character of the phrase must be the same as its first, so the decoder can construct the phrase too. Despite the slight inelegance of having to deal with this exception, LZW gives good compression and is relatively easy to implement efficiently.

The LZW algorithms for encoding and decoding are shown in Figure 2.38. The dictionary D is initialized to contain the q symbols in the alphabet (typically $q = 256$ for bytewise coding). The text is coded by looking for the longest match in

```
"I never heerd a skilful old married
 feller of twenty␣years' standing
 pipe ''my wife'' in a more used note
␣than 'a did," said Jacob Smallbury. "I
t might have been␣a little more true
to nater if't had been spoke a little␣c
hillier, but that wasn't to be
expected just now.␣"That improvement w
ill come wi' time," said Jan,␣twirli
ng his eye.␣Then Oak laughed, and
 Bathsheba smiled (for she␣never laughe
d readily now), and their friends
 turned to␣go.␣
```

Figure 2.39 Phrases parsed by Unix *compress* coding the Hardy book.

the dictionary, and a new entry is created at each step by concatenating the next input symbol with the code just used (concatenation of strings is represented by "++"). The decoder begins with the same initial dictionary and adds an entry each time an input code is processed. However, the last character of the last entry in the dictionary (entry *d*) is unknown until the next code is read from the input, and it is represented by a question mark in Figure 2.38.

There are several variants to LZW, which are the result of fine-tuning. The Unix utility *compress* is one of the more widely used variants. Here, the number of bits used to indicate phrases is gradually increased instead of using more bits than necessary when there are few phrases in the dictionary. *Compress* also places a maximum on the number of phrases that are constructed, which in turn limits the amount of memory required for coding. When the dictionary is full, adaptation ceases. Compression performance is monitored, and if it deteriorates significantly, the dictionary is cleared and rebuilt from scratch. Figure 2.39 shows how the Hardy book is parsed into phrases by *compress*. A box is drawn around each phrase that is transmitted. This segment is from the end of the book, and each phrase is represented in 15 bits. The dictionary contains many common phrases at this stage, such as *never* and *ave been.* Only rarely is a single character coded as a 15-bit reference, and whole words are often represented by a single pointer.

Many other variations on Ziv-Lempel coding have been proposed. They can be classified according to how they parse strings from the input, how they represent pointers in the output, and how they prevent the dictionary from using too much memory. Hybrids of LZ77 and LZ78 have also been described; these build up a limited list of previous strings that can be referenced but also allow the length of the string to be specified so that partial matches can be used.

2.7 Synchronization

Good compression methods, including most of the schemes described above, perform best when compressing large files. This tends to preclude random access because the decompression algorithms are sequential by nature. Full-text retrieval systems require random access, and special measures need to be taken to facilitate this.

There are two reasons that the better compression methods require files to be decoded from the beginning: they use variable-length codes, and their models are adaptive. With variable-length codes, it is not possible to begin decoding at an arbitrary position in the file since we cannot be sure of starting on the boundary between two codewords. The situation is even worse for arithmetic coding because there isn't a boundary between codewords. This section describes how this can be overcome by creating synchronization points.

Adaptive modeling exacerbates the situation because, even if a codeword boundary is known, the model required for decoding can be constructed only using all of the preceding text. To achieve synchronization with adaptive modeling, large files must be broken up into small sections of, say, a few kilobytes. It is not likely that good compression can be obtained by compressing each of the small sections independently. Instead, the compression model needs to be "primed" before the section is compressed. A good choice for the priming text is a sample of the text being compressed. The same sample can be used for all the sections, provided that their styles are similar. Alternatively, the sections could be grouped according to the kind of text that they store and a separate priming text used for each group. If speed is important, the system could store a primed model, rather than reconstruct it each time it is needed. However, the size of a model is generally much greater than the size of the text from which it is built.

For full-text retrieval it is usually preferable to use a static model rather than an adaptive one. Static models are more appropriate given the static nature of a large textual database, and the need for random access decoding is of paramount importance.

In this section we will look at two methods for achieving random access in compressed files. The first is *synchronization points*, where the coding is reset to a known state at certain points, and the second is *self-synchronizing codes*, which allow decoding to start at a random point because it will automatically come into synchronization after a while.

Creating synchronization points

In a full-text retrieval system, the main text usually consists of a number of *documents*, which represent the smallest unit to which random access is required. In an uncompressed text, a document can be identified simply by specifying how many bytes it is from the start of the file. However, when a Huffman code or other variable-length code is used, a document will not necessarily begin on a byte boundary. One solution is to give the bit offset of the document from the start of the file, requiring an additional three bits in each entry of the index.