5 Dictionary Techniques

5.1 Overview

In the previous two chapters we looked at coding techniques that assume a source that generates a sequence of independent symbols. As most sources are correlated to start with, the coding step is generally preceded by a decorrelation step. In this chapter we will look at techniques that incorporate the structure in the data in order to increase the amount of compression. These techniques—both static and adaptive (or dynamic)—build a list of commonly occurring patterns and encode these patterns by transmitting their index in the list. They are most useful with sources that generate a relatively small number of patterns quite frequently, such as text sources and computer commands. We discuss applications to text compression, modem communications, and image compression.

5.2 Introduction

In many applications, the output of the source consists of recurring patterns. A classic example is a text source in which certain patterns or words recur constantly. Also, there are certain patterns that simply do not occur, or if they do, occur with great rarity. For example, we can be reasonably sure that the word *Limpopo*[1] occurs in a very small fraction of the text sources in existence.

A very reasonable approach to encoding such sources is to keep a list, or *dictionary*, of frequently occurring patterns. When these patterns appear in the source output, they are encoded with a reference to the dictionary. If the pattern does not appear in the dictionary, then it can be encoded using some other, less efficient, method. In effect we are splitting

[1] "How the Elephant Got Its Trunk" in *Just So Stories* by Rudyard Kipling.

the input into two classes, frequently occurring patterns and infrequently occurring patterns. For this technique to be effective, the class of frequently occurring patterns, and hence the size of the dictionary, must be much smaller than the number of all possible patterns.

Suppose we have a particular text that consists of four-character words, three characters from the 26 lowercase letters of the English alphabet followed by a punctuation mark. Suppose our source alphabet consists of the 26 lowercase letters of the English alphabet and the punctuation marks comma, period, exclamation mark, question mark, semicolon, and colon. In other words, the size of the input alphabet is 32. If we were to encode the text source one character at a time, treating each character as an equally likely event, we would need 5 bits per character. Treating all 32^4 ($= 2^{20} = 1{,}048{,}576$) four-character patterns as equally likely, we have a code that assigns 20 bits to each four-character pattern. Let us now put the 256 most likely four-character patterns into a dictionary. The transmission scheme works as follows: Whenever we want to send a pattern that exists in the dictionary, we will send a 1-bit flag, say, a 0, followed by an 8-bit index corresponding to the entry in the dictionary. If the pattern is not in the dictionary, we will send a 1 followed by the 20-bit encoding of the pattern. If the pattern we encounter is not in the dictionary, we will actually use more bits than in the original scheme, 21 instead of 20. But if it is in the dictionary, we will send only 9 bits. The utility of our scheme will depend on the percentage of the words we encounter that are in the dictionary. We can get an idea about the utility of our scheme by calculating the average number of bits per pattern. If the probability of encountering a pattern from the dictionary is p, then the average number of bits per pattern R is given by

$$R = 9p + 21(1-p) = 21 - 12p. \tag{5.1}$$

For our scheme to be useful, R should have a value less than 20. This happens when $p \geq 0.084$. This does not seem like a very large number. However, note that if all patterns were occurring in an equally likely manner, the probability of encountering a pattern from the dictionary would be less than 0.00025!

We do not simply want a coding scheme that performs slightly better than the simple-minded approach of coding each pattern as equally likely; we would like to improve the performance as much as possible. In order for this to happen, p should be as large as possible. This means that we should carefully select patterns that are most likely to occur as entries in the dictionary. To do this, we have to have a pretty good idea about the structure of the source output. If we do not have information of this sort available to us prior to the encoding of a particular source output, we need to acquire this information somehow when we are encoding. If we feel we have sufficient prior knowledge, we can use a *static* approach; if not, we can take an *adaptive* approach. We will look at both these approaches in this chapter.

5.3 Static Dictionary

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

Other words such as "Sophomore," "credits," and so on will occur quite often. Depending on the location of the university, certain digits in social security numbers are more likely to occur. For example, in Nebraska most student ID numbers begin with the digits 505. In fact, most entries will be of a recurring nature. In this situation, it is highly efficient to design a compression scheme based on a static dictionary containing the recurring patterns. Similarly, there could be a number of other situations in which an application-specific or data-specific static-dictionary-based coding scheme would be the most efficient. It should be noted that these schemes would work well only for the applications and data they were designed for. If these schemes were to be used with different applications, they may cause an expansion of the data instead of compression.

A static dictionary technique that is less specific to a single application is *digram coding*. We describe this in the next section.

5.3.1 Digram Coding

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

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

Example 5.3.1:

Suppose we have a source with a five-letter alphabet $\mathcal{A} = \{a, b, c, d, r\}$. Based on knowledge about the source, we build the dictionary shown in Table 5.1.

TABLE 5.1 A sample dictionary.

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

Suppose we wish to encode the sequence

abracadabra

The encoder reads the first two characters *ab* and checks to see if this pair of letters exists in the dictionary. It does and is encoded using the codeword 101. The encoder then reads the next two characters *ra* and checks to see if this pair occurs in the dictionary. It does not, so the encoder sends out the code for *r*, which is 100, then reads in one more character, *c*, to make the two-character pattern *ac*. This does exist in the dictionary and is encoded as 110. Continuing in this fashion, the remainder of the sequence is coded. The output string for the given input sequence is 1011001101111101100000. ♦

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

Pair	Count	Pair	Count
eƀ	1128	*ar*	314
ƀt	838	*at*	313
ƀƀ	823	*ƀw*	309
th	817	*te*	296
he	712	*ƀs*	295
in	512	*dƀ*	272
sƀ	494	*ƀo*	266
er	433	*io*	257
ƀa	425	*co*	256
tƀ	401	*re*	247
en	392	*ƀ$*	246
on	385	*rƀ*	239
nƀ	353	*di*	230
ti	322	*ic*	229
ƀi	317	*ct*	226

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

Pair	Count	Pair	Count
ƀƀ	5728	*st*	442
nlƀ	1471	*le*	440
; *nl*	1133	*ut*	440
in	985	*f*(	416
nt	739	*ar*	381
=*ƀ*	687	*or*	374
ƀi	662	*rƀ*	373
tƀ	615	*en*	371
ƀ=	612	*er*	358
);	558	*ri*	357
,*ƀ*	554	*at*	352
nlnl	506	*pr*	351
ƀf	505	*te*	349
eƀ	500	*an*	348
*ƀ**	444	*lo*	347

A list of the 30 most frequently occurring pairs of characters in an earlier version of this chapter is shown in Table 5.2. For comparison, the 30 most frequently occurring pairs of characters in a set of C programs is shown in Table 5.3.

In these tables, ƀ corresponds to a space and *nl* corresponds to a new line. Notice how different the two tables are. It is easy to see that a dictionary designed for compressing LaTeX documents would not work very well when compressing C programs. However, generally we want techniques that will be able to compress a variety of source outputs. If we wanted to compress computer files, we do not want to change techniques based on the content of the file. Rather, we would like the technique to *adapt* to the characteristics of the source output. We discuss adaptive-dictionary-based techniques in the next section.

5.4 Adaptive Dictionary

Most adaptive-dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 [54] and 1978 [55]. These papers provide two different approaches to adaptively building dictionaries, and each approach has given rise to a number of variations. The approaches based on the 1977 paper are said to belong to the LZ77 family (also known as LZ1), while the approaches based on the 1978 paper are said to belong to the LZ78, or LZ2, family. The transposition of the initials is a historical accident and is a convention we will observe in this book. In the following sections, we first describe an implementation of each approach followed by some of the more well-known variations.

5.4.1 The LZ77 Approach

In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window as shown in Figure 5.1. The window consists of two parts, a *search buffer* that contains a portion of the recently encoded sequence, and a *look-ahead buffer* that contains the next portion of the sequence to be encoded. In Figure 5.1, the search buffer contains eight symbols, while the look-ahead buffer contains seven symbols. In practice, the sizes of the buffers are significantly larger; however, for the purpose of explanation, we will keep the buffer sizes small.

To encode the sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the look-ahead

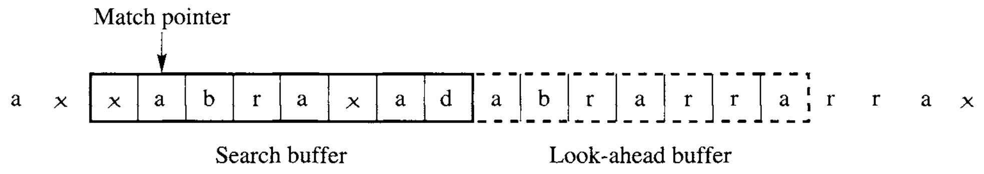

FIGURE 5.1 Encoding using the LZ77 approach.

buffer. The distance of the pointer from the look-ahead buffer is called the *offset*. The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the length of the match. The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triple $\langle o, l, c\rangle$, where o is the offset, l is the length of the match, and c is the codeword corresponding to the symbol in the look-ahead buffer that follows the match. For example, in Figure 5.1 the pointer is pointing to the beginning of the longest match. The offset o in this case is 7, the length of the match l is 4, and the symbol in the look-ahead buffer following the match is p.

The reason for sending the third element in the triple is to take care of the situation where no match for the symbol in the look-ahead buffer can be found in the search buffer. In this case, the offset and match-length values are set to 0, and the third element of the triple is the code for the symbol itself.

If the size of the search buffer is S, the size of the window (search and look-ahead buffers) is W, and the size of the source alphabet is A, then the number of bits needed to code the triple using fixed-length codes is $\lceil \log_2 S\rceil + \lceil \log_2 W\rceil + \lceil \log_2 A\rceil$. Notice that the second term is $\lceil \log_2 W\rceil$, not $\lceil \log_2 S\rceil$. The reason for this is that the length of the match can actually exceed the length of the search buffer. We will see how this happens in Example 5.4.1.

In the following example, we will look at three different possibilities that may be encountered during the coding process:

1. There is no match for the next character to be encoded in the window.
2. There is a match.
3. The matched string extends inside the look-ahead buffer.

Example 5.4.1: The LZ77 approach

Suppose the sequence to be encoded is

$$\ldots cabracadabrarrarrad \ldots$$

Suppose the length of the window is 13, the size of the look-ahead buffer is six, and the current condition is as follows:

cabraca	*dabrar*

with *dabrar* in the look-ahead buffer. We look back in the already encoded portion of the window to find a match for d. As we can see, there is no match, so we transmit the triple $\langle 0, 0, C(d)\rangle$. The first two elements of the triple show that there is no match to d in the search buffer, while $C(d)$ is the code for the character d. This seems like a wasteful way to encode a single character, and we will have more to say about this later.

For now, let's continue with the encoding process. As we have encoded a single character, we move the window by one character. Now the contents of the buffer are

$$\boxed{abracad}\boxed{abrarr}$$

with *abrarr* in the look-ahead buffer. Looking back from the current location, we find a match to *a* at an offset of two. The length of this match is one. Looking further back, we have another match for *a* at an offset of four; again the length of the match is one. Looking back even further in the window, we have a third match for *a* at an offset of seven. However, this time the length of the match is four (see Figure 5.2). So we encode the string *abra* with the triple $\langle 7, 4, C(r)\rangle$, and move the window forward by five characters. The window now contains the following characters:

$$\boxed{adabrar}\boxed{rarrad}$$

Now the look-ahead buffer contains the string *rarrad*. Looking back in the window, we find a match for *r* at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.

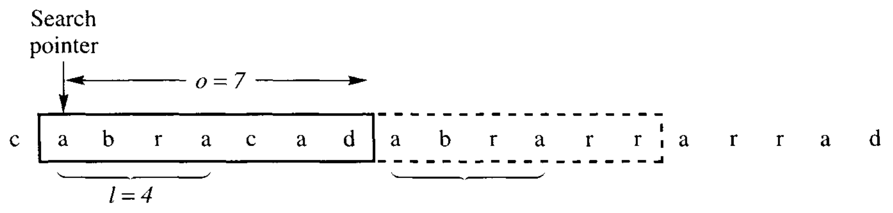

FIGURE 5. 2 **The encoding process.**

Why this is so will become clearer when we decode the sequence. To see how the decoding works, let us assume that we have decoded the sequence *cabraca* and we receive the triples $\langle 0, 0, C(d)\rangle$, $\langle 7, 4, C(r)\rangle$, and $\langle 3, 5, C(d)\rangle$. The first triple is easy to decode; there was no match within the previously decoded string, and the next symbol is *d*. The decoded string is now *cabracad*. The first element of the next triple tells the decoder to move the copy pointer back seven characters, and copy four characters from that point. The decoding process works as shown in Figure 5.3.

Finally, let's see how the triple $\langle 3, 5, C(d)\rangle$ gets decoded. We move back three characters and start copying. The first three characters we copy are *rar*. The copy pointer moves once again, as shown in Figure 5.4, to copy the recently copied character *r*. Similarly, we copy the next character *a*. Even though we started copying only three characters back, we end up decoding five characters. Notice that the match only has to *start* in the search buffer; it can extend into the look-ahead buffer. In fact, if the last character in the look-ahead buffer

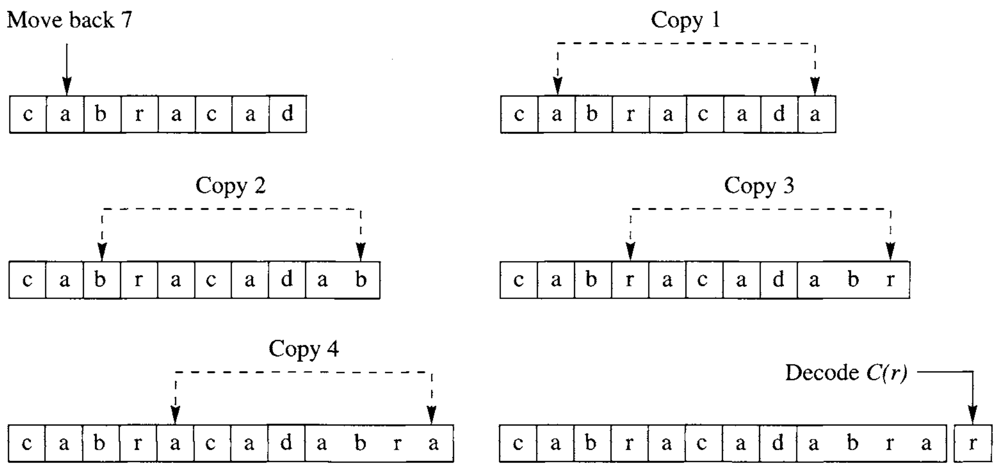

FIGURE 5.3 **Decoding of the triple ⟨7, 4, *C*(*r*)⟩.**

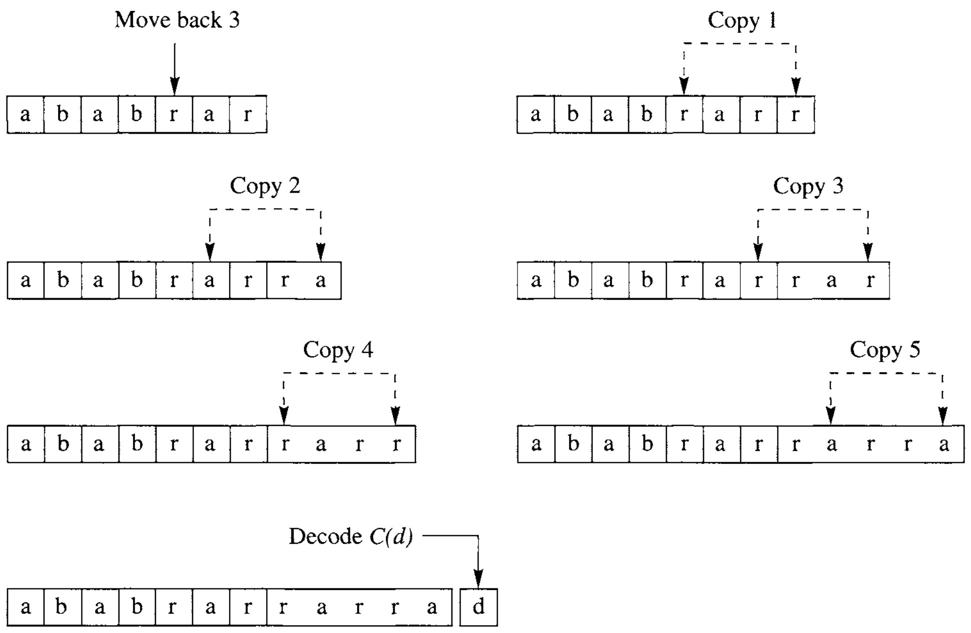

FIGURE 5.4 **Decoding the triple ⟨3, 5, *C*(*d*)⟩.**

had been r instead of d, followed by several more repetitions of rar, the entire sequence of repeated rars could have been encoded with a single triple. ♦

As we can see, the LZ77 scheme is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. The authors of this algorithm showed that asymptotically the performance of this algorithm approached the best that could be obtained by using a scheme that had full knowledge about the statistics of the source. While this may be true asymptotically, in practice there are a number of ways of improving the performance of the LZ77 algorithm as described here. Furthermore, by using the recent portions of the sequence, there is an assumption of sorts being used here—that is, that patterns recur "close" together. As we shall see, in LZ78 the authors removed this "assumption" and came up with an entirely different adaptive-dictionary-based scheme. Before we get to that, let us look at the different variations of the LZ77 algorithm.

Variations on the LZ77 Theme

There are a number of ways that the LZ77 scheme can be made more efficient, and most of these have appeared in the literature. Many of the improvements deal with the efficient encoding of the triples. In the description of the LZ77 algorithm, we assumed that the triples were encoded using a fixed-length code. However, if we were willing to accept more complexity, we could encode the triples using variable-length codes. As we saw in earlier chapters, these codes can be adaptive or, if we were willing to use a two-pass algorithm, they can be semiadaptive. Popular compression packages, such as PKZip, Zip, LHarc, PNG, gzip, and ARJ, all use an LZ77-based algorithm followed by a variable-length coder.

Other variations on the LZ77 algorithm include varying the size of the search and look-ahead buffers. To make the search buffer large requires the development of more effective search strategies. Such strategies can be implemented more effectively if the contents of the search buffer are stored in a manner conducive to fast searches.

The simplest modification to the LZ77 algorithm, and one that is used by most variations of the LZ77 algorithm, is to eliminate the situation where we use a triple to encode a single character. Use of a triple is highly inefficient, especially if a large number of characters occur infrequently. The modification to get rid of this inefficiency is simply the addition of a flag bit, to indicate whether what follows is the codeword for a single symbol. By using this flag bit we also get rid of the necessity for the third element of the triple. Now all we need to do is to send a pair of values corresponding to the offset and length of match. This modification to the LZ77 algorithm is referred to as LZSS [56, 57].

5.4.2 The LZ78 Approach

The LZ77 approach implicitly assumes that like patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding.

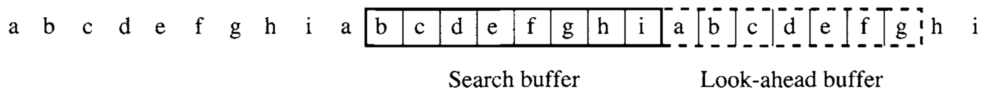

FIGURE 5.5 **The Achilles' heel of LZ77.**

However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured. The worst-case situation would be where the sequence to be encoded was periodic with a period longer than the search buffer. Consider Figure 5.5.

This is a periodic sequence with a period of nine. If the search buffer had been just one symbol longer, this sequence could have been significantly compressed. As it stands, none of the new symbols will have a match in the search buffer and will have to be represented by separate codewords. As this involves sending along overhead (a 1-bit flag for LZSS and a triple for the original LZ77 algorithm), the net result will be an expansion rather than a compression.

Although this is an extreme situation, there are less drastic circumstances in which the finite view of the past would be a drawback. The LZ78 algorithm solves this problem by dropping the reliance on the search buffer and keeping an explicit dictionary. This dictionary has to be built at both the encoder and decoder, and care must be taken that the dictionaries are built in an identical manner. The inputs are coded as a double $\langle i, c\rangle$, with i being an index corresponding to the dictionary entry that was the longest match to the input, and c being the code for the character in the input following the matched portion of the input. As in the case of LZ77, the index value of 0 is used in the case of no match. This double then becomes the newest entry in the dictionary. Thus, each new entry into the dictionary is one new symbol concatenated with an existing dictionary entry. To see how the LZ78 algorithm works, consider the following example.

Example 5.4.2: The LZ78 approach

Let us encode the following sequence using the LZ78 approach:

$$wabba\not{b}wabba\not{b}wabba\not{b}wabba\not{b}woo\not{b}woo\not{b}woo^2$$

where $\not{b}$ stands for space. Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to 0. The first three encoder outputs are $\langle 0, C(w)\rangle$, $\langle 0, C(a)\rangle$, $\langle 0, C(b)\rangle$, and the dictionary looks like Table 5.4.

The fourth symbol is a b, which is the third entry in the dictionary. If we append the next symbol, we would get the pattern ba, which is not in the dictionary, so we encode these two symbols as $\langle 3, C(a)\rangle$, and add the pattern ba as the fourth entry in the dictionary. Continuing in this fashion, the encoder output and the dictionary develop as in Table 5.5. Notice that the entries in the dictionary generally keep getting longer, and if this particular

[2] "The Monster Song" from *Sesame Street.*

TABLE 5.4 The initial dictionary.

Index	Entry
1	w
2	a
3	b

TABLE 5.5 Development of dictionary.

	Dictionary	
Encoder Output	Index	Entry
$\langle 0, C(w)\rangle$	1	w
$\langle 0, C(a)\rangle$	2	a
$\langle 0, C(b)\rangle$	3	b
$\langle 3, C(a)\rangle$	4	ba
$\langle 0, C(\cancel{b})\rangle$	5	$\cancel{b}$
$\langle 1, C(a)\rangle$	6	wa
$\langle 3, C(b)\rangle$	7	bb
$\langle 2, C(\cancel{b})\rangle$	8	$a\cancel{b}$
$\langle 6, C(b)\rangle$	9	wab
$\langle 4, C(\cancel{b})\rangle$	10	$ba\cancel{b}$
$\langle 9, C(b)\rangle$	11	$wabb$
$\langle 8, C(w)\rangle$	12	$a\cancel{b}w$
$\langle 0, C(o)\rangle$	13	o
$\langle 13, C(\cancel{b})\rangle$	14	$o\cancel{b}$
$\langle 1, C(o)\rangle$	15	wo
$\langle 14, C(w)\rangle$	16	$o\cancel{b}w$
$\langle 13, C(o)\rangle$	17	oo

sentence was repeated often, as it is in the song, after a while the entire sentence would be an entry in the dictionary. ♦

While the LZ78 algorithm has the ability to capture patterns and hold them indefinitely, it also has a rather serious drawback. As seen from the example, the dictionary keeps growing without bound. In a practical situation, we would have to stop the growth of the dictionary at some stage, and then either prune it back or treat the encoding as a fixed dictionary scheme. We will discuss some possible approaches when we study applications of dictionary coding.

Variations on the LZ78 Theme—The LZW Algorithm

There are a number of ways the LZ78 algorithm can be modified, and as is the case with the LZ77 algorithm, anything that can be modified probably has been. The most well-known modification, one that initially sparked much of the interest in the LZ algorithms, is a modification by Terry Welch known as LZW [58]. Welch proposed a technique for removing the necessity of encoding the second element of the pair $\langle i, c\rangle$. That is, the encoder would only send the index to the dictionary. In order to do this, the dictionary has to be primed with all the letters of the source alphabet. The input to the encoder is accumulated in a pattern p as long as p is contained in the dictionary. If the addition of another letter a results in a pattern $p * a$ ($*$ denotes concatenation) that is not in the dictionary, then the index of p is transmitted to the receiver, the pattern $p * a$ is added to the dictionary, and we start another pattern with the letter a. The LZW algorithm is best understood with an example. In the following two examples, we will look at the encoder and decoder operations for the same sequence used to explain the LZ78 algorithm.

Example 5.4.3: The LZW algorithm—encoding

We will use the sequence previously used to demonstrate the LZ78 algorithm as our input sequence:

wabbab̸wabbab̸wabbab̸wabbab̸woob̸woob̸woo

Assuming that the alphabet for the source is {*b̸*, *a*, *b*, *o*, *w*}, the LZW dictionary initially looks like Table 5.6.

TABLE 5.6 Initial LZW dictionary.

Index	Entry
1	*b̸*
2	*a*
3	*b*
4	*o*
5	*w*

The encoder first encounters the letter *w*. This "pattern" is in the dictionary so we concatenate the next letter to it, forming the pattern *wa*. This pattern is not in the dictionary, so we encode *w* with its dictionary index 5, add the pattern *wa* to the dictionary as the sixth element of the dictionary, and begin a new pattern starting with the letter *a*. As *a* is in the dictionary, we concatenate the next element *b* to form the pattern *ab*. This pattern is not in the dictionary, so we encode *a* with its dictionary index value 2, add the pattern *ab* to the dictionary as the seventh element of the dictionary, and start constructing a new pattern with the letter *b*. We continue in this manner, constructing two-letter patterns, until we reach the letter *w* in the second *wabba*. At this point the output of the encoder consists entirely of indices from the initial dictionary: 5 2 3 3 2 1. The dictionary at this point looks like Table 5.7. (The 12th entry in the dictionary is still under construction.) The next symbol in the sequence is *a*. Concatenating this to *w*, we get the pattern *wa*. This pattern already exists in the dictionary (item 6), so we read the next symbol, which is *b*. Concatenating this to *wa*, we get the pattern *wab*. This pattern does not exist in the dictionary, so we include it as the 12th entry in the dictionary and start a new pattern with the symbol *b*. We also encode *wa* with its index value of 6. Notice that after a series of two-letter entries, we now have a three-letter entry. As the encoding progresses, the length of the entries keeps increasing. The longer entries in the dictionary indicate that the dictionary is capturing more of the structure in the sequence. The dictionary at the end of the encoding process is shown in Table 5.8. Notice that the 12th through the 19th entries are all either three or four letters in length. Then we encounter the pattern *woo* for the first time and we drop back to two-letter patterns for three more entries, after which we go back to entries of increasing length.

TABLE 5.7 **Constructing the 12th entry of the LZW dictionary.**

Index	Entry
1	*b̸*
2	*a*
3	*b*
4	*o*
5	*w*
6	*wa*
7	*ab*
8	*bb*
9	*ba*
10	*ab̸*
11	*b̸*w
12	*w*...

TABLE 5.8 **The LZW dictionary for encoding wabbab̸wabbab̸wabbab̸wabbab̸woob̸woob̸woo.**

Index	Entry	Index	Entry
1	*b̸*	14	*ab̸w*
2	*a*	15	*wabb*
3	*b*	16	*bab̸*
4	*o*	17	*b̸wa*
5	*w*	18	*abb*
6	*wa*	19	*bab̸w*
7	*ab*	20	*wo*
8	*bb*	21	*oo*
9	*ba*	22	*ob̸*
10	*ab̸*	23	*b̸wo*
11	*b̸w*	24	*oob̸*
12	*wab*	25	*b̸woo*
13	*bba*		

The encoder output sequence is 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4. ♦

Example 5.4.4: The LZW algorithm—decoding

In this example we will take the encoder output from the previous example and decode it using the LZW algorithm. The encoder output sequence in the previous example was

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

This becomes the decoder input sequence. The decoder starts with the same initial dictionary as the encoder (Table 5.6).

The index value 5 corresponds to the letter *w*, so we decode *w* as the first element of our sequence. At the same time, in order to mimic the dictionary construction procedure of the encoder, we begin construction of the next element of the dictionary. We start with the letter *w*. This pattern exists in the dictionary, so we do not add it to the dictionary and continue with the decoding process. The next decoder input is 2, which is the index corresponding to the letter *a*. We decode an *a* and concatenate it with our current pattern to form the pattern *wa*. As this does not exist in the dictionary, we add it as the sixth element of the dictionary and start a new pattern beginning with the letter *a*. The next four inputs 3 3 2 1 correspond to the letters *bbab̸* and generate the dictionary entries *ab*, *bb*, *ba*, and *ab̸*. The dictionary now looks like Table 5.9, where the 11th entry is under construction.

TABLE 5.9 Constructing the 11th entry of the LZW dictionary while decoding.

Index	Entry
1	*b̸*
2	*a*
3	*b*
4	*o*
5	*w*
6	*wa*
7	*ab*
8	*bb*
9	*ba*
10	*ab̸*
11	*b̸*. . .

The next input is 6, which is the index of the pattern *wa*. Therefore, we decode a *w* and an *a*. We first concatenate *w* to the existing pattern, which is *b̸*, and form the pattern *b̸w*. As *b̸w* does not exist in the dictionary, it becomes the 11th entry. The new pattern now starts with the letter *w*. We had previously decoded the letter *a*, which we now concatenate to *w* to obtain the pattern *wa*. This pattern is contained in the dictionary, so we decode the next input, which is 8. This corresponds to the entry *bb* in the dictionary. We decode the first *b* and concatenate it to the pattern *wa* to get the pattern *wab*. This pattern does not exist in the dictionary, so we add it as the 12th entry in the dictionary and start a new pattern with the letter *b*. Decoding the second *b* and concatenating it to the new pattern, we get the pattern *bb*. This pattern exists in the dictionary, so we decode the next element in the sequence of encoder outputs. Continuing in this fashion, we can decode the entire sequence. Notice that the dictionary being constructed by the decoder is identical to that constructed by the encoder. ♦

There is one particular situation in which the method of decoding the LZW algorithm described above breaks down. Suppose we had a source with an alphabet $\mathcal{A} = \{a, b\}$, and we were to encode the sequence beginning with *abababab*.... The encoding process is still the same. We begin with the initial dictionary shown in Table 5.10 and end up with the final dictionary shown in Table 5.11.

The transmitted sequence is 1 2 3 5.... This looks like a relatively straightforward sequence to decode. However, when we try to do so, we run into a snag. Let us go through the decoding process and see what happens.

We begin with the same initial dictionary as the encoder (Table 5.10). The first two elements in the received sequence 1 2 3 5... are decoded as a and b, giving rise to the third dictionary entry ab, and the beginning of the next pattern to be entered in the dictionary, b. The dictionary at this point is shown in Table 5.12.

TABLE 5.10 **Initial dictionary for *abababab*.**

Index	Entry
1	a
2	b

TABLE 5.11 **Final dictionary for *abababab*.**

Index	Entry
1	a
2	b
3	ab
4	ba
5	aba
6	$abab$
7	$b\ldots$

TABLE 5.12 **Constructing the fourth entry of the dictionary while decoding.**

Index	Entry
1	a
2	b
3	ab
4	$b\ldots$

TABLE 5.13 **Constructing the fifth entry (stage one).**

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*a*...

TABLE 5.14 **Constructing the fifth entry (stage two).**

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*ab*...

The next input to the decoder is 3. This corresponds to the dictionary entry *ab*. Decoding each in turn, we first concatenate *a* to the pattern under construction to get *ba*. This pattern is not contained in the dictionary, so we add this to the dictionary (keep in mind, we have not used the *b* from *ab* yet), which now looks like Table 5.13.

The new entry starts with the letter *a*. We have only used the first letter from the pair *ab*. Therefore, we now concatenate *b* to *a* to obtain the pattern *ab*. This pattern is contained in the dictionary, so we continue with the decoding process. The dictionary at this stage looks like Table 5.14.

The first four entries in the dictionary are complete, while the fifth entry is still under construction. However, the very next input to the decoder is 5, which corresponds to the incomplete entry! How do we decode an index for which we do not as yet have a complete dictionary entry?

The situation is actually not as bad as it looks. (Of course, if it were, we would not now be studying LZW.) While we may not have a fifth entry for the dictionary, we do have the beginnings of the fifth entry, which is *ab*.... Let us, for the moment, pretend that we do indeed have the fifth entry and continue with the decoding process. If we had a fifth entry, the first two letters of the entry would be *a* and *b*. Concatenating *a* to the partial new entry we get the pattern *aba*. This pattern is not contained in the dictionary, so we add this to our dictionary, which now looks like Table 5.15. Notice that we now have the fifth entry in the dictionary, which is *aba*. We have already decoded the *ab* portion of *aba*. We can now decode the last letter *a* and continue on our merry way.

This means that the LZW decoder has to contain an exception handler to handle the special case of decoding an index that does not have a corresponding complete entry in the decoder dictionary.

TABLE 5.15 Completion of the fifth entry.

Index	Entry
1	a
2	b
3	ab
4	ba
5	aba
6	$a\ldots$

5.5 Applications

Since the publication of Terry Welch's article [58], there has been a steadily increasing number of applications that use some variant of the LZ78 algorithm. Among the LZ78 variants, by far the most popular is the LZW algorithm. In this section we describe two of the best-known applications of LZW: GIF, and V.42 bis. While the LZW algorithm was initially the algorithm of choice patent concerns has lead to increasing use of the LZ77 algorithm. The most popular implementation of the LZ77 algorithm is the *deflate* algorithm initially designed by Phil Katz. It is part of the popular *zlib* library developed by Jean-loup Gailly and Mark Adler. Jean-loup Gailly also used deflate in the widely used *gzip* algorithm. The *deflate* algorithm is also used in PNG which we describe below.

5.5.1 File Compression—UNIX `compress`

The UNIX `compress` command is one of the earlier applications of LZW. The size of the dictionary is adaptive. We start with a dictionary of size 512. This means that the transmitted codewords are 9 bits long. Once the dictionary has filled up, the size of the dictionary is doubled to 1024 entries. The codewords transmitted at this point have 10 bits. The size of the dictionary is progressively doubled as it fills up. In this way, during the earlier part of the coding process when the strings in the dictionary are not very long, the codewords used to encode them also have fewer bits. The maximum size of the codeword, $b_{\max}$, can be set by the user to between 9 and 16, with 16 bits being the default. Once the dictionary contains $2^{b_{\max}}$ entries, `compress` becomes a static dictionary coding technique. At this point the algorithm monitors the compression ratio. If the compression ratio falls below a threshold, the dictionary is flushed, and the dictionary building process is restarted. This way, the dictionary always reflects the local characteristics of the source.

5.5.2 Image Compression—The Graphics Interchange Format (GIF)

The Graphics Interchange Format (GIF) was developed by Compuserve Information Service to encode graphical images. It is another implementation of the LZW algorithm and is very similar to the `compress` command. The compressed image is stored with the first byte

TABLE 5.16 Comparison of GIF with arithmetic coding.

Image	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	51,085	53,431	31,847
Sensin	60,649	58,306	37,126
Earth	34,276	38,248	32,137
Omaha	61,580	56,061	51,393

being the minimum number of bits b per pixel in the original image. For the images we have been using as examples, this would be eight. The binary number 2^b is defined to be the *clear code*. This code is used to reset all compression and decompression parameters to a start-up state. The initial size of the dictionary is 2^{b+1}. When this fills up, the dictionary size is doubled, as was done in the `compress` algorithm, until the maximum dictionary size of 4096 is reached. At this point the compression algorithm behaves like a static dictionary algorithm. The codewords from the LZW algorithm are stored in blocks of characters. The characters are 8 bits long, and the maximum block size is 255. Each block is preceded by a header that contains the block size. The block is terminated by a block terminator consisting of eight 0s. The end of the compressed image is denoted by an end-of-information code with a value of $2^b + 1$. This codeword should appear before the block terminator.

GIF has become quite popular for encoding all kinds of images, both computer-generated and "natural" images. While GIF works well with computer-generated graphical images, and pseudocolor or color-mapped images, it is generally not the most efficient way to losslessly compress images of natural scenes, photographs, satellite images, and so on. In Table 5.16 we give the file sizes for the GIF-encoded test images. For comparison, we also include the file sizes for arithmetic coding the original images and arithmetic coding the differences.

Notice that even if we account for the extra overhead in the GIF files, for these images GIF barely holds its own even with simple arithmetic coding of the original pixels. While this might seem odd at first, if we examine the image on a pixel level, we see that there are very few repetitive patterns compared to a text source. Some images, like the Earth image, contain large regions of constant values. In the dictionary coding approach, these regions become single entries in the dictionary. Therefore, for images like these, the straight forward dictionary coding approach does hold its own. However, for most other images, it would probably be preferable to perform some preprocessing to obtain a sequence more amenable to dictionary coding. The PNG standard described next takes advantage of the fact that in natural images the pixel-to-pixel variation is generally small to develop an appropriate preprocessor. We will also revisit this subject in Chapter 7.

5.5.3 Image Compression—Portable Network Graphics (PNG)

The PNG standard is one of the first standards to be collaboratively developed over the Internet. The impetus for it was an announcement in December 1994 by Unisys (which had acquired the patent for LZW from Sperry) and CompuServe that they would start charging royalties to authors of software that included support for GIF. The announcement resulted in an uproar in the segment of the compression community that formed the core of the Usenet group comp.compression. The community decided that a patent-free replacement for GIF should be developed, and within three months PNG was born. (For a more detailed history of PNG as well as software and much more, go to the PNG website maintained by Greg Roelof, http://www.libpng.org/pub/png/.)

Unlike GIF, the compression algorithm used in PNG is based on LZ77. In particular, it is based on the *deflate* [59] implementation of LZ77. This implementation allows for match lengths of between 3 and 258. At each step the encoder examines three bytes. If it cannot find a match of at least three bytes it puts out the first byte and examines the next three bytes. So, at each step it either puts out the value of a single byte, or literal, or the pair $< match\ length,\ offset >$. The alphabets of the *literal* and *match length* are combined to form an alphabet of size 286 (indexed by $0--285$). The indices $0--255$ represent literal bytes and the index 256 is an end-of-block symbol. The remaining 29 indices represent codes for ranges of lengths between 3 and 258, as shown in Table 5.17. The table shows the index, the number of selector bits to follow the index, and the lengths represented by the index and selector bits. For example, the index 277 represents the range of lengths from 67 to 82. To specify which of the sixteen values has actually occurred, the code is followed by four selector bits.

The index values are represented using a Huffman code. The Huffman code is specified in Table 5.18.

The *offset* can take on values between 1 and 32,768. These values are divided into 30 ranges. The thirty range values are encoded using a Huffman code (different from the Huffman code for the *literal* and *length* values) and the code is followed by a number of selector bits to specify the particular distance within the range.

We have mentioned earlier that in natural images there is not great deal of repetition of sequences of pixel values. However, pixel values that are spatially close also tend to have values that are similar. The PNG standard makes use of this structure by estimating the value of a pixel based on its causal neighbors and subtracting this estimate from the pixel. The difference modulo 256 is then encoded in place of the original pixel. There are four different ways of getting the estimate (five if you include no estimation), and PNG allows

TABLE 5.17 Codes for representations of *match length* [59].

Index	# of selector bits	Length	Index	# of selector bits	Length	Index	# of selector bits	Length
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11, 12	275	3	51–58	285	0	258
266	1	13, 14	276	3	59–66			

TABLE 5.18 Huffman codes for the *match length* alphabet [59].

Index Ranges	# of bits	Binary Codes
0–143	8	00110000 through 10111111
144–255	9	110010000 through 111111111
256–279	7	0000000 through 0010111
280–287	8	11000000 through 11000111

TABLE 5.19 Comparison of PNG with GIF and arithmetic coding.

Image	PNG	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	31,577	51,085	53,431	31,847
Sensin	34,488	60,649	58,306	37,126
Earth	26,995	34,276	38,248	32,137
Omaha	50,185	61,580	56,061	51,393

the use of a different method of estimation for each row. The first way is to use the pixel from the row above as the estimate. The second method is to use the pixel to the left as the estimate. The third method uses the average of the pixel above and the pixel to the left. The final method is a bit more complex. An initial estimate of the pixel is first made by adding the pixel to the left and the pixel above and subtracting the pixel to the upper left. Then the pixel that is closest to the initial esitmate (upper, left, or upper left) is taken as the estimate. A comparison of the performance of PNG and GIF on our standard image set is shown in Table 5.19. The PNG method clearly outperforms GIF.

5.5.4 Compression over Modems—V.42 bis

The ITU-T Recommendation V.42 bis is a compression standard devised for use over a telephone network along with error-correcting procedures described in CCITT Recommendation V.42. This algorithm is used in modems connecting computers to remote users. The algorithm described in this recommendation operates in two modes, a transparent mode and a compressed mode. In the transparent mode, the data are transmitted in uncompressed form, while in the compressed mode an LZW algorithm is used to provide compression.

The reason for the existence of two modes is that at times the data being transmitted do not have repetitive structure and therefore cannot be compressed using the LZW algorithm. In this case, the use of a compression algorithm may even result in expansion. In these situations, it is better to send the data in an uncompressed form. A random data stream would cause the dictionary to grow without any long patterns as elements of the dictionary. This means that most of the time the transmitted codeword would represent a single letter from the source alphabet. As the dictionary size is much larger than the source alphabet size, the number of bits required to represent an element in the dictionary is much more than the number of bits required to represent a source letter. Therefore, if we tried to compress a sequence that does not contain repeating patterns, we would end up with more bits to transmit than if we had not performed any compression. Data without repetitive structure are often encountered when a previously compressed file is transferred over the telephone lines.

The V.42 bis recommendation suggests periodic testing of the output of the compression algorithm to see if data expansion is taking place. The exact nature of the test is not specified in the recommendation.

In the compressed mode, the system uses LZW compression with a variable-size dictionary. The initial dictionary size is negotiated at the time a link is established between the transmitter and receiver. The V.42 bis recommendation suggests a value of 2048 for the dictionary size. It specifies that the minimum size of the dictionary is to be 512. Suppose the initial negotiations result in a dictionary size of 512. This means that our codewords that are indices into the dictionary will be 9 bits long. Actually, the entire 512 indices do not correspond to input strings; three entries in the dictionary are reserved for control codewords. These codewords in the compressed mode are shown in Table 5.20.

When the numbers of entries in the dictionary exceed a prearranged threshold C_3, the encoder sends the STEPUP control code, and the codeword size is incremented by 1 bit. At the same time, the threshold C_3 is also doubled. When all available dictionary entries are filled, the algorithm initiates a reuse procedure. The location of the first string entry in the dictionary is maintained in a variable N_5. Starting from N_5, a counter C_1 is incremented until it finds a dictionary entry that is not a prefix to any other dictionary entry. The fact that this entry is not a prefix to another dictionary entry means that this pattern has not been encountered since it was created. Furthermore, because of the way it was located, among patterns of this kind this pattern has been around the longest. This reuse procedure enables the algorithm to prune the dictionary of strings that may have been encountered in the past but have not been encountered recently, on a continual basis. In this way the dictionary is always matched to the current source statistics.

To reduce the effect of errors, the CCITT recommends setting a maximum string length. This maximum length is negotiated at link setup. The CCITT recommends a range of 6–250, with a default value of 6.

The V.42 bis recommendation avoids the need for an exception handler for the case where the decoder receives a codeword corresponding to an incomplete entry by forbidding the use of the last entry in the dictionary. Instead of transmitting the codeword corresponding to the last entry, the recommendation requires the sending of the codewords corresponding

TABLE 5.20 Control codewords in compressed mode.

Codeword	Name	Description
0	ETM	Enter transparent mode
1	FLUSH	Flush data
2	STEPUP	Increment codeword size

to the constituents of the last entry. In the example used to demonstrate this quirk of the LZW algorithm, instead of transmitting the codeword 5, the V.42 bis recommendation would have forced us to send the codewords 3 and 1.

5.6 Summary

In this chapter we have introduced techniques that keep a dictionary of recurring patterns and transmit the index of those patterns instead of the patterns themselves in order to achieve compression. There are a number of ways the dictionary can be constructed.

- In applications where certain patterns consistently recur, we can build application-specific static dictionaries. Care should be taken not to use these dictionaries outside their area of intended application. Otherwise, we may end up with data expansion instead of data compression.
- The dictionary can be the source output itself. This is the approach used by the LZ77 algorithm. When using this algorithm, there is an implicit assumption that recurrence of a pattern is a local phenomenon.
- This assumption is removed in the LZ78 approach, which dynamically constructs a dictionary from patterns observed in the source output.

Dictionary-based algorithms are being used to compress all kinds of data; however, care should be taken with their use. This approach is most useful when structural constraints restrict the frequently occurring patterns to a small subset of all possible patterns. This is the case with text, as well as computer-to-computer communication.

Further Reading

1. *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1], provides an excellent exposition of dictionary-based coding techniques.
2. *The Data Compression Book*, by M. Nelson and J.-L. Gailley [60], also does a good job of describing the Ziv-Lempel algorithms. There is also a very nice description of some of the software implementation aspects.
3. *Data Compression*, by G. Held and T.R. Marshall [61], contains a description of digram coding under the name "diatomic coding." The book also includes BASIC programs that help in the design of dictionaries.
4. The PNG algorithm is described in a very accessible manner in "PNG Lossless Compression," by G. Roelofs [62] in the *Lossless Compression Handbook*.
5. A more in-depth look at dictionary compression is provided in "Dictionary-Based Data Compression: An Algorithmic Perspective," by S.C. Şahinalp and N.M. Rajpoot [63] in the *Lossless Compression Handbook*.

5.7 Projects and Problems

1. To study the effect of dictionary size on the efficiency of a static dictionary technique, we can modify Equation (5.1) so that it gives the rate as a function of both p and the dictionary size M. Plot the rate as a function of p for different values of M, and discuss the trade-offs involved in selecting larger or smaller values of M.

2. Design and implement a digram coder for text files of interest to you.

   (a) Study the effect of the dictionary size, and the size of the text file being encoded on the amount of compression.

   (b) Use the digram coder on files that are not similar to the ones you used to design the digram coder. How much does this affect your compression?

3. Given an initial dictionary consisting of the letters *a b r y ƀ*, encode the following message using the LZW algorithm: *aƀbarƀarrayƀbyƀbarrayarƀbay*.

4. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.21.

TABLE 5.21 **Initial dictionary for Problem 4.**

Index	Entry
1	*a*
2	*ƀ*
3	*h*
4	*i*
5	*s*
6	*t*

   (a) The output of the LZW encoder is the following sequence:

6	3	4	5	2	3	1	6	2	9	11	16	12	14	4	20	10	8	23	13

   Decode this sequence.

   (b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

5. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.22.

   (a) The output of the LZW encoder is the following sequence:

3	1	4	6	8	4	2	1	2	5	10	6	11	13	6

   Decode this sequence.

TABLE 5.22 **Initial dictionary for Problem 5.**

Index	Entry
1	a
2	$b\!\!/$
3	r
4	t

(b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

6. Encode the following sequence using the LZ77 algorithm:

$$barrayarb\!\!/barb\!\!/byb\!\!/barrayarb\!\!/bay$$

Assume you have a window size of 30 with a look-ahead buffer of size 15. Furthermore, assume that $C(a) = 1$, $C(b) = 2$, $C(b\!\!/) = 3$, $C(r) = 4$, and $C(y) = 5$.

7. A sequence is encoded using the LZ77 algorithm. Given that $C(a) = 1$, $C(b\!\!/) = 2$, $C(r) = 3$, and $C(t) = 4$, decode the following sequence of triples:

$$\langle 0,0,3\rangle\ \langle 0,0,1\rangle\ \langle 0,0,4\rangle\ \langle 2,8,2\rangle\ \langle 3,1,2\rangle\ \langle 0,0,3\rangle\ \langle 6,4,4\rangle\ \langle 9,5,4\rangle$$

Assume that the size of the window is 20 and the size of the look-ahead buffer is 10. Encode the decoded sequence and make sure you get the same sequence of triples.

8. Given the following primed dictionary and the received sequence below, build an LZW dictionary *and* decode the transmitted sequence.

Received Sequence: 4, 5, 3, 1, 2, 8, 2, 7, 9, 7, 4

Decoded Sequence:______________________________________

Initial dictionary:

(a) S

(b) $b\!\!/$

(c) I

(d) T

(e) H