Chapter 3

Compression

In Chapter 1 we examined the fundamental unit of information, the bit, and its various abstract representations: the Boolean bit, the circuit bit, the control bit, the physical bit, the quantum bit, and the classical bit. Our never-ending quest for improvement made us want representations of single bits that are smaller, faster, stronger, smarter, safer, and cheaper.

In Chapter 2 we considered some of the issues surrounding the representation of complex objects by arrays of bits (at this point, Boolean bits). The relationship between the objects to be represented (the symbols) and the corresponding arrays of bits (known as codewords) is known as a code. We naturally want codes that are smaller and stronger, i.e., that use shorter codewords and are less vulnerable to errors. In this chapter we will consider how to get smaller codes. In Chapter 4 we will look at techniques of avoiding errors.

In Chapter 2 we considered systems of the sort shown in Figure 3.1, in which symbols are encoded into bit strings, which are transported (in space and/or time) to a decoder, which then recreates the original symbols.

![Generalized communication system](http://www.mit.edu/Courses/6.050/2013/notes/chapter3.pdf)

Typically the same code is used for a sequence of symbols, one after another. Such a sequence forms a message. The role of data compression is to convert the string of bits representing a message into a shorter string for more economical transmission, storage, or processing. The result is the system in Figure 3.2, with a compressor and an expander added. Ideally, the expander should exactly reverse the action of the compressor so that the coder and decoder need not be changed.

On first thought, this approach might seem surprising. Why is there any reason to believe that the same...
3.1 Variable-Length Encoding

In Chapter 2 Morse code was discussed as an example of a source code in which more frequently occurring letters of the English alphabet are represented by shorter codewords, and less frequently occurring letters by longer codewords. On average, messages sent using these codewords are shorter than they would be if all codewords had the same length. Variable-length encoding can be done either in the source encoder or the compressor. A general procedure for variable-length encoding will be given in Chapter 5, so a discussion of this technique is put off until that chapter.

3.2 Run Length Encoding

Suppose a message consists of long sequences of just a few symbols or characters. Then it could be encoded as a list of symbols along with the number of times they occur in each sequence. For example, the message “a B B B B c c c B B a a a” could be encoded as “a 1 B 5 c 3 B 2 a 4”. This technique works well in a relatively small set of circumstances. A good example is an image of the German flag, which could be encoded as so many black pixels, so many red pixels, and so many yellow pixels, with a great saving over specifying every pixel.

Figure 3.3: Flag of Germany (black band on top, red in middle, yellow on bottom)
Another example comes from fax technology, where a document is scanned and long groups of white (or black) pixels are transmitted as merely the number of such pixels (since there are only white and black pixels, it is not even necessary to specify the color since it can be assumed to be the other color).

Run length encoding does not work well for messages unless they have repeated sequences of the same symbol. Although it may be effective for drawings or black-and-white scanned images, it is not well suited for photographs, because too many symbols for all the different shades would be needed.

### 3.3 Static Dictionary

If a code has unused codewords, these may be assigned, as abbreviations, to frequently occurring sequences of symbols. Then such sequences could be encoded with no more bits than would be needed for a single symbol. For example, if English text is being encoded in ASCII and the DEL character is regarded as unnecessary, then its code point 127 might be assigned to the common word “the”. Practical codes offer numerous examples of this technique. The list of the codewords and their meanings is called a codebook, or dictionary. The compression technique considered here uses a dictionary which is static in the sense that it does not change from one message to the next.

An example will illustrate this technique. Before the electric telegraph, there were other schemes for transmitting messages long distances. A mechanical telegraph described in some detail by Wilson was put in place in 1796 by the British Admiralty to communicate between its headquarters in London and various ports, including Plymouth, Yarmouth, and Deal. It consisted of a series of cabins, each within sight of the next one, with six large shutters which could be rotated to horizontal (open) or vertical (closed) positions. See Figure 3.4. In operation, all shutters were open until a message was to be sent. Then all shutters were closed to signal the start of a message (operators at the cabins were supposed to look for new messages every five minutes). Then the message was sent in a sequence of shutter patterns, ending with the all-open pattern.

![British shutter telegraph cabin, 1797](image)

Figure 3.4: British shutter telegraph cabin, 1797, showing the six shutters closed, and an open window with a telescope to view another cabin (from T. Standage, “The Victorian Internet,” Berkley Books, New York; 1998; p. 15).

---

There were six shutters, and therefore 64 ($2^6$) patterns. Two of these were control codes (all-open was “idle” or “stop,” and all-closed was “start”). The other 62 were available for 24 letters of the alphabet (26 if J and U were included, which was not considered essential because they were recent additions to the English alphabet), 10 digits, an end-of-word marker, and an end-of-page marker. This left over 20 unused patterns which were assigned to commonly occurring words or phrases. The particular abbreviations used varied from time to time, but included common words like “the,” locations such as “Portsmouth,” and other words of importance to the Admiralty such as “west,” “Admiral,” “French,” and “frigate.” It also included phrases like “Commander of the West India Fleet” and “To sail, the first fair wind, to the southward.”

Perhaps the most intriguing entries in the codebook were “Court-Martial to sit” and “Sentence of court-martial to be put into execution.” Apparently courts-martial were common enough and messages about them frequent enough to justify dedicating two out of 64 code patterns to them.

This telegraph system worked well. During one demonstration a message went from London to Plymouth and back, a distance of 500 miles, in three minutes. That’s 13 times the speed of sound.

As this example suggests, long messages can be shortened considerably with a set of well chosen abbreviations. However, there is an inherent risk: the effect of errors can be much greater. If full text is transmitted, letter by letter, a single error causes a misspelling or at worst the wrong word, which humans can usually detect and correct. With abbreviations, a single incorrect bit could result in a possible but unintended meaning: “east” might be changed to “south,” or a single letter might be changed to “Sentence of court-martial to be put into execution” with significant consequences.

If abbreviations are used to compress messages, there must be a codebook showing all the abbreviations in use, distributed before the first message is sent. Because it is distributed only once, the cost of distribution is low on a per-message basis. But it has to be carefully constructed to work with all messages expected, and cannot be changed to match the needs of individual messages.

This technique works well for sets of messages that are quite similar, as might have been the case with 18th-century naval communications. It is not well suited for more diverse messages. This mechanical telegraph was never adapted for commercial or public use, which would have given it a more diverse set of messages and, perhaps, motivated the invention of other compression techniques.

### 3.4 Semi-adaptive Dictionary

The static-dictionary approach uses one dictionary, defined in advance, that applies to all messages. If, instead, a new dictionary could be defined for each message, the compression could be greater as sequences of symbols found often in the message could be made into dictionary entries.

Doing so would have several drawbacks, however. First, the new dictionary would have to be transmitted along with the encoded message, resulting in increased overhead. Second, the message would have to be analyzed to discover the best set of dictionary entries, and therefore the entire message would have to be available for analysis before any part of it could be encoded; as a result, this technique would have large latency. Third, the computer calculating the dictionary would need to have enough memory to store the entire message.

These disadvantages have limited the use of semi-adaptive dictionary compression schemes.

### 3.5 Dynamic Dictionary

What would be good for many applications would be an encoding scheme using a dictionary that is calculated on the fly (as the message is processed), does not need to accompany the message, and can be used before the end of the message has been processed. On first consideration, this might seem impossible. However, such a scheme is known and in wide use. It is the **LZW** compression technique, named after Abraham Lempel, Jacob Ziv, and Terry Welch. Lempel and Ziv actually had a series of techniques, sometimes referred to as LZ77 and LZ78, but the modification by Welch in 1984 gave the technique all the desired properties.
Welch wanted to reduce the number of bits sent to a recording head in disk drives, partly to increase the effective capacity of the disks, and partly to improve the speed of data transmission. His scheme is described below, in Section 3.7, for both the encoder and decoder. It has been widely used, in many contexts, for reversible compression of data. When applied to text files on a typical computer, the encoded files are typically half the size of the original ones. It is used in popular compression products such as Stuffit or Disk Doubler. When used on color images of drawings, with large areas that are the exact same color, the compression can be more than a factor of two. The commonly used GIF image format uses LZW compression. When used with photographs, with their gradual changes of color, the savings are much more modest.

Because this is a reversible compression technique, the original data can be reconstructed exactly, without approximation, by the decoder.

The LZW technique seems to have many advantages. However, it had one major disadvantage. It was not freely available—it was patented. The U.S. patent, No. 4,558,302, expired June 20, 2003, and patents in other countries in June, 2004, but before it did, it generated a controversy that is still an unpleasant memory for many because of the way it was handled.

### 3.5.1 The LZW Patent

Welch worked for Sperry Research Center at the time he developed his technique, and he published a paper\(^2\) describing the technique. Its advantages were quickly recognized, and it was used in a variety of compression schemes, including the Graphics Interchange Format GIF developed in 1987 by CompuServe (a national Internet Service Provider) for the purpose of reducing the size of image files in their computers. Those who defined GIF did not realize that the LZW algorithm, on which GIF was based, was patented. The article by Welch did not warn that a patent was pending. The World Wide Web came into prominence during the early 1990s, and the first graphical browsers accepted GIF images. Consequently, Web site developers routinely used GIF images, thinking the technology was in the public domain, which had been CompuServe’s intention.

By 1994, Unisys, the successor company to Sperry, realizing the value of this patent, decided to try to make money from it. They approached CompuServe, who didn’t pay too much attention at first, apparently not thinking that the threat was real. Finally, CompuServe took Unisys seriously, and the two companies together announced on December 24, 1994, that any developers writing software that creates or reads GIF images would have to license the technology from Unisys. Web site developers were not sure if their use of GIF images made them responsible for paying royalties, and they were not amused at the thought of paying for every GIF image on their sites. Soon Unisys recognized how unpopular this was, and backed off on their demands. On January 27, 1995, they announced they would not try to collect for use of existing images or for images produced by tools distributed before the end of 1994, but did insist on licensing graphics tools starting in 1995. Images produced by licensed tools would be allowed on the Web without additional payment.

In 1999, Unisys decided to collect payment from individual Web sites that might contain images from unlicensed tools, at the rate of $5000 per site, with no exception for nonprofits or small, low-traffic sites. It is not known how many Web sites actually paid that amount; it is reported that in the first eight months only one did. The feeling among many Web site developers was one of frustration and anger. Although Unisys had avoided a public-relations disaster in 1995, they had one on their hands in 1999. There were very real cultural differences between the free-wheeling Web community, often willing to share freely, and the business community, whose standard practices were designed to help make money (which, after all, is a company’s primary mission).

A non-infringing public-domain image-compression standard called PNG was created to replace GIF, but the browser manufacturers were slow to adopt yet another image format. Also, everybody knew that the patents would expire soon. The controversy has now gone away except in the memory of a few who felt particularly strongly. It is still cited as justification for or against changes in the patent system, or even the concept of software patents in general.

---

3.6 Irreversible Techniques

As for PNG, it offers some technical advantages over GIF (particularly better transparency features) and is now well supported by all popular browsers.

3.5.2 How does LZW work?

The technique, for both encoding and decoding, is illustrated with a text example in Section 3.7.

3.6 Irreversible Techniques

This section is not yet available. Sorry.

It will include as examples JPEG image compression and MP3 audio compression, and show how human perception dictates what type of information can be discarded during compression.

A demonstration of MP3 compression at http://www-mtl.mit.edu/Courses/6.050/2013/notes/mp3.html illustrates the tradeoff between file size and audio quality. A demonstration of the tradeoff between file size and image quality for images is planned. This page will illustrate the effect of compression using both GIF and JPEG on a variety of types of images.
3.7 Detail: LZW Compression

The LZW compression technique is described below and applied to two examples. Both encoders and decoders are considered. The LZW compression algorithm is “reversible,” meaning that it does not lose any information—the decoder can reconstruct the original message exactly.

3.7.1 LZW Algorithm, Example 1

Consider the encoding and decoding of the text message

itty bitty bit bin

(this peculiar phrase was designed to have repeated strings so that the dictionary builds up rapidly).

The initial set of dictionary entries is 8-bit character code with code points 0–255, with ASCII as the first 128 characters, including those in Table 3.1 which appear in the string above. Dictionary entry 256 is defined as “clear dictionary” or “start,” and 257 as “end of transmission” or “stop.” The encoded message is a sequence of numbers, the codes representing dictionary entries. Initially most dictionary entries consist of a single character, but as the message is analyzed new dictionary entries are defined that stand for strings of two or more characters. The result is summarized in Table 3.2.

<table>
<thead>
<tr>
<th>32</th>
<th>space</th>
<th>116</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>b</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
<td>256</td>
<td>start</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>257</td>
<td>stop</td>
</tr>
</tbody>
</table>

Table 3.1: LZW Example 1 Starting Dictionary

**Encoding algorithm:** Define a place to keep new dictionary entries while they are being constructed and call it **new-entry**. Start with new-entry empty. Send the start code. Then append to the new-entry the characters, one by one, from the string being compressed. As soon as new-entry fails to match any existing dictionary entry, put new-entry into the dictionary, using the next available code point, and send the code for the string without the last character (this entry is already in the dictionary). Then use the last character received as the first character of the next new-entry. When the input string ends, send the code for whatever is in new-entry followed by the stop code. That’s all there is to it.

For the benefit of those who appreciate seeing algorithms written like a computer program, this encoding algorithm is shown in Figure 3.5. When this procedure is applied to the string in question, the first character

```
0  # encoding algorithm
1  reset dictionary
2  send start code
3  for each character {
4    if new-entry appended with character is not in dictionary {
5      send code for new-entry
6      add new-entry appended with character as new dictionary entry
7      set new-entry blank
8    }
9    append character to new-entry
10  }
11  send code for new-entry
12  send stop code
```

Figure 3.5: LZW encoding algorithm
3.7 Detail: LZW Compression

Input | New dictionary entry | Transmission | New dictionary entry | Output
--- | --- | --- | --- | ---
105 | i | – | 256 (start) | –
116 | t | 258 | i t | 105 | i | – | i
116 | t | 259 | t t | 116 | t | 258 | i t | t
121 | y | 260 | t y | 116 | t | 259 | t t | t
32 | space | 261 | y space | 121 | y | 260 | t y | y
98 | b | 262 | space b | 32 | space | 261 | y space | space
105 | i | 263 | b i | 98 | b | 262 | space b | b
116 | t | – | – | – | – | – | –
116 | t | 264 | i t t | 258 | i t | 263 | b i | i t
121 | y | – | – | – | – | – | –
32 | space | 265 | t y space | 260 | t y | 264 | i t t | t y
98 | b | 266 | space-b i | – | – | – | –
105 | i | 267 | i t space | 258 | i t | 266 | space b i | i t
98 | b | – | – | – | – | – | –
105 | i | – | – | – | – | – | –
110 | n | 268 | space b i n | 266 | space b i | 267 | i t space | space b i
– | – | – | – | 110 | n | 268 | space b i n | n
– | – | – | – | – | – | – | –
257 (stop)

| 8-bit characters input |

Table 3.2: LZW Example 1 Transmission Summary

is “i” and the string consisting of just that character is already in the dictionary. So the next character is appended to new-entry, and the result is “it” which is not in the dictionary. Therefore the string which was in the dictionary, “i,” is sent and the string “i t” is added to the dictionary, at the next available position, which is 258. The new-entry is reset to be just the last character, which was not sent, so it is “t”. The next character “t” is appended and the result is “tt” which is not in the dictionary. The process repeats until the end of the string is reached.

For a while at the beginning the additional dictionary entries are all two-character strings, and there is a string transmitted for every new character encountered. However, when one of those two-character strings is encountered for the second time, its code gets sent (using fewer bits than would be required for two characters sent separately) and a new three-character dictionary entry is defined. In this example it happens with the string “i t t” (this message was designed to make this happen earlier than would be expected with normal text). Later in this example, the code for a three-character string gets transmitted, and a four-character dictionary entry defined.

In this example the codes are sent to a receiver which is expected to decode the message and produce as output the original string. The receiver does not have access to the encoder’s dictionary and therefore the decoder must build up its own dictionary.

**Decoding algorithm:** If the start code is received, reset the dictionary and set new-entry empty. For the next received code, output the character represented by the code and also place it in new-entry. Then for subsequent codes received, append the first character of the string represented by the code to new-entry, insert the result in the dictionary, then output the string for the received code and also place it in new-entry to start the next dictionary entry. When the stop code is received, nothing needs to be done; new-entry can be abandoned.
This algorithm is shown in program format in Figure 3.6.

```plaintext
0  # decoding algorithm
1  for each received code until stop code {
2    if code is start code {
3        reset dictionary
4        get next code
5        get string from dictionary  # it will be a single character
6    }
7    else {
8        get string from dictionary
9        update last dictionary entry by appending first character of string
10    }
11  add string as new dictionary entry
12  output string
13 }
```

Figure 3.6: LZW decoding algorithm

Note that the coder and decoder each create the dictionary on the fly; the dictionary therefore does not have to be explicitly transmitted, and the coder deals with the text in a single pass.

Does this work, i.e., is the number of bits needed for transmission reduced? We sent 18 8-bit characters (144 bits) in 14 9-bit transmissions (126 bits), a savings of 12.5%, for this very short example. For typical text there is not much reduction for strings under 500 bytes. Larger text files are often compressed by a factor of 2, and drawings even more.

### 3.7.2 LZW Algorithm, Example 2

Encode and decode the text message

```
itty bitty nitty grrritty bit bin
```

(again, this peculiar phrase was designed to have repeated strings so that the dictionary forms rapidly; it also has a three-long sequence `rrr` which illustrates one aspect of this algorithm). The initial set of dictionary entries include the characters in Table 3.3, which are found in the string, along with control characters for start and stop.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>256</td>
<td>start</td>
</tr>
<tr>
<td>257</td>
<td>stop</td>
</tr>
</tbody>
</table>

Table 3.3: LZW Example 2 Starting Dictionary

The same algorithms used in Example 1 can be applied here. The result is shown in Table 3.4. Note that the dictionary builds up quite rapidly, and there is one instance of a four-character dictionary entry transmitted. Was this compression effective? Definitely. A total of 33 8-bit characters (264 bits) were sent in 22 9-bit transmissions (198 bits, even including the start and stop characters), for a saving of 25% in bits.

There is one place in this example where the decoder needs to do something unusual. Normally, on receipt of a transmitted codeword, the decoder can look up its string in the dictionary and then output it and use its first character to complete the partially formed last dictionary entry, and then start the next dictionary entry. Thus only one dictionary lookup is needed. However, the algorithm presented above uses
two lookups, one for the first character, and a later one for the entire string. Why not use just one lookup for greater efficiency?

There is a special case illustrated by the transmission of code 271 in this example, where the string corresponding to the received code is not complete. The first character can be found but then before the entire string is retrieved, the entry must be completed. This happens when a character or a string appears for the first time three times in a row, and is therefore rare. The algorithm above works correctly, at a cost of an extra lookup that is seldom needed and may slow the algorithm down. A faster algorithm with a single dictionary lookup works reliably only if it detects this situation and treats it as a special case.
### Table 3.4: LZW Example 2 Transmission Summary

<table>
<thead>
<tr>
<th>Input</th>
<th>New dictionary entry</th>
<th>Transmission</th>
<th>New dictionary entry</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>105 i</td>
<td>t 258 i t</td>
<td>256 (start)</td>
<td>105 i</td>
<td>i t</td>
</tr>
<tr>
<td>116 t</td>
<td>t 259 t t</td>
<td>116 t</td>
<td>258 i t</td>
<td>t</td>
</tr>
<tr>
<td>121 y</td>
<td>y 260 t y</td>
<td>116 t</td>
<td>259 t t</td>
<td>t</td>
</tr>
<tr>
<td>32 space</td>
<td>y space</td>
<td>121 y</td>
<td>260 t y</td>
<td>y</td>
</tr>
<tr>
<td>98 b</td>
<td>b 262 space b</td>
<td>32 space</td>
<td>261 y space</td>
<td>space</td>
</tr>
<tr>
<td>105 i</td>
<td>i 263 b i</td>
<td>98 b</td>
<td>262 space b</td>
<td>b</td>
</tr>
<tr>
<td>116 t</td>
<td>t 264 i t t</td>
<td>258 i t</td>
<td>263 b i</td>
<td>i t</td>
</tr>
<tr>
<td>121 y</td>
<td>y 265 t y space</td>
<td>260 t y</td>
<td>264 i t t</td>
<td>t y</td>
</tr>
<tr>
<td>110 n</td>
<td>n 266 space n</td>
<td>32 space</td>
<td>265 t y space</td>
<td>space</td>
</tr>
<tr>
<td>105 i</td>
<td>i 267 n i</td>
<td>110 n</td>
<td>266 space n</td>
<td>n</td>
</tr>
<tr>
<td>116 t</td>
<td>t 268 i t t</td>
<td>258 i t</td>
<td>267 n i</td>
<td>i t t</td>
</tr>
<tr>
<td>121 y</td>
<td>y 269 y space g</td>
<td>261 y space</td>
<td>268 i t t</td>
<td>y space</td>
</tr>
<tr>
<td>114 r</td>
<td>r 270 g r</td>
<td>103 g</td>
<td>269 y space g</td>
<td>g</td>
</tr>
<tr>
<td>114 r</td>
<td>r 271 r r</td>
<td>114 r</td>
<td>270 g r</td>
<td>r</td>
</tr>
<tr>
<td>114 r</td>
<td>r 272 r r i</td>
<td>271 r r</td>
<td>271 r r</td>
<td>r r</td>
</tr>
<tr>
<td>116 t</td>
<td>t 273 i t t y space</td>
<td>268 i t t y</td>
<td>272 r r i</td>
<td>i t t y</td>
</tr>
<tr>
<td>98 b</td>
<td>b 274 space b i</td>
<td>262 space b</td>
<td>273 i t t y space</td>
<td>space b</td>
</tr>
<tr>
<td>116 t</td>
<td>t 275 i t space</td>
<td>258 i t</td>
<td>274 space b i</td>
<td>it</td>
</tr>
<tr>
<td>98 b</td>
<td>b 276 space b i n</td>
<td>274 space b i</td>
<td>275 i t space</td>
<td>space b i</td>
</tr>
<tr>
<td>110 n</td>
<td>n 276 space b i n</td>
<td>274 space b i</td>
<td>275 i t space</td>
<td>space b i</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>110 n</td>
<td>n</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>276 space b i n</td>
<td>n</td>
</tr>
<tr>
<td>257</td>
<td>(stop)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>