*MIT 6.02* **DRAFT** *Lecture Notes*
*Spring 2011*
*Comments, questions or bug reports?*
*Please contact 6.02-staff@mit.edu*

# CHAPTER 1
# Encoding Information

In this lecture and the next, we'll be looking into *compression* techniques, which attempt to encode a message so as to transmit the same information using fewer bits. We'll be studying *lossless compression* where the recipient of the message can recover the original message exactly.

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.

## 1.1 Fixed-length vs. Variable-length Codes

Many forms of information 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, 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: 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 $x$. The mapping of information we wish to transmit or store to bit sequences to represent that information is referred to as a *code*. When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data, the resulting mapping is called a *source code*. Source codes are distinct from *channel codes* we studied in Chapters 6–10: source codes *remove redundancy* and compress the data, while channel codes *add redundancy* to improve the error resilience of the data.

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 codes (fewer bits) and infrequently occurring symbols using longer codes (more bits).

We'd expect that, on the 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 $\smile$). Examining past messages, we find that each of the four grades occurs with the probabilities shown in Figure **??**.

| 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 1-1: Possible grades shown with probabilities, fixed- and variable-length encodings**

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.

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 message 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 **??** 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 the 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 first 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 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?

## 1.2 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. How do we measure the *information content* of a message? Claude Shannon proposed that we define information as a mathematical quantity expressing the probability of occurrence of a particular sequence of symbols as contrasted with that of alternative sequences.

Suppose that we're faced with $N$ equally probable choices and we receive information that narrows it down to $M$ choices. Shannon offered the following formula for the information received:

$$\log_2(N/M) \text{ bits of information} \tag{1.1}$$

Information is measured in *bits*, which you can interpret as the number of binary digits required to encode the choice(s). Some examples:

**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 is same as the sum of the previous three examples: information is cumulative if there's no redundancy.

We can generalize equation (**??**) to deal with circumstances when the $N$ choices are not equally probable. Let $p_i$ be the probability that the $i^{th}$ choice occurs. Then the amount of information received when learning of choice $i$ is

$$\text{Information from } i^{th} \text{ choice} = \log_2(1/p_i) \text{ bits} \tag{1.2}$$

More information is received when learning of an unlikely choice (small $p_i$) than learning of a likely choice (large $p_i$). This jibes with our intuition about compression developed in §**??**: commonly occurring symbols have a higher $p_i$ and thus convey less information, so we'll use fewer bits when encoding such symbols. Similarly, infrequently occurring symbols have a lower $p_i$ and thus convey more information, so we'll use more bits when encoding such symbols. This exactly matches our goal of matching the size of the transmitted data to the information content of the message.

We can use equation (**??**) to compute the information content when learning of a choice by computing the weighted average of the information received for each particular choice:

$$\text{Information content in a choice} = \sum_{i=1}^{N} p_i \log_2(1/p_i) \tag{1.3}$$

This quantity is referred to as the *information entropy* or *Shannon's entropy* and is a lower bound on the amount of information which must be sent, on the 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 (**??**)? 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 in order 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 doesn't have enough information to tell which of the choices actually occurred.

Equation (**??**) 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 **??**, we can update the figure using equation (**??**):

| 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 1-2: Possible grades shown with probabilities and information content**

Using equation (**??**) 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 **??** 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 below.

Finding a "good" code – one where the length of the encoded message matches the information content – 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 (**??**):

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

For $p_H = 0.999$, this evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on the 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!

One final observation: 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.

## 1.3 Huffman Codes

Let's turn our attention to developing an efficient encoding given a list of symbols to be transmitted and their probabilities of occurrence in the messages to be encoded. We'll use what we've learned above: more likely symbols should have short encodings, less likely symbols should have longer encodings.

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

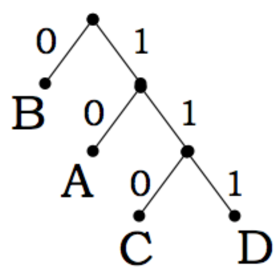


**Figure 1-3: Variable-length code from Figure ?? diagrammed as binary tree**

To encode a symbol using the tree, start at the root (the topmost node) 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. So $B$ is encoded as 0, $C$ is encoded as 110, and so on. Decoding reverses the process: use the bits from encoded message to guide a traversal of the tree starting at the root, consuming one bit each time a branch decision is required; when a symbol is reached at a leaf of the tree, that's next decoded message symbol. This process is repeated until all the encoded message bits have been consumed. So 111100 is decoded as: $111 \to D$, $10 \to A$, $0 \to B$.

Looking at the tree, we see that the most-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 to devise an algorithm for building the decoding tree for an *optimal* variable-length code while writing a term paper for a graduate course here at M.I.T. The codes are optimal in the sense that there are no other variable-length codes that produce, on the average, shorter encoded messages. Note there are many equivalent optimal codes: the 0/1 labels on any pair of branches can be reversed, giving a different encoding that has the same expected length.

Huffman's insight was the build the decoding tree *bottom up* starting with the least-probable symbols. Here are the steps involved, along with a worked example based on the variable-length code in Figure **??**:

1. Create a set $S$ of tuples, each tuple consists 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 tuples to form a new tuple (representing an interior node of the decoding tree) and compute its associated probability by summing the two probabilities from the tuples. Add this new tuple to $S$.

   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 representing the root of the decoding tree.

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

*Voila!* The result is the binary tree representing an optimal 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."

With Huffman's algorithm in hand, we can explore more complicated variable-length codes where we consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example:

| 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 1-4: Results from encoding more than one grade at a time**

We see that we can approach the Shannon lower bound of 1626 bits for 1000 grades by encoding grades in larger groups at a time, but at a cost of a more complex encoding and decoding process.

We conclude with some observations about Huffman codes:

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. We can group symbols into larger meta-symbols and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

- Huffman codes have the biggest impact on the average length of the encoded message when some symbols are substantially more probable than other symbols.

- Using *a priori* symbol probabilities (e.g., the frequency of letters in English when encoding English text) is convenient, but, in practice, symbol probabilities change message-to-message, or even within a single message.

The last observation suggests it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. This is the subject of the next lecture.

## ■ Exercises

Solutions to these exercises can be found in the tutorial problems for this lecture.

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. 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?

3. X is an unknown 8-bit binary number. You are given another 8-bit binary number, Y, and told that the Hamming distance between X (the unknown number) and Y (the number you know) is one. 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) Design a variable length Huffman code that minimizes the average number of bits in messages encoding 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's the average length of such messages?

6. You're playing an on-line 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) 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) Using your code from part (c) what is 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) Using Huffman's algorithm, construct a variable-length code assuming that each vowel is encoded individually. Please 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?