Lempel-Ziv Encoding

In many texts certain sequences of characters occur with high frequency. In English, for example, the word "the" occurs more often than any other sequence of three letters, with "and", "ion", and "ing" close behind. If we include the space character, there are other very common sequences, including longer ones like "of the". Although it is impossible to improve on Huffman encoding with any method that assigns a fixed encoding to each character, we can do better by encoding entire *sequences* of characters with just a few bits. The method of this section takes advantage of frequently occurring character sequences of any length. It typically produces an even smaller representation than is possible with Huffman trees, and unlike basic Huffman encoding it reads through the text only once and requires no extra space for overhead in the compressed representation.

The algorithm makes use of a "dictionary" that stores character sequences chosen dynamically from *w.* With each character sequence the dictionary associates a number; if *s* is a character sequence, we use *#(s)* to denote the number assigned to s by the dictionary. The number *#(s)* is called the code or code number of *s.* All codes have the same length in bits; a typical code size is twelve bits, which permits a maximum dictionary size of $2^{12} = 4096$ character sequences. The dictionary is initialized with all possible one-character sequences, that is, the elements of $\sum$ are assigned the code numbers 0 through $|\sum| - 1$ and all other code numbers are initially unassigned.

The text *w* is encoded using a greedy heuristic: at each step, determine the longest prefix p of *w* that is in the dictionary, output the code number of p, and remove p from the front of *w;* call p the **current match.** At each step we also modify the dictionary by adding a new string and assigning it the next unused code number. (We'll consider later the problem of what to do if the dictionary fills up, leaving no code numbers available.) The string to be added consists of the current match concatenated to the first character of the remainder of *w.* It turns out to be simpler to wait until the next step to add this string; that is, at each step we determine the current match, then add to the dictionary the match from the *previous* step concatenated to the first character of the current match. No string is added to the dictionary in the very first step.

Figure *5.9* demonstrates this process on the (admittedly contrived) example string COCOA AND BANANAS. In the first step #(C) is output and nothing is inserted in the dictionary. In the next step O is matched, so #(O) is output and CO is inserted in the dictionary. In step 3 the sequence CO is found in the dictionary, so its code is output and OC is inserted in the dictionary. Continuing in this way, fourteen codes are output to encode the example string. When very long strings are compressed by this method, longer and longer sequences are added to the dictionary; eventually, short code numbers can represent very long strings. Moreover, the dictionary becomes "tailored" to *w* because of the way strings are chosen for inclusion. When *w* consists of English text, for example, the words and even phrases that appear often in *w* eventually find their way into the dictionary and are subsequently encoded as single code numbers.

Step	Output	Add to Dictionary	Step	Output	Add to Dictionary
1	#(C)	_	8	#(D)	ND
2	#(O)	CO	9	#(□)	D□
3	#(CO)	OC	10	#(B)	□B
4	#(A)	COA	11	#(AN)	BA
5	#(□)	A□	12	#(AN)	ANA
6	#(A)	□A	13	#(A)	ANA
7	#(N)	AN	14	#(S)	AS

Figure 5.9 Lempel-Ziv encoding of COCOA AND BANANAS. The symbol □ denotes the space character, and *#(s)* is associated with string *s* in the dictionary. Note that duplicate strings may be added to the dictionary.

Decoding is almost the same as encoding. First of all, the compressed representation consists simply of

a sequence of code numbers; it is easy to retrieve them one by one since the length in bits of a single code number is fixed. The dictionary is *not* saved anywhere; as we shall see, the decoding process reconstructs at each step the same dictionary that the encoding process used (as in adaptive Huffman encoding). Consider the example of Figure *5.9* from the point of view of the decoder. It first sees the code for C, which is in the initial dictionary, so it knows that C is the first character of the text. In the next step, it reads the code for 0; like the encoder, it now adds CO to the dictionary. The code number assigned to CO will be correct since both encoder and decoder assign the first unused number to new strings in the dictionary. The general decoding step is similar to the general encoding step: read a code, look it up in the dictionary and output the associated character sequence *s,* then add to the dictionary the sequence consisting of the *previous* sequence concatenated to the first character of a. (The *LookUp* will always succeed, but see Problem 34 for an interesting variation.) The complete decoder is shown in Algorithm 5.3.

Many implementation details remain to be discussed. For example, what should we do when the dictionary is full, that is, when all code numbers have been assigned? There are several possibilities:

- Stop placing new entries in the dictionary, encoding the rest of the text using the dictionary created so far.
- Clear the dictionary completely (except for the one-character sequences) and start afresh, allowing new sequences to accumulate.
- Discard infrequently used sequences from the dictionary and reuse their code numbers. (Of course, this requires keeping some statistical information during encoding and decoding.)
- Switch to larger code numbers. Adding even a single bit doubles the number of available codes. This scheme can be repeated until the dictionary grows too large to be stored in main memory.

```
procedure LZDecode(bitstream b):
{Recover the string encoded in b}
    {D is a dictionary associated code numbers with strings}
    D ← MakeEmpySet()
     nextcode ← 0                    {The next code number to be assigned}
     {Insert each single-character string into the dictionaryl
    foreach c ∈ ∑ do
        Insert(nextcode, c, D)
        nextcode ← nextcode + 1
     {Special first step with no dictionary updates}
        current ← LookUp(ReadOneCodeNumber(b), D)
    Output(current)
    {Main loop}
    until Empty(b) do
        previous ← current
        current ← LookUp(ReadOneCodeNumber(b), D)
        insert(nextcode, Concat (previous , current[0]), D)
        nextcode ← nextcode + 1
        Output(current)
```

Algorithm 5.3 Lempel-Ziv decoding. The functions *MakeEmptySet*, *insert*, and *LookUp* are abstract operations on dictionaries, discussed more fully in Chapter 6. *Output* is an unspecified procedure that handles the encoded string as it is recovered.

(Yet another possibility is discussed on page 484.) The appropriateness of one method over another depends on the amount of storage and processor power available, but also on the characteristics of the data being compressed. For example, it is easy to stop putting new entries in the dictionary, but if the input is very long and of gradually changing character then clearing the dictionary is a better idea. Of course, no matter which method is used it is essential that the encoder and decoder agree on the method so that their dictionaries stay synchronized! A more interesting problem is how best to store the dictionary during

encoding and decoding, to facilitate the special kinds of *LookUps* performed by the encoder (Problem 33). The appropriate data structure is the *trie*, which we discuss in Chapter 8.

To illustrate the potential savings that can be realized with the techniques of pus section, we give here the results of an experiment using several methods of encoding. The text was a near-final version of this book (including commands used for formatting) comprising 1322028 eight-bit characters, for a total of 0576224 bits. A classical two-pass Huffman compression algorithm produced an encoded text of 6469752 bits, about 61% of the size of the original. A one-pass adaptive Huffman algorithm produced 6470800 bits, saving an entire pass through the text at a cost of only about 1000 bits (although requiring nearly an order of magnitude more computation time). But a variant of the Lempel-Ziv method that begins with twelve-bit codes and allows code size to grow yielded a compressed text of 4493168 bits, about 42.5% of the original size.