*MIT 6.02* **DRAFT** *Lecture Notes*
*Last update: February 8, 2012*
*Comments, questions or bug reports?*
*Please contact hari at mit.edu*

# CHAPTER 2
# Information, Entropy, and the Motivation for Source Codes

The theory of *information* developed by Claude Shannon (SM EE '40, PhD Math '40) in the late 1940s is one of the most impactful ideas of the past century, and has changed the theory and practice of many fields of technology. The development of communication systems and networks has benefited greatly from Shannon's work. In this chapter, we will first develop the intution behind *information* and formally define it as a mathematical quantity and connect it to another property of data sources, *entropy*.

We will then show how these notions guide us to efficiently *compress* and *decompress* a data source before communicating (or storing) it without distorting the quality of information being received. A key underlying idea here is *coding*, or more precisely, *source coding*, which takes each message (or "symbol") being produced by any source of data and associate each message (symbol) with a *codeword*, while achieving several desirable properties. This mapping between input messages (symbols) and codewords is called a **code**. Our focus will be on *lossless* compression (source coding) techniques, where the recipient of any uncorrupted message can recover the original message exactly (we deal with corrupted bits later in later chapters).

## ■ 2.1 Information and Entropy

One of Shannon's brilliant insights, building on earlier work by Hartley, was to realize that *regardless of the application and the semantics of the messages involved*, a general definition of information is possible. When one abstracts away the details of an application, the task of communicating something between two parties, $S$ and $R$, boils down to $S$ picking one of several (possibly infinite) messages and sending that message to $R$. Let's take the simplest example, when a sender wishes to send one of two messages—for concreteness, let's say that the message is to say which way the British are coming:

- "1" if by land.
- "2" if by sea.

(Had the sender been from Course VI, it would've almost certainly been "0" if by land and "1" if by sea!)

Let's say we have no prior knowledge of how the British might come, so each of these choices (messages) is equally probable. In this case, the amount of information conveyed by the sender specifying the choice is **1 bit**. Intuitively, that bit, which can take on one of two values, can be used to *encode* the particular choice. If we have to communicate a sequence of such independent events, say 1000 such events, we can encode the outcome using 1000 bits of information, each of which specifies the outcome of an associated event.

On the other hand, suppose we somehow knew that the British were far more likely to come by land than by sea (say, because there is a severe storm forecast). Then, if the message in fact says that the British are coming by sea, much more information is being conveyed than if the message said that that they were coming by land. To take another example, far more information is conveyed by my telling you that the temperature in Boston on a January day is 75°F, than if I told you that the temperature is 32°F!

The conclusion you should draw from these examples is that any quantification of "information" about an event should depend on the *probability* of the event. The greater the probability of an event, the smaller the information associated with knowing that the event has occurred.

### ■ 2.1.1 Information definition

Using such intuition, Hartley proposed the following definition of the information associated with an event whose probability of occurrence is $p$:

$$I \equiv \log(1/p) = -\log(p). \tag{2.1}$$

This definition satisfies the basic requirement that it is a decreasing function of $p$. But so do an infinite number of functions, so what is the intuition behind using the logarithm to define information? And what is the base of the logarithm?

The second question is easy to address: you can use any base, because $\log_a(1/p) = \log_b(1/p)/\log_b a$, for any two bases $a$ and $b$. Following Shannon's convention, *we will use base 2*,[1] in which case the unit of information is called a **bit**.[2]

The answer to the first question, why the logarithmic function, is that the resulting definition has several elegant resulting properties, and it is the simplest function that provides these properties. One of these properties is **additivity**. If you have two independent events (i.e., events that have nothing to do with each other), then the probability that they both occur is equal to the *product* of the probabilities with which they each occur. What we would like is for the corresponding information to *add up*. For instance, the event that it rained in Seattle yesterday and the event that the number of students enrolled in 6.02 exceeds 150 are independent, and if I am told something about both events, the amount of information I now have should be the sum of the information in being told individually of the occurrence of the two events.

The logarithmic definition provides us with the desired additivity because, given two

[1] And we won't mention the base; if you see a log in this chapter, it will be to base 2 unless we mention otherwise.

[2] If we were to use base 10, the unit would be *Hartleys*, and if we were to use the natural log, base $e$, it would be *nats*, but no one uses those units in practice.

independent events $A$ and $B$ with probabilities $p_A$ and $p_B$,

$$I_A + I_B = \log(1/p_A) + \log(1/p_B) = \log \frac{1}{p_A p_B} = \log \frac{1}{P(A \text{ and } B)}.$$

### ■ 2.1.2 Examples

Suppose that we're faced with $N$ equally probable choices. What is the information received when I tell you which of the $N$ choices occurred?

Because the probability of each choice is $1/N$, the information is $\log(1/(1/N)) = \log N$ bits.

Now suppose there are initially $N$ equally probable choices, and I tell you something that narrows the possibilities down to one of $M$ equally probable choices. How much information have I given you about the choice?

We can answer this question by observing that you now know that the probability of the choice narrowing done from $N$ equi-probable possibilities to $M$ equi-probable ones is $M/N$. Hence, the information you have received is $\log(1/(M/N)) = \log(N/M)$ bits. (Note that when $M = 1$, we get the expected answer of $\log N$ bits.)

We can therefore write a convenient rule:

> Suppose we have received information that narrows down a set of $N$ equi-probable choices to one of $M$ equi-probable choices. Then, we have received $\log(N/M)$ bits of information.

Some examples may help crystallize this concept:

**One flip of a fair coin**

Before the flip, there are two equally probable choices: heads or tails. After the flip, we've narrowed it down to one choice. Amount of information = $\log_2(2/1) = 1$ bit.

**Roll of two dice**

Each die has six faces, so in the roll of two dice there are 36 possible combinations for the outcome. Amount of information = $\log_2(36/1) = 5.2$ bits.

**Learning that a randomly chosen decimal digit is even**

There are ten decimal digits; five of them are even (0, 2, 4, 6, 8). Amount of information = $\log_2(10/5) = 1$ bit.

**Learning that a randomly chosen decimal digit $\geq 5$**

Five of the ten decimal digits are greater than or equal to 5. Amount of information = $\log_2(10/5) = 1$ bit.

**Learning that a randomly chosen decimal digit is a multiple of 3**

Four of the ten decimal digits are multiples of 3 (0, 3, 6, 9). Amount of information = $\log_2(10/4) = 1.322$ bits.

**Learning that a randomly chosen decimal digit is even, $\geq 5$, and a multiple of 3**

Only one of the decimal digits, 6, meets all three criteria. Amount of information =

$\log_2(10/1) = 3.322$ bits. Note that this information is same as the sum of the previous three examples: information is cumulative if the joint probability of the events revealed to us *factors* intothe product of the individual probabilities.

In this example, we can calculate the probability that they all occur together, and compare that answer with the product of the probabilities of each of them occurring individually. Let event $A$ be "the digit is even", event $B$ be "the digit is $\geq 5$, and event $C$ be "the digit is a multiple of 3". Then, $P(A \text{ and } B \text{ and } C) = 1/10$ because there is only one digit, 6, that satisfies all three conditions. $P(A) \cdot P(B) \cdot P(C) = 1/2 \cdot 1/2 \cdot 4/10 = 1/10$ as well. The reason information adds up is that $\log(1/P(AandBandC) = \log 1/P(A) + \log 1/P(B) + \log(1/P(C)$.

Note that *pairwise indepedence* between events is actually not necessary for information from three (or more) events to add up. In this example, $P(AandB) = P(A) \cdot P(B|A) = 1/2 \cdot 2/5 = 1/5$, while $P(A) \cdot P(B) = 1/2 \cdot 1/2 = 1/4$.

**Learning that a randomly chosen decimal digit is a prime**

Four of the ten decimal digits are primes—2, 3, 5, and 7. Amount of information = $\log_2(10/4) = 1.322$ bits.

**Learning that a randomly chosen decimal digit is even and prime**

Only one of the decimal digits, 2, meets both criteria. Amount of information = $\log_2(10/1) = 3.322$ bits. Note that this quantity is *not* the same as the sum of the information contained in knowing that the digit is even and the digit is prime. The reason is that those events are not independent: the probability that a digit is even *and* prime is $1/10$, and is *not* the product of the probabilities of the two events (i.e., not equal to $1/2 \times 4/10$).

To summarize: more information is received when learning of the occurrence of an unlikely event (small $p$) than learning of the occurrence of a more likely event (large $p$). The information learned from the occurrence of an event of probability $p$ is defined to be $\log(1/p)$.

### ■ 2.1.3 Entropy

Now that we know how to measure the information contained in a given event, we can quantify the *expected information* in a set of possible outcomes. Specifically, if an event $i$ occurs with probability $p_i, 1 \leq i \leq N$ out of a set of $N$ events, then the average or expected information is given by

$$H(p_1, p_2, \ldots p_N) = \sum_{i=1}^{N} p_i \log(1/p_i). \tag{2.2}$$

$H$ is also called the **entropy** (or *Shannon entropy*) of the probability distribution. Like information, it is also measured in bits. It is simply the sum of several terms, each of which is the information of a given event weighted by the probability of that event occurring. It is often useful to think of the entropy as *the average or expected* **uncertainty** *associated with this set of events*.

In the important special case of two *mutually exclusive* events (i.e., exactly one of the two

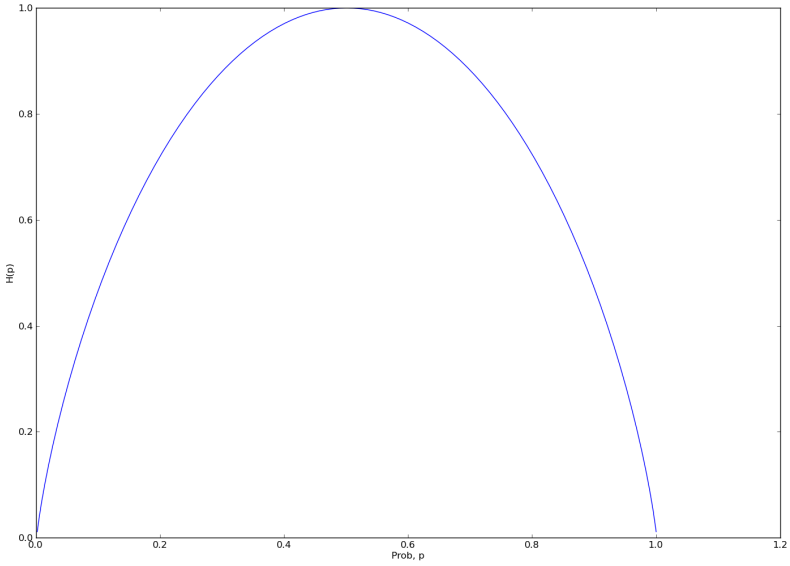


**Figure 2-1:** $H(p)$ **as a function of** $p$**, maximum when** $p = 1/2$**.**

events can occur), occuring with probabilities $p$ and $1 - p$, respectively, the entropy

$$H(p, 1-p) = -p \log p - (1-p) \log(1-p). \tag{2.3}$$

We will be lazy and refer to this special case, $H(p, 1-p)$ as simply $H(p)$.

This entropy as a function of $p$ is plotted in Figure 2-1. It is symmetric about $p = 1/2$, with its maximum value of 1 bit occuring when $p = 1/2$. Note that $H(0) = H(1) = 0$; although $\log(1/p) \to \infty$ as $p \to 0, \lim_{p \to 0} p \log(1/p) \to 0$.

It is easy to verify that the expression for $H$ from Equation (2.2) is always non-negative. Moreover, $H(p_1, p_2, \ldots p_N) \leq \log N$ always.

## ■ 2.2 Source Codes

We now turn to the problem of *source coding*, i.e., taking a set of messages that need to be sent from a sender and *encoding* them in a way that is efficient. The notions of information and entropy will be fundamentally important in this effort.

Many messages have an obvious encoding, e.g., an ASCII text file consists of sequence of individual characters, each of which is independently encoded as a separate byte. There are other such encodings: images as a raster of color pixels (e.g., 8 bits each of red, green and blue intensity), sounds as a sequence of samples of the time-domain audio waveform, etc. What makes these encodings so popular is that they are produced and consumed by our computer's peripherals—characters typed on the keyboard, pixels received from a digital camera or sent to a display, and digitized sound samples output to the computer's audio chip.

All these encodings involve a sequence of fixed-length symbols, each of which can be

easily manipulated independently. For example, to find the $42^{nd}$ character in the file, one just looks at the $42^{nd}$ byte and interprets those 8 bits as an ASCII character. A text file containing 1000 characters takes 8000 bits to store. If the text file were HTML to be sent over the network in response to an HTTP request, it would be natural to send the 1000 bytes (8000 bits) exactly as they appear in the file.

But let's think about how we might compress the file and send fewer than 8000 bits. If the file contained English text, we'd expect that the letter *e* would occur more frequently than, say, the letter *x*. This observation suggests that if we encoded *e* for transmission using *fewer* than 8 bits—and, as a trade-off, had to encode less common characters, like *x*, using more than 8 bits—we'd expect the encoded message to be shorter *on average* than the original method. So, for example, we might choose the bit sequence 00 to represent *e* and the code 100111100 to represent *x*.

This intuition is consistent with the definition of the amount of information: commonly occurring symbols have a higher $p_i$ and thus convey less information, so we need fewer bits to encode such symbols. Similarly, infrequently occurring symbols like *x* have a lower $p_i$ and thus convey more information, so we'll use more bits when encoding such symbols. This intuition helps meet our goal of matching the size of the transmitted data to the information content of the message.

The mapping of information we wish to transmit or store into bit sequences is referred to as a **code**. Two examples of codes (fixed-length and variable-length) are shown in Figure 2-2, mapping different grades to bit sequences in one-to-one fashion. The fixed-length code is straightforward, but the variable-length code is not arbitrary, but has been carefully designed, as we will soon learn. Each bit sequence in the code is called a **codeword**.

When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data (ideally, to match the expected number of bits to the underlying entropy), the resulting mapping is called a **source code**. Source codes are distinct from **channel codes** we will study in later chapters. Source codes *remove redundancy* and compress the data, while channel codes *add redundancy* in a controlled way to improve the error resilience of the data in the face of bit errors and erasures caused by imperfect communication channels. This chapter and the next are about source codes.

We can generalize this insight about encoding common symbols (such as the letter *e*) more succinctly than uncommon symbols into a strategy for *variable-length codes*:

> Send commonly occurring symbols using shorter codewords (fewer bits) and infrequently occurring symbols using longer codewords (more bits).

We'd expect that, on average, encoding the message with a variable-length code would take fewer bits than the original fixed-length encoding. Of course, if the message were all *x*'s the variable-length encoding would be longer, but our encoding scheme is designed to optimize the expected case, not the worst case.

Here's a simple example: suppose we had to design a system to send messages containing 1000 6.02 grades of *A*, *B*, *C* and *D* (MIT students rarely, if ever, get an F in 6.02 $\ddot\smile$). Examining past messages, we find that each of the four grades occurs with the probabilities shown in Figure 2-2.

With four possible choices for each grade, if we use the fixed-length encoding, we need 2 bits to encode a grade, for a total transmission length of 2000 bits when sending 1000 grades.

| Grade | Probability | Fixed-length Code | Variable-length Code |
|---|---|---|---|
| $A$ | $1/3$ | 00 | 10 |
| $B$ | $1/2$ | 01 | 0 |
| $C$ | $1/12$ | 10 | 110 |
| $D$ | $1/12$ | 11 | 111 |

**Figure 2-2: Possible grades shown with probabilities, fixed- and variable-length encodings**

Fixed-length encoding for $BCBAAB$: 01 10 01 00 00 01 (12 bits)

With a fixed-length code, the size of the transmission doesn't depend on the actual message—sending 1000 grades always takes exactly 2000 bits.

Decoding a message sent with the fixed-length code is straightforward: take each pair of received bits and look them up in the table above to determine the corresponding grade. Note that it's possible to determine, say, the $42^{nd}$ grade without decoding any other of the grades—just look at the $42^{nd}$ pair of bits.

Using the variable-length code, the number of bits needed for transmitting 1000 grades depends on the grades.

Variable-length encoding for $BCBAAB$: 0 110 0 10 10 0 (10 bits)

If the grades were all $B$, the transmission would take only 1000 bits; if they were all $C$'s and $D$'s, the transmission would take 3000 bits. But we can use the grade probabilities given in Figure 2-2 to compute the *expected length of a transmission* as

$$1000[(\frac{1}{3})(2) + (\frac{1}{2})(1) + (\frac{1}{12})(3) + (\frac{1}{12})(3)] = 1000[1\frac{2}{3}] = 1666.7 \text{ bits}$$

So, on average, using the variable-length code would shorten the transmission of 1000 grades by 333 bits, a savings of about 17%. Note that to determine, say, the $42^{nd}$ grade, we would need to decode the first 41 grades to determine where in the encoded message the $42^{nd}$ grade appears.

Using variable-length codes looks like a good approach if we want to send fewer bits on average, but preserve all the information in the original message. On the downside, we give up the ability to access an arbitrary message symbol without first decoding the message up to that point.

One obvious question to ask about a particular variable-length code: is it the best encoding possible? Might there be a different variable-length code that could do a better job, i.e., produce even shorter messages on the average? How short can the messages be on the average? We turn to this question next.

## ■ 2.3 How Much Compression Is Possible?

Ideally we'd like to design our compression algorithm to produce as few bits as possible: just enough bits to represent the information in the message, but no more. Ideally, we will be able to use no more bits than the amount of information, as defined in Section 2.1, contained in the message, at least on average.

Specifically, the entropy, defined by Equation (2.2), tells us the expected amount of information in a message, when the message is drawn from a set of possible messages, each occurring with some probability. The entropy is a lower bound on the amount of information that must be sent, on average, when transmitting data about a particular choice.

What happens if we violate this lower bound, i.e., we send fewer bits on the average than called for by Equation (2.2)? In this case the receiver will not have sufficient information and there will be some remaining ambiguity—exactly what ambiguity depends on the encoding, but to construct a code of fewer than the required number of bits, some of the choices must have been mapped into the same encoding. Thus, when the recipient receives one of the overloaded encodings, it will not have enough information to unambiguously determine which of the choices actually occurred.

Equation (2.2) answers our question about how much compression is possible by giving us a lower bound on the number of bits that must be sent to resolve all ambiguities at the recipient. Reprising the example from Figure 2-2, we can update the figure using Equation (2.1).

| Grade | $p_i$ | $log_2(1/p_i)$ |
|---|---|---|
| $A$ | $1/3$ | 1.58 bits |
| $B$ | $1/2$ | 1 bit |
| $C$ | $1/12$ | 3.58 bits |
| $D$ | $1/12$ | 3.58 bits |

**Figure 2-3: Possible grades shown with probabilities and information content.**

Using equation (2.2) we can compute the information content when learning of a particular grade:

$$\sum_{i=1}^{N} p_i \log_2(\frac{1}{p_i}) = (\frac{1}{3})(1.58) + (\frac{1}{2})(1) + (\frac{1}{12})(3.58) + (\frac{1}{12})(3.58) = 1.626 \text{ bits}$$

So encoding a sequence of 1000 grades requires transmitting 1626 bits on the average. The variable-length code given in Figure 2-2 encodes 1000 grades using 1667 bits on the average, and so doesn't achieve the maximum possible compression. It turns out the example code does as well as possible when encoding one grade at a time. To get closer to the lower bound, we would need to encode sequences of grades—more on this idea below.

Finding a "good" code—one where the length of the encoded message matches the information content (i.e., the entropy)—is challenging and one often has to think "outside the box". For example, consider transmitting the results of 1000 flips of an unfair coin where probability of heads is given by $p_H$. The information content in an unfair coin flip can be computed using equation (2.3):

$$p_H \log_2(1/p_H) + (1 - p_H) \log_2(1/(1 - p_H))$$

For $p_H = 0.999$, this entropy evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on average, just 11.4 bits? The recipient of the encoded message must be able to tell for each of the 1000 flips which were heads and which were tails. Hint:

with a budget of just 11 bits, one obviously can't encode each flip separately!

In fact, some effective codes leverage the context in which the encoded message is being sent. For example, if the recipient is expecting to receive a Shakespeare sonnet, then it's possible to encode the message using just 8 bits if one knows that there are only 154 Shakespeare sonnets. That is, if the sender and receiver both know the sonnets, and the sender just wishes to tell the receiver which sonnet to read or listen to, he can do that using a very small number of bits, just $\log 154$ bits if all the sonnets are equi-probable!

## 2.4 Why Compression?

There are several reasons for using compression:

- Shorter messages take less time to transmit and so the complete message arrives more quickly at the recipient. This is good for both the sender and recipient since it frees up their network capacity for other purposes and reduces their network charges. For high-volume senders of data (such as Google, say), the impact of sending half as many bytes is economically significant.
- Using network resources sparingly is good for *all* the users who must share the internal resources (packet queues and links) of the network. Fewer resources per message means more messages can be accommodated within the network's resource constraints.
- Over error-prone links with non-negligible bit error rates, compressing messages before they are channel-coded using error-correcting codes can help improve throughput because all the redundancy in the message can be designed in to improve error resilience, after removing any other redundancies in the original message. It is better to design in redundancy with the explicit goal of correcting bit errors, rather than rely on whatever sub-optimal redundancies happen to exist in the original message.

Compression is traditionally thought of as an *end-to-end function*, applied as part of the application-layer protocol. For instance, one might use lossless compression between a web server and browser to reduce the number of bits sent when transferring a collection of web pages. As another example, one might use a compressed image format such as JPEG to transmit images, or a format like MPEG to transmit video. However, one may also apply compression at the link layer to reduce the number of transmitted bits and eliminate redundant bits (before possibly applying an error-correcting code over the link). When applied at the link layer, compression only makes sense if the data is inherently compressible, which means it cannot already be compressed and must have enough redundancy to extract compression gains.

The next chapter describes two compression (source coding) schemes: Huffman Codes and Lempel-Ziv-Welch (LZW) compression.

## Exercises

1. Several people at a party are trying to guess a 3-bit binary number. Alice is told that the number is odd; Bob is told that it is not a multiple of 3 (i.e., not 0, 3, or 6); Charlie

is told that the number contains exactly two 1's; and Deb is given all three of these clues. How much information (in bits) did each player get about the number?

2. After careful data collection, Alyssa P. Hacker observes that the probability of "HIGH" or "LOW" traffic on Storrow Drive is given by the following table:

| | HIGH traffic level | LOW traffic level |
|---|---|---|
| *If the Red Sox are playing* | $P$(HIGH traffic) = 0.999 | $P$(LOW traffic) = 0.001 |
| *If the Red Sox are not playing* | $P$(HIGH traffic) = 0.25 | $P$(LOW traffic) = 0.75 |

   (a) If it is known that the Red Sox are playing, then how much information in bits is conveyed by the statement that the traffic level is LOW. Give your answer as a mathematical expression.

   (b) Suppose it is known that the Red Sox are **not** playing. What is the entropy of the corresponding probability distribution of traffic? Give your answer as a mathematical expression.

3. $X$ is an unknown 4-bit binary number picked uniformly at random from the set of all possible 4-bit numbers. You are given another 4-bit binary number, $Y$, and told that the Hamming distance between $X$ (the unknown number) and $Y$ (the number you know) is *two*. How many bits of information about $X$ have you been given?

4. In Blackjack the dealer starts by dealing 2 cards each to himself and his opponent: one face down, one face up. After you look at your face-down card, you know a total of three cards. Assuming this was the first hand played from a new deck, how many bits of information do you have about the dealer's face down card after having seen three cards?

5. The following table shows the undergraduate and MEng enrollments for the School of Engineering.

| Course (Department) | # of students | % of total |
|---|---|---|
| I (Civil & Env.) | 121 | 7% |
| II (Mech. Eng.) | 389 | 23% |
| III (Mat. Sci.) | 127 | 7% |
| VI (EECS) | 645 | 38% |
| X (Chem. Eng.) | 237 | 13% |
| XVI (Aero & Astro) | 198 | 12% |
| Total | 1717 | 100% |

   (a) When you learn a randomly chosen engineering student's department you get some number of bits of information. For which student department do you get the least amount of information?

   (b) **After studying Huffman codes in the next chapter,** design a Huffman code to encode the departments of randomly chosen groups of students. Show your Huffman tree and give the code for each course.

(c) If your code is used to send messages containing only the encodings of the departments for each student in groups of 100 randomly chosen students, what is the average length of such messages?

6. You're playing an online card game that uses a deck of 100 cards containing 3 Aces, 7 Kings, 25 Queens, 31 Jacks and 34 Tens. In each round of the game the cards are shuffled, you make a bet about what type of card will be drawn, then a single card is drawn and the winners are paid off. The drawn card is reinserted into the deck before the next round begins.

   (a) How much information do you receive when told that a Queen has been drawn during the current round?

   (b) Give a numeric expression for the information content received when learning about the outcome of a round.

   (c) **After you learn about Huffman codes in the next chapter,** construct a variable-length Huffman encoding that minimizes the length of messages that report the outcome of a sequence of rounds. The outcome of a single round is encoded as A (ace), K (king), Q (queen), J (jack) or X (ten). Specify your encoding for each of A, K, Q, J and X.

   (d) **Again, after studying Huffman codes,** use your code from part (c) to calculate the expected length of a message reporting the outcome of 1000 rounds (i.e., a message that contains 1000 symbols)?

   (e) The Nevada Gaming Commission regularly receives messages in which the outcome for each round is encoded using the symbols $A, K, Q, J$, and $X$. They discover that a large number of messages describing the outcome of 1000 rounds (i.e., messages with 1000 symbols) can be compressed by the LZW algorithm into files each containing 43 bytes in total. They decide to issue an indictment for running a crooked game. Why did the Commission issue the indictment?

7. Consider messages made up entirely of vowels ($A, E, I, O, U$). Here's a table of probabilities for each of the vowels:

| $l$ | $p_l$ | $\log_2(1/p_l)$ | $p_l \log_2(1/p_l)$ |
|---|---|---|---|
| $A$ | 0.22 | 2.18 | 0.48 |
| $E$ | 0.34 | 1.55 | 0.53 |
| $I$ | 0.17 | 2.57 | 0.43 |
| $O$ | 0.19 | 2.40 | 0.46 |
| $U$ | 0.08 | 3.64 | 0.29 |
| Totals | 1.00 | 12.34 | 2.19 |

   (a) Give an expression for the number of bits of information you receive when learning that a particular vowel is either $I$ or $U$.

   (b) **After studying Huffman codes in the next chapter,** use Huffman's algorithm to construct a variable-length code assuming that each vowel is encoded individually. Draw a diagram of the Huffman tree and give the encoding for each of the vowels.

(c) Using your code from part (B) above, give an expression for the expected length in bits of an encoded message transmitting 100 vowels.

(d) Ben Bitdiddle spends all night working on a more complicated encoding algorithm and sends you email claiming that using his code the expected length in bits of an encoded message transmitting 100 vowels is 197 bits. Would you pay good money for his implementation?

*MIT 6.02* **DRAFT** *Lecture Notes*
*Last update: February 13, 2012*
*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.

Non-instantaneous codes are hard to decode, though they could be uniquely decodable. For example, consider the following encoding:

"A" $\rightarrow$ 0
"B" $\rightarrow$ 01
"C" $\rightarrow$ 011
"D" $\rightarrow$ 111

This example code is not instantaneous. If we received the string 01111101, we wouldn't be able to decode the first symbol as "A" on seeing the first '0'. In fact, we can't be sure that the first symbol is "B" either. One would, in general, have to wait for the end of the message, and start the decoding from there. In this case, the sequence of symbols works out to "BDB".

This example code turns out to be uniquely decodable, but that is not always the case with a non-instantaneous code (in contrast, all instantaneous codes admit a unique decoding, which is obviously an important property).

As an example of a non-instantaneous code that is not useful (i.e., not uniquely decodable), consider

"A" $\rightarrow$ 0
"B" $\rightarrow$ 1
"C" $\rightarrow$ 01
"D" $\rightarrow$ 11

With this code, there exist many sequences of bits that do not map to a unique symbol sequence; for example, "01" could be either "AB" or just "C".

We will restrict our investigation to only instantaneous codes; most lossless compression codes are 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, 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 (they are optimal under certain conditions, stated below). First, Huffman codes, which are optimal instantaneous codes when the probabilities of the various symbols are given, and the symbols are independently and identically distributed (iid) with these probabilities, and we restrict ourselves to "symbol-by-symbol" mapping of symbols to codewords. It is a prefix-free 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, nor the IID assumption.

## ■ 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:
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

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

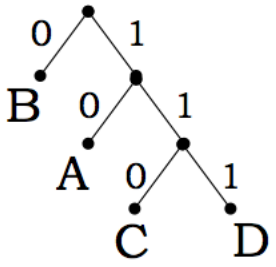


**Figure 3-1: Variable-length code from Figure 2-2 shown in the form of a code tree.**

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.

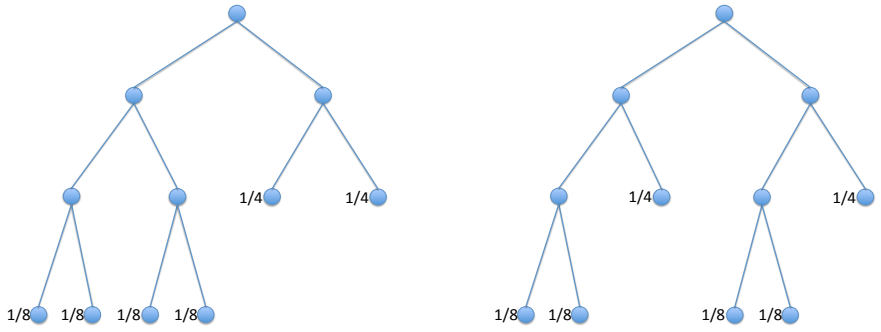


**Figure 3-2: An example of two non-isomorphic Huffman code trees, both optimal.**

**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 probabilities $1/4, 1/4, 1/8, 1/8, 1/8, 1/8$. 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 in messages in IID fashion 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., $p_x \geq p_y \geq p_z$. 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 produces a code tree whose expected length is optimal, when restricted to symbol-by-symbol coding with symbols drawn in IID fashion from a known symbol probability distribution.*

*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

```
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.**

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
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 $b$.

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.

[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 "abcabcabcabcabcabcabcabcabcabcabc"**

| 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** $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

- 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.

## ■ 3.4 Acknowledgments

Thanks to Anirudh Sivaraman for several useful comments and Muyiwa Ogunnika for a bug fix.

## ■ 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".)

7. Explain whether each of these statements is True or False. Recall that a codeword in LZW is an index into the string table.

   (a) Suppose the sender adds two strings with corresponding codewords $c_1$ and $c_2$ in that order to its string table. Then, it **may** transmit $c_2$ for the first time **before** it transmits $c_1$.

(b) Suppose the string table never gets full. If there is an entry for a string $s$ in the string table, then the sender **must** have previously sent a distinct codeword for every non-null prefix of string $s$. (If $s \equiv p + s'$ where $+$ is the string concatenation operation and $s'$ is some non-null string, then $p$ is said to be a prefix of $s$.)