*MIT 6.02* **DRAFT** *Lecture Notes*
*Fall 2011 (Last update: September 12, 2011)*
*Comments, questions or bug reports?*
*Please contact hari at mit.edu*

# CHAPTER 3
# Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW)

This chapter discusses **source coding**, specifically two algorithms to compress messages (i.e., a sequence of symbols). The first, Huffman coding, is efficient when one knows the probabilities of the different symbols one wishes to send. In the context of Huffman coding, a message can be thought of as a sequence of symbols, with each symbol drawn independently from some known distribution. The second, LZW (for Lempel-Ziv-Welch) is an **adaptive compression** algorithm that does not assume any a priori knowledge of the symbol probabilities. Both Huffman codes and LZW are widely used in practice, and are a part of many real-world standards such as GIF, JPEG, MPEG, MP3, and more.

## ■ 3.1 Properties of Good Source Codes

Suppose the source wishes to send a message, i.e., a sequence of **symbols**, drawn from some alphabet. The alphabet could be text, it could be bit sequences corresponding to a digitized picture or video obtained from a digital or analog source (we will look at an example of such a source in more detail in the next chapter), or it could be something more abstract (e.g., "ONE" if by land and "TWO" if by sea, or $h$ for heavy traffic and $\ell$ for light traffic on a road).

A **code** is a mapping between symbols and **codewords**. The reason for doing the mapping is that we would like to adapt the message into a form that can be manipulated (processed), stored, and transmitted over communication channels. Codewords made of bits ("zeroes and ones") are a convenient and effective way to achieve this goal.

For example, if we want to communicate the grades of students in 6.02, we might use the following encoding:

"A" $\rightarrow$ 1
"B" $\rightarrow$ 01
"C" $\rightarrow$ 000
"D" $\rightarrow$ 001

Then, if we want to transmit a sequence of grades, we might end up sending a message such as 0010001110100001. The receiver can decode this received message as the sequence of grades "DCAAABCB" by looking up the appropriate contiguous and non-overlapping substrings of the received message in the code (i.e., the mapping) shared by it and the source.

**Instantaneous codes.** A useful property for a code to possess is that a symbol corresponding to a received codeword be decodable as soon as the corresponding codeword is received. Such a code is called an **instantaneous code**. The example above is an instantaneous code. The reason is that if the receiver has already decoded a sequence and now receives a "1", then it knows that the symbol *must* be "A". If it receives a "0", then it looks at the next bit; if that bit is "1", then it knows the symbol is "B"; if the next bit is instead "0", then it does not yet know what the symbol is, but the *next* bit determines uniquely whether the symbol is "C" (if "0") or "D" (if "1"). Hence, this code is instantaneous.

**Code trees and prefix-free codes.** A convenient way to visualize codes is using a *code tree*, as shown in Figure 3-1 for an instantaneous code with the following encoding:

"A" $\rightarrow$ 10
"B" $\rightarrow$ 0
"C" $\rightarrow$ 110
"D" $\rightarrow$ 111

In general, a code tree is a binary tree with the symbols at the nodes of the tree and the edges of the tree are labeled with "0" or "1" to signify the encoding. To find the encoding of a symbol, the receiver simply walks the path from the root (the top-most node) to that symbol, emitting the label on the edges traversed.

If, in a code tree, the symbols are all at the leaves, then the code is said to be **prefix-free**, because no codeword is a prefix of another codeword. Prefix-free codes (and code trees) are naturally instantaneous, which makes them attractive.[1]

**Expected code length.** Our final definition is for the expected length of a code. Given $N$ symbols, with symbol $i$ occurring with probability $p_i$, if we have a code in which symbol $i$ has length $l_i$ in the code tree (i.e., the codeword is $\ell_i$ bits long), then the expected length of the code is $\sum_{i=1}^{N} p_i \ell_i$.

In general, codes with small expected code length are interesting and useful because they allow us to **compress** messages, delivering messages without any loss of information but consuming fewer bits than without the code. Because one of our goals in designing communication systems is efficient sharing of the communication links among different users or conversations, the ability to send data in as few bits as possible is important.

We say that a code is *optimal* if its expected code length, $L$, is the minimum among all possible codes. The corresponding code tree gives us the optimal mapping between symbols and codewords, and is usually not unique. Shannon proved that the expected code length of any decodable code cannot be smaller than the entropy, $H$, of the underlying probability distribution over the symbols. He also showed the existence of codes that achieve entropy asymptotically, as the length of the coded messages approaches $\infty$. Thus,

[1]Somewhat unfortunately, several papers and books use the term "prefix code" to mean the same thing as a "prefix-free code". Caveat emptor.

an optimal code will have an expected code length that "matches" the entropy for long messages.

The rest of this chapter describes two optimal codes. First, Huffman codes, which are optimal instantaneous codes when the probabilities of the various symbols are given and we restrict ourselves to mapping individual symbols to codewords. It is a prefix-free, instantaneous code, satisfying the property $H \leq L \leq H + 1$. Second, the LZW algorithm, which adapts to the actual distribution of symbols in the message, not relying on any a priori knowledge of symbol probabilities.

## ■ 3.2 Huffman Codes

Huffman codes give an efficient encoding for a list of symbols to be transmitted, when we know their probabilities of occurrence in the messages to be encoded. We'll use the intuition developed in the previous chapter: more likely symbols should have shorter encodings, less likely symbols should have longer encodings.

If we draw the variable-length code of Figure 2-2 as a code tree, we'll get some insight into how the encoding algorithm should work:

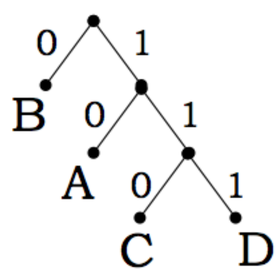


**Figure 3-1: Variable-length code from Figure 2-2 shown in the form of a code tree.**

To encode a symbol using the tree, start at the root and traverse the tree until you reach the symbol to be encoded—the encoding is the concatenation of the branch labels in the order the branches were visited. The destination node, which is always a "leaf" node for an instantaneous or prefix-free code, determines the path, and hence the encoding. So $B$ is encoded as 0, $C$ is encoded as 110, and so on. Decoding complements the process, in that now the path (codeword) determines the symbol, as described in the previous section. So 111100 is decoded as: $111 \rightarrow D$, $10 \rightarrow A$, $0 \rightarrow B$.

Looking at the tree, we see that the more probable symbols (e.g., $B$) are near the root of the tree and so have short encodings, while less-probable symbols (e.g., $C$ or $D$) are further down and so have longer encodings. David Huffman used this observation while writing a term paper for a graduate course taught by Bob Fano here at M.I.T. in 1951 to devise an algorithm for building the decoding tree for an optimal variable-length code.

Huffman's insight was to build the decoding tree *bottom up*, starting with the least probable symbols and applying a greedy strategy. Here are the steps involved, along with a worked example based on the variable-length code in Figure 2-2. The input to the algorithm is a set of symbols and their respective probabilities of occurrence. The output is the code tree, from which one can read off the codeword corresponding to each symbol.

1. **Input:** A set $S$ of tuples, each tuple consisting of a message symbol and its associated probability.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.083, C), (0.083, D)\}$

2. Remove from $S$ the two tuples with the smallest probabilities, resolving ties arbitrarily. Combine the two symbols from the removed tuples to form a new tuple (which will represent an interior node of the code tree). Compute the probability of this new tuple by adding the two probabilities from the tuples. Add this new tuple to $S$. (If $S$ had $N$ tuples to start, it now has $N - 1$, because we removed two tuples and added one.)

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.167, C \wedge D)\}$

3. Repeat step 2 until $S$ contains only a single tuple. (That last tuple represents the root of the code tree.)

   Example, iteration 2: $S \leftarrow \{(0.5, B), (0.5, A \wedge (C \wedge D))\}$
   Example, iteration 3: $S \leftarrow \{(1.0, B \wedge (A \wedge (C \wedge D)))\}$

*Et voila!* The result is a code tree representing a variable-length code for the given symbols and probabilities. As you'll see in the Exercises, the trees aren't always "tall and thin" with the left branch leading to a leaf; it's quite common for the trees to be much "bushier." As a simple example, consider input symbols $A, B, C, D, E, F, G, H$ with equal probabilities of occurrences (1/8 for each). In the first pass, one can pick any two as the two lowest-probability symbols, so let's pick $A$ and $B$ without loss of generality. The combined $AB$ symbol has probability 1/4, while the other six symbols have probability 1/8 each. In the next iteration, we can pick any two of the symbols with probability 1/8, say $C$ and $D$. Continuing this process, we see that after four iterations, we would have created four sets of combined symbols, each with probability 1/4 each. Applying the algorithm, we find that the code tree is a complete binary tree where every symbol has a codeword of length 3, corresponding to all combinations of 3-bit words (000 through 111).

Huffman codes have the biggest reduction in the expected length of the encoded message when some symbols are substantially more probable than other symbols. If all symbols are equiprobable, then all codewords are roughly the same length, and there are (nearly) fixed-length encodings whose expected code lengths approach entropy and are thus close to optimal.

## ■ 3.2.1 Properties of Huffman Codes

We state some properties of Huffman codes here. We don't prove these properties formally, but provide intuition about why they hold.

**Non-uniqueness.** In a trivial way, because the 0/1 labels on any pair of branches in a code tree can be reversed, there are in general multiple different encodings that all have the same expected length. In fact, there *may* be multiple optimal codes for a given set of symbol probabilities, and depending on how ties are broken, Huffman coding can produce different *non-isomorphic* code trees, i.e., trees that look different structurally and aren't just relabelings of a single underlying tree. For example, consider six symbols with proba-

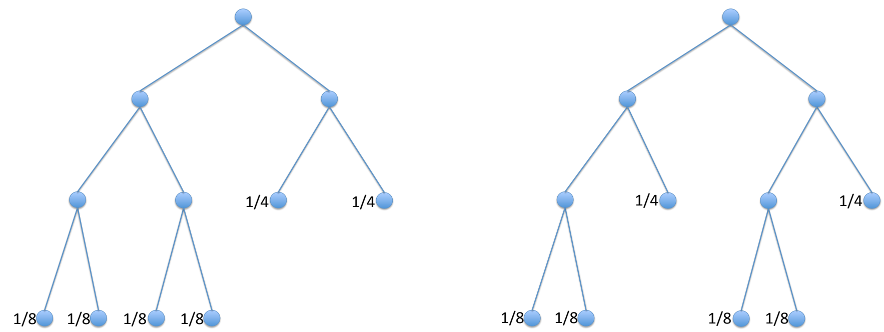


**Figure 3-2: An example of two non-isomorphic Huffman code trees, both optimal.**

bilities $1/4, 1/4, 1/8, 1/8, 1/8, 1/8$. The two code trees shown in Figure 3-2 are both valid Huffman (optimal) codes.

**Optimality.** Huffman codes are optimal in the sense that there are no other codes with shorter expected length, when restricted to instantaneous (prefix-free) codes and symbols occur independently in messages from a known probability distribution.

We state here some propositions that are useful in establishing the optimality of Huffman codes.

**Proposition 3.1** *In any optimal code tree for a prefix-free code, each node has either zero or two children.*

To see why, suppose an optimal code tree has a node with one child. If we take that node and move it up one level to its parent, we will have reduced the expected code length, and the code will remain decodable. Hence, the original tree was not optimal, a contradiction.

**Proposition 3.2** *In the code tree for a Huffman code, no node has exactly one child.*

To see why, note that we always combine the two lowest-probability nodes into a single one, which means that in the code tree, each internal node (i.e., non-leaf node) comes from two combined nodes (either internal nodes themselves, or original symbols).

**Proposition 3.3** *There exists an optimal code in which the two least-probable symbols:*

- *have the longest length, and*
- *are siblings, i.e., their codewords differ in exactly the one bit (the last one).*

*Proof.* Let $z$ be the least-probable symbol. If it is not at maximum depth in the optimal code tree, then some other symbol, call it $s$, must be at maximum depth. But because $p_z < p_s$, if we swapped $z$ and $s$ in the code tree, we would end up with a code with smaller expected length. Hence, $z$ must have a codeword at least as long as every other codeword.

Now, symbol $z$ must have a sibling in the optimal code tree, by Proposition 3.1. Call it $x$. Let $y$ be the symbol with second lowest probability; i.e., $p_x \geq p_y \geq p_x$. If $p_x = p_y$, then the proposition is proved. Let's swap $x$ and $y$ in the code tree, so now $y$ is a sibling of $z$. The expected code length of this code tree is not larger than the pre-swap optimal code tree, because $p_x$ is strictly greater than $p_y$, proving the proposition. ■

**Theorem 3.1** *Huffman coding over a set of symbols with known probabilities produces a code tree whose expected length is optimal.*

*Proof.* Proof by induction on $n$, the number of symbols. Let the symbols be $x_1, x_2, \ldots, x_{n-1}, x_n$ and let their respective probabilities of occurrence be $p_1 \geq p_2 \geq \ldots \geq p_{n-1} \geq p_n$. From Proposition 3.3, there exists an optimal code tree in which $x_{n-1}$ and $x_n$ have the longest length and are siblings.

Inductive hypothesis: Assume that Huffman coding produces an optimal code tree on an input with $n-1$ symbols with associated probabilities of occurrence. The base case is trivial to verify.

Let $H_n$ be the expected cost of the code tree generated by Huffman coding on the $n$ symbols $x_1, x_2, \ldots, x_n$. Then, $H_n = H_{n-1} + p_{n-1} + p_n$, where $H_{n-1}$ is the expected cost of the code tree generated by Huffman coding on $n-1$ input symbols $x_1, x_2, \ldots x_{n-2}, x_{n-1,n}$ with probabilities $p_1, p_2, \ldots, p_{n-2}, (p_{n-1} + p_n)$.

By the inductive hypothesis, $H_{n-1} = L_{n-1}$, the expected cost of the optimal code tree over $n-1$ symbols. Moreover, from Proposition 3.3, there exists an optimal code tree over $n$ symbols for which $L_n = L_{n-1} + (p_{n-1} + p_n)$. Hence, there exists an optimal code tree whose expected cost, $L_n$, is equal to the expected cost, $H_n$, of the Huffman code over the $n$ symbols. ■

**Huffman coding with grouped symbols.** The entropy of the distribution shown in Figure 2-2 is 1.626. The per-symbol encoding of those symbols using Huffman coding produces a code with expected length 1.667, which is noticeably larger (e.g., if we were to encode 10,000 grades, the difference would be about 410 bits). Can we apply Huffman coding to get closer to entropy?

One approach is to *group* symbols into larger "metasymbols" and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

Consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example from Figure 2-2:

| Size of grouping | Number of leaves in tree | Expected length for 1000 grades |
|---|---|---|
| 1 | 4 | 1667 |
| 2 | 16 | 1646 |
| 3 | 64 | 1637 |
| 4 | 256 | 1633 |

**Figure 3-3: Results from encoding more than one grade at a time.**

We see that we can come closer to the Shannon lower bound (i.e., entropy) of 1.626 bits by encoding grades in larger groups at a time, but at a cost of a more complex encoding and decoding process. This approach still has two problems: first, it requires knowledge of the individual symbol probabilities, and second, it assumes that the probability of each symbol is independent and identically distributed. In practice, however, symbol probabilities change message-to-message, or even within a single message.

This last observation suggests that it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. The LZW algorithm, presented in the next section, is such a method.

## ■ 3.3 LZW: An Adaptive Variable-length Source Code

Let's first understand the compression problem better by considering the problem of digitally representing and transmitting the text of a book written in, say, English. A simple approach is to analyze a few books and estimate the probabilities of different letters of the alphabet. Then, treat each letter as a symbol and apply Huffman coding to compress a document.

This approach is reasonable but ends up achieving relatively small gains compared to the best one can do. One big reason why is that the probability with which a letter appears in any text is not always the same. For example, a priori, "x" is one of the least frequently appearing letters, appearing only about 0.3% of the time in English text. But if in the sentence "... nothing can be said to be certain, except death and ta___", the next letter is almost certainly an "x". In this context, no other letter can be more certain!

Another reason why we might expect to do better than Huffman coding is that it is often unclear what the best symbols might be. For English text, because individual letters vary in probability by context, we might be tempted to try out words. It turns out that word occurrences also change in probability depend on context.

An approach that *adapts* to the material being compressed might avoid these shortcomings. One approach to adaptive encoding is to use a two pass process: in the first pass, count how often each symbol (or pairs of symbols, or triples—whatever level of grouping you've chosen) appears and use those counts to develop a Huffman code customized to the contents of the file. Then, in the second pass, encode the file using the customized Huffman code. This strategy is expensive but workable, yet it falls short in several ways. Whatever size symbol grouping is chosen, it won't do an optimal job on encoding recurring groups of some different size, either larger or smaller. And if the symbol probabilities change dramatically at some point in the file, a one-size-fits-all Huffman code won't be optimal; in this case one would want to change the encoding midstream.

A different approach to adaptation is taken by the popular **Lempel-Ziv-Welch (LZW)** algorithm. This method was developed originally by Ziv and Lempel, and subsequently improved by Welch. As the message to be encoded is processed, the LZW algorithm builds a *string table* that maps symbol sequences to/from an $N$-bit index. The string table has $2^N$ entries and the transmitted code can be used at the decoder as an index into the string table to retrieve the corresponding original symbol sequence. The sequences stored in the table can be arbitrarily long. The algorithm is designed so that the string table can be reconstructed by the decoder based on information in the encoded stream—the table, while central to the encoding and decoding process, is never transmitted! This property is crucial to the understanding of the LZW method.

```
initialize TABLE[0 to 255] = code for individual bytes
STRING = get input symbol
while there are still input symbols:
   SYMBOL = get input symbol
   if STRING + SYMBOL is in TABLE:
       STRING = STRING + SYMBOL
   else:
       output the code for STRING
       add STRING + SYMBOL to TABLE
       STRING = SYMBOL
output the code for STRING
```

**Figure 3-4: Pseudo-code for the LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity.**

```
initialize TABLE[0 to 255] = code for individual bytes
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING

while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined: // needed because sometimes the
        ENTRY = STRING + STRING[0] // decoder may not yet have entry!
    else:
        ENTRY = TABLE[CODE]
    output ENTRY
    add STRING+ENTRY[0] to TABLE
    STRING = ENTRY
```

**Figure 3-5: Pseudo-code for LZW adaptive variable-length decoder.**

When encoding a byte stream,[2] the first $2^8 = 256$ entries of the string table, numbered 0 through 255, are initialized to hold all the possible one-byte sequences. The other entries will be filled in as the message byte stream is processed. The encoding strategy works as follows and is shown in pseudo-code form in Figure 3-4. First, accumulate message bytes as long as the accumulated sequences appear as some entry in the string table. At some point, appending the next byte $b$ to the accumulated sequence $S$ would create a sequence $S + b$ that's not in the string table, where $+$ denotes appending $b$ to $S$. The encoder then executes the following steps:

1. It transmits the $N$-bit code for the sequence $S$.
2. It adds a new entry to the string table for $S + b$. If the encoder finds the table full when it goes to add an entry, it reinitializes the table before the addition is made.
3. it resets $S$ to contain only the byte $b$.

[2] A byte is a contiguous string of 8 bits.

| $S$ | msg. byte | lookup | result | transmit | string table |
|---|---|---|---|---|---|
| – | a | – | – | – | – |
| a | b | ab | not found | index of a | table[256] = ab |
| b | c | bc | not found | index of b | table[257] = bc |
| c | a | ca | not found | index of c | table[258] = ca |
| a | b | ab | found | – | – |
| ab | c | abc | not found | 256 | table[259] = abc |
| c | a | ca | found | – | – |
| ca | b | cab | not found | 258 | table[260] = cab |
| b | c | bc | found | – | – |
| bc | a | bca | not found | 257 | table[261] = bca |
| a | b | ab | found | – | – |
| ab | c | abc | found | – | – |
| abc | a | abca | not found | 259 | table[262] = abca |
| a | b | ab | found | – | – |
| ab | c | abc | found | – | – |
| abc | a | abca | found | – | – |
| abca | b | abcab | not found | 262 | table[263] = abcab |
| b | c | bc | found | – | – |
| bc | a | bca | found | – | – |
| bca | b | bcab | not found | 261 | table[264] = bcab |
| b | c | bc | found | – | – |
| bc | a | bca | found | – | – |
| bca | b | bcab | found | – | – |
| bcab | c | bcabc | not found | 264 | table[265] = bcabc |
| c | a | ca | found | – | – |
| ca | b | cab | found | – | – |
| cab | c | cabc | not found | 260 | table[266] = cabc |
| c | a | ca | found | – | – |
| ca | b | cab | found | – | – |
| cab | c | cabc | found | – | – |
| cabc | a | cabca | not found | 266 | table[267] = cabca |
| a | b | ab | found | – | – |
| ab | c | abc | found | – | – |
| abc | a | abca | found | – | – |
| abca | b | abcab | found | – | – |
| abcab | c | abcabc | not found | 263 | table[268] = abcabc |
| c | – end – | – | – | index of c | – |

**Figure 3-6: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabcabc"**

| received | string table | decoding |
|---|---|---|
| a | – | a |
| b | table[256] = ab | b |
| c | table[257] = bc | c |
| 256 | table[258] = ca | ab |
| 258 | table[259] = abc | ca |
| 257 | table[260] = cab | bc |
| 259 | table[261] = bca | abc |
| 262 | table[262] = abca | abca |
| 261 | table[263] = abcab | bca |
| 264 | table[264] = bacb | bcab |
| 260 | table[265] = bcabc | cab |
| 266 | table[266] = cabc | cabc |
| 263 | table[267] = cabca | abcab |
| c | table[268] = abcabc | c |

**Figure 3-7: LZW decoding of the sequence** $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

This process repeats until all the message bytes are consumed, at which point the encoder makes a final transmission of the $N$-bit code for the current sequence $S$.

Note that for every transmission done by the encoder, the encoder makes a new entry in the string table. With a little cleverness, the decoder, shown in pseudo-code form in Figure 3-5, can figure out what the new entry must have been as it receives each N-bit code. With a *duplicate* string table at the decoder constructed as the algorithm progresses at the decoder, it is possible to recover the original message: just use the received $N$-bit code as index into the decoder's string table to retrieve the original sequence of message bytes.

Figure 3-6 shows the encoder in action on a repeating sequence of *abc*. Notice that:

- The encoder algorithm is greedy—it is designed to find the longest possible match in the string table before it makes a transmission.
- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.
- Since the encoder operates without any knowledge of what's to come in the message stream, there may be entries in the string table that don't correspond to a sequence that's repeated, i.e., some of the possible $N$-bit codes will never be transmitted. This property means that the encoding isn't optimal—a prescient encoder could do a better job.
- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.
- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

Figure 3-7 shows the operation of the decoder on the transmit sequence produced in Figure 3-6. As each $N$-bit code is received, the decoder deduces the correct entry to make in the string table (i.e., the same entry as made at the encoder) and then uses the $N$-bit code as index into the table to retrieve the original message sequence.

There is a special case, which turns out to be important, that needs to be dealt with. There are three instances in Figure 3-7 where the decoder receives an index $(262, 264, 266)$ that it has not previously entered in *its* string table. So how does it figure out what these correspond to? A careful analysis, which you could do, shows that this situation only happens when the associated string table entry has its last symbol identical to its first symbol. To handle this issue, the decoder can simply complete the partial string that it is building up into a table entry (abc, bac, cab respectively, in the three instances in Figure 3-7) by repeating its first symbol at the end of the string (to get abca, bacb, cabc respectively, in our example), and then entering this into the string table. This step is captured in the pseudo-code in Figure 3-5 by the logic of the "if" statement there.

We conclude this chapter with some interesting observations about LZW compression:

- A common choice for the size of the string table is 4096 ($N = 12$). A larger table means the encoder has a longer memory for sequences it has seen and increases the possibility of discovering repeated sequences across longer spans of message. However, dedicating string table entries to remembering sequences that will never be seen again decreases the efficiency of the encoding.

- Early in the encoding, the encoder uses entries near the beginning of the string table, i.e., the high-order bits of the string table index will be 0 until the string table starts to fill. So the $N$-bit codes we transmit at the outset will be numerically small. Some variants of LZW transmit a variable-width code, where the width grows as the table fills. If $N = 12$, the initial transmissions may be only 9 bits until entry number 511 in the table is filled (i.e., 512 entries filled in all), then the code expands to 10 bits, and so on, until the maximum width $N$ is reached.

- Some variants of LZW introduce additional special transmit codes, e.g., CLEAR to indicate when the table is reinitialized. This allows the encoder to reset the table pre-emptively if the message stream probabilities change dramatically, causing an observable drop in compression efficiency.

- There are many small details we haven't discussed. For example, when sending $N$-bit codes one bit at a time over a serial communication channel, we have to specify the order in the which the $N$ bits are sent: least significant bit first, or most significant bit first. To specify $N$, serialization order, algorithm version, etc., most compressed file formats have a header where the encoder can communicate these details to the decoder.

## ■ Exercises

1. Huffman coding is used to compactly encode the species of fish tagged by a game warden. If 50% of the fish are bass and the rest are evenly divided among 15 other species, how many bits would be used to encode the species when a bass is tagged?

2. Consider a Huffman code over four symbols, $A$, $B$, $C$, and $D$. Which of these is a valid Huffman encoding? Give a brief explanation for your decisions.

   (a) $A:0, B:11, C:101, D:100$.

   (b) $A:1, B:01, C:00, D:010$.

   (c) $A:00, B:01, C:110, D:111$

3. Huffman is given four symbols, $A$, $B$, $C$, and $D$. The probability of symbol $A$ occurring is $p_A$, symbol $B$ is $p_B$, symbol $C$ is $p_C$, and symbol $D$ is $p_D$, with $p_A \geq p_B \geq p_C \geq p_D$. Write down a single condition (equation or inequality) that is both necessary and sufficient to guarantee that, when Huffman constructs the code bearing his name over these symbols, each symbol will be encoded using exactly two bits. Explain your answer.

4. Describe the contents of the string table created when encoding a very long string of all $a$'s using the simple version of the LZW encoder shown in Figure 3-4. In this example, if the decoder has received $E$ encoded symbols (i.e., string table indices) from the encoder, how many $a$'s has it been able to decode?

5. Consider the pseudo-code for the LZW decoder given in Figure 3-4. Suppose that this decoder has received the following five codes from the LZW encoder (these are the first five codes from a longer compression run):

```
97 -- index of 'a' in the translation table
98 -- index of 'b' in the translation table
257 -- index of second addition to the translation table
256 -- index of first addition to the translation table
258 -- index of third addition to in the translation table
```

   After it has finished processing the fifth code, what are the entries in the translation table and what is the cumulative output of the decoder?

   **table[256]:** ___________

   **table[257]:** ___________

   **table[258]:** ___________

   **table[259]:** ___________

   **cumulative output from decoder:** _________________

6. Consider the LZW compression and decompression algorithms as described in this chapter. Assume that the scheme has an initial table with code words 0 through 255 corresponding to the 8-bit ASCII characters; character "a" is 97 and "b" is 98. The receiver gets the following sequence of code words, each of which is 10 bits long:

   97 97 98 98 257 256

   (a) What was the original message sent by the sender?

(b) By how many bits is the compressed message shorter than the original message (each character in the original message is 8 bits long)?

(c) What is the first string of length 3 added to the compression table? (If there's no such string, your answer should be "None".)