Introduction

So far we have considered a single alphabet of letters or symbols, and applied compression according to their probabilities. For small alphabets \( \mathcal{A} \), the coding efficiency may not be very efficient, and alphabet extension may improve efficiency. Encoding substrings or phrases of symbols from the symbol alphabet provide two benefits:

1. We encode more than one symbol with each code-word, and
2. with longer codewords the code becomes more efficient.

In the early days of coding, alphabet extension to the form of grouping two symbols (or three symbols) into digrams\(^1\), (or trigrams). Extending the alphabet in such a manner increased the efficiency of the prefix (Huffman) codes.

Another successful approach to data compression for alphabets such as ASCII that represent text data (computer-resident data such as books, articles, email) is based on dictionary techniques. A dictionary stores words or phrases, and conceptually each word can be indexed by an integer. The text is converted to a sequence of indices, which is more compact than the text itself.\(^2\)

Some dictionary methods apply to restricted alphabets of just letters, numbers, punctuation marks, and an important space character to delimit words. A word is a sequence of letters separated by spaces or punctuation. Non-words may be compressed by algorithms other than a dictionary lookup.

Noteworthy one-pass approaches include the two Ziv-Lempel (or Lempel-Ziv) algorithms that use sub-sequences that mix letters, spaces, and nonletter characters. The two algorithms are respectively LZ77 (or LZ1) published in 1977 \([\text{ZL77}]\), and LZ78 (or LZ2) published in 1978 \([\text{ZL78}]\).

The Ziv-Lempel or Lempel-Ziv algorithms are more sophisticated than alphabet extension. LZ77 uses the past history as a dictionary on the assumption that subsequences seen before will be seen again. LZ78 has a clever way of dynamically creating a set of substrings that are approximately equally likely, allowing code-words are of the same length. Therefore, code design is relatively simple.

8.1 Alphabet Extension

Consider a symbol alphabet \( a, b, c \) whose respective probabilities are \( \{2/3, 1/6, 1/6\} \). Since \( p(a) > 1/2 \), the prefix code is inefficient. There alphabet has 9 bigrams: \( aa, ab, ac, ba, bb, bc, ca \), \( cb \), \( cc \). Now, the probability of \( aa \) is \( p(a) \times p(a) = 4/9 \), which is close to, but less than \( 1/2 \). However the code has become more efficient than coding the three-symbol alphabet directly.

Efficient codes for small alphabets also follow from phrases representing the leaves of the parse tree. See Figure 1. Consider the following phrases: \( aaa, aab, aac, ab, ac, b, c \) of respective probabilities \( \{16/54, 4/54, 4/54, 6/54, 6/54, 6/54, 9/54, 9/54\} \). These probabilities can feed the Huffman algorithm for an optimal prefix code. The result should be more efficient than seven bigram symbols instead the 9 symbols that represent the leaves of the parse tree. Figure 1 is in the form of a parse tree, and a Huffman code based on the phrase probabilities is shown in Figure 2.

---

\(^1\) Also called bigrams.

\(^2\) Otherwise we would not use the dictionary technique.

---

Figure 1. Parse tree for alphabet extension.
8.2 LZ76: precursor to parsing

In their 1976 paper [LZ76], the basic technique is to “parse” a sequence into phrases, and following a phrase with a real or imaginary comma as a delimiter. Thus popular subsequences of symbols are treated as a unit. Both the algorithms LZ77 and LZ78 operate using a dictionary of based on the decoded data. The coding operation provides directions to the decoder about where to find the next phrase to be decoded. For the original LZ77 [ZL77], the dictionary was the entire length of the decoded sequence but practical versions using a fixed-length sliding window buffer into the past. LZ78 similarly has a maximum size of the dictionary, and its operation more closely resembles a technique described in [ZL78].

The basic idea of LZ76 was not to do compression, but rather to estimate the complexity (compressibility) of an individual data sequence. The idea is that in one scan of the sequence, recognize the longest phrase previously placed in the dictionary. The next step adds that phrase concatenated by the next symbol to the dictionary, and begins the search for the next phrase.

For example, consider the following segment of the sequence: \ldots, abbabcadabab\ldots.

Let the comma following the \ldots, at the beginning denote the comma following the previously recognized phrase plus the concatenation with the next symbol. Also assume that \texttt{abbabc} is in the dictionary and that \texttt{abbabc}a is not in the dictionary. Following this recursion, the sequence looks as follows:

\ldots, ababa, dbab \ldots

The phrase \texttt{abbabc} is added to the dictionary, where a is the innovative symbol, because it was added to the longest recognized phrase \texttt{abbabc} to form the new phrase. The parser would start looking for the longest phrase in the dictionary beginning with: \ldots, dbab \ldots

Each new phrase is composed of the existing phrase in the dictionary followed by the innovative symbol.

Initially, the dictionary has an index for the null phrase \texttt{\lambda}. In the early phase of the algorithm, the null phrase index is frequently the only phrase in the dictionary that represents the longest recognized phrase. Each new symbol seen is represented by the null phrase followed by the novel symbol. Beginning with an initial index value of 1 for the first phrase, each new phrase is assigned the next sequential phrase number. As more phrases are added to the dictionary, the algorithm reaches an increasing number of bits for each phrase.

In the beginning, the comma insertion rate is slightly greater than 1 symbol per comma. As the number of phrases grows, so does the number of symbols per comma. After awhile, if the distribution is skewed, the more popular symbols are found in longer phrases and the less popular symbols in shorter phrases. LZ76 uses a metric relative to the average number of symbols per phrase to indicate the compressibility of a finite sequence.

We summarize LZ76 as having three parts as follows.

1. The algorithm recursively parses, one symbol at a time, the next recognized phrase from the unprocessed part of the sequence. (A recognized phrase is a phrase already in the dictionary.) The parsing algorithm begins with the first symbol of the unprocessed data, and builds the longest \textit{recognized phrase} in the dictionary.

Part 1 \textit{stops} when the next symbol, added to the so-far recognized phrase is \textit{not found} in the dictionary.

2. The next symbol following the longest \textit{recognized phrase} of Part 1 is called the \textit{innovative symbol}. In part 2, the algorithm concatenates the innovative symbol to the end of the recognized phrase, forming a new phrase. The new phrase is added to the dictionary, and Part 2 ends.

3. The point of the LZ76 algorithm is to estimate the average number of symbols per comma for the individual sequence. Part 3 does the necessary bookkeeping by counting how many symbols have been processed so far, and maintaining the count of phrases in the dictionary. Part 3 ends after updating the counts, and the algorithm restarts the process of recognizing the next phrase with Part 1.

Complexity begins at the rate of “one symbol per comma”, but as the dictionary builds, the average number of “symbols per comma” increases. The compression improves (unless the distribution is uniform) until a limit is reached: the so-called “complexity of the string”. If the symbols are equally likely, we don’t expect compression.
8.3 The LZ77 algorithm family

The original LZ77 algorithm [ZL77] used a shift-register memory whose length is arbitrarily large. The shift register remembers the first part of sequence S: the symbols already seen and compressed. The shift-register memory is called the history buffer. The LZ77 encoder algorithm examines the subsequence of data symbols at the head of the yet-to-be coded part of the data sequence. The encoder tries to find the location in the history buffer of the longest matching substring for the head of the uncoded portion. The matched portion is identified by a pointer into the history buffer, and the length of the match. The matched substring is sometimes called a “phrase”. After recognizing the longest phrase and encoding it by its position and length, the algorithm added the next (innovative) symbol to the code string without compression.

The task of the decoder is relatively simple. The decoder initializes the decoding operation by storing the pointer in counter Pointer and storing the length in counter Length. The first symbol is retrieved from the history buffer by using Pointer and copied to the output string. Next, the Pointer and Length counters are decremented. If Length is not zero, the symbol retrieval from the history buffer and copy operation to the output string are repeated until the length counter reaches 0. At this point, the decoder copies the phrase to the output string. Next the decoder copies the next (innovative) symbol from the codestring and places it in the history buffer.

8.3.1 Practical LZ77 Algorithms

The original LZ77 had no efficient way to handle a symbol not already in the history buffer except the innovative symbol method. Since that time, the LZ77 family has evolved more efficient methods. Several algorithms use the first bit of the code word to indicate whether a single symbol or a substring of two or more symbols was encoded.

A patent of IBM [Jac77] that was filed prior to LZ77 described the basic idea of the pointer and the length relative to decoding, but did not describe the encoder. The IBM patent has since expired, so decoding a pointer into a history buffer and copying the number of symbols given in the length field is public domain.

Another patent by Waterhouse of Ferranti [Wat87], issued in Britain and currently owned by Stac Electronics, describes an LZ77 implementation that hashes the first three characters of the head of the yet-to-be coded data string. If the three characters have been seen before, the hash address has a pointer to the history buffer where the three characters may be found.

The Stac Electronics “Stacker” algorithm [WGB91] is based on LZ77, and the Waterhouse patent. The Stacker algorithm has been adopted for the QIC Quarter-Inch tape Cartridge (or minicartridge) compression standard known as QIC-122.

The QIC-122 algorithm uses a 2096-byte “sliding block” history buffer. Substrings or phrases are identified by a pointer variable whose value is called the Offset (distance) from the last (i.e., most recent) character to enter the buffer. The most recent (last) character to enter into the buffer has offset 1. Offset 0 is reserved for the End Marker that denotes the End of File condition. The number of bytes in the phrase is the Length variable. The Length value initializes a Length counter that tells the decoder how many bytes to copy, beginning with the Offset byte. After each byte is copied, the respective values of the pointer variable and the Length variable are decremented. When the length variable (or counter) reaches zero, the phrase had been copied. The decoder uses the BNF syntax to parse the next codeword from the QIC-122 code-stream.

For the scheme to be decodable, the decoder must keep the same history buffer. The decoder uses the Offset to locate the first character of the word, and copies the given number (Length) of bytes to the file being decoded.

The Stacker algorithm also has a Raw Data mode in case a two-byte or longer match cannot be found. In Raw Data mode, the Compression Flag Field (the first bit of each codeword is set to 0. The 0-valued Flag Field means that the next 8 bits represent a Raw Byte. The Raw Byte mode is used the first time a byte (symbol) value is seen, or when the next two-byte sequence is not found in the history buffer.

The QIC-122 standard was adopted by Quarter-Inch Cartridge Drive Standards, Inc., of Santa Barbara CA. This description of LZ77 is based on documentation found in “Data Compression Format for 1/4-inch Data Cartridge Tape Drives”, QIC-122, Revision B, 6 Feb 91. The algorithm description uses Backus-Naur Form (BNF) syntax, a meta-grammar for describing computer languages. Our description is based on the QIC-122 standard. In the BNF that follows, values 0 and 1 are terminal symbols: there is no “rewrite” rule that replaces them with something else. Variable names, enclosed in “angle brackets”, represent non-terminal expressions: <name>. Non-terminal expressions are always replaced by something else. BNF is a very powerful language for describing programming syntax, as we shall see. 4

QIC-122 Encoding Format Description

---

3The description of the invention also mentions the use of a two-byte or one-byte address to retrieve a pointer to a previous use.

4Data compression is a field that offers practical examples of many principles of computer science: the use of the Backus-Naur Form for QIC-122 is just one of many such examples.
The QIC-122 algorithm for the encoder (and decoder) for maintaining the history buffer, once the length and hptr (pointer into the history buffer) have been encoded (or decoded), follows as a simple C program snippet.

```c
for (i=0; i<length; i++, hptr++)
    history[hptr] = history[(hptr - offset) & 2047];
```

Appendix A of the QIC-122 document provides the following example:

<table>
<thead>
<tr>
<th>Input Stream</th>
<th>Output Stream</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>+----+ A 0 01000001</td>
<td>ASCII &quot;A&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B 0 01000100</td>
<td>ASCII &quot;B&quot;</td>
</tr>
<tr>
<td></td>
<td>+----+ A 0 01000001</td>
<td>ASCII &quot;A&quot;</td>
</tr>
</tbody>
</table>

| +----+ A 1 1 0000001 1100 | String Length |
| | A 1 1 0000001 1100 | String Length |
| | A 1 1 0000011 01 | ASCII "C" |
| | +----+ B 0 01000010 01 | String Length |
| | A 1 1 0000010 10 | String Length |
| | +----+ A 1 1 0000000 0 End of Data |

For the above example, the output byte stream in the hexadecimal number system is:

20 90 88 38 1C 21 E2 5C 15 80.

Figure 3 shows the steps of the above table. At the time of the encoding, the respective arrowhead shows the point where the history buffer is on the left and the yet-to-be coded data is on the right. For the first encoding, note that the history buffer is empty because there is nothing to the left of the first arrowhead. For the second arrowhead the history buffer only has character A. At the "End of Data" arrow, the yet-to-be coded data is empty because there is nothing to the right of the arrowhead.

![Diagram of the encoding process](image)

Figure 3. Example Coding of ABAABACAABABABA

8.4 The LZ78 family

The Unix `compress` algorithm is based on LZ78. Unix compress is based on the Terry Welsh algorithm [Wel84], which called it LZW. The LZW patent [Wel85] is owned by Unisys. IBM also has a patent [MW89] that covers essentially the same algorithm, called Terse, developed by Victor Miller and Mark Wegman. The patent contains a PL/I program that embodies the patent claims. Another description of the Terse algorithm appears in [MW89]. The Terse algorithm is the Ziv-Lempel (Z-L) algorithm used for...
An LZW class coder can be optionally used for the type of compression currently used in modems. The GIF algorithm, popularized by CompuServe(R), also uses LZW for the compression of pailitized 8-bit index table for color images on the IBM PC class of computer. Because of royalties, a freeware compression algorithm called PNG was developed to replace GIF. Whereas GIF is based on LZW, PNG is based on LZ77. One of the authors of PNG is also an author of the Free Software Foundation's gzip program.

Before compressing an image with GIF, there needs to be 256 table entries for the three colors red, green, and blue. These colors are the RGB colors, 8-bits per Red, Green, and Blue field, for respectively exciting the red, green, and blue phosphors that illuminate the screen of today's color displays. Much clip-art employs so-called 24-bit color and a 256-color look-up table.

The basic idea of LZ78 is to parse subsequences from the data file as the file is being compressed. The original version of the paper dealt with asymptotic properties. It did not initialize the dictionary to the symbol alphabet of the file, nor did it limit the size of the dictionary. The dictionary built up as in LZ76. Both LZW initialize the dictionary to the symbol set, and also limit the size of the dictionary.

- **How does one grow a dictionary from scratch?**
  One can use an algorithm that recognizes a phrase \( p \) (so-called in the Ziv-Lempel papers) and after encoding \( p \) by its index, adds a phrase one character longer than \( p \) to the dictionary.

- **But how does one know what phrase to add?**
  Phrase \( p \) just encoded has to be followed by some character: that's the character added to \( p \) to form the new phrase. Suppose the character following phrase \( p \) is "b". Then the "new" (innovated) symbol is b and the new phrase added to the dictionary is \( pb \). The decoder knows because the encoder sends b as the (8-bit ASCII) byte that immediately follows the prefix coded index for the phrase \( p \).

- **What index is assigned to the new phrase?**
  The index for the new phrase is one greater than the index assigned to the previously created phrase.

- **How do we get started?**
  In LZ78, Ziv and Lempel assigned index 0, a null phrase. So the first phrase is the null phrase, and the innovative character is the first character seen.

- **How do we know to stop decoding?**
  The LZ78 paper did not address this but Terse and LZW assign a code (eg, 0) as an End-Of-File symbol.

The LZW and Terse algorithms differ from LZ78 in "getting started". For units of 8-bit bytes, they seed the dictionary with indices 1 through 256 as representing byte values \( 0 \times 00, \ldots, 0 \times ff \). They have initially 257 phrases. The first 254 phrases added have a 9-bit index understood by both encoder and decoder. When the dictionary size reaches 512 entries, a 10-bit index is next used for all the phrases in the dictionary, by using a preceding 0 to the phrase numbers. Both Terse and LZW stop growing the dictionary after the index reaches 12 bits (i.e., when the dictionary contains 4096 phrases).

The original LZ78 algorithm has a simple equivalent way of viewing where the compression comes from [Lan83a]. The phrases are viewed as equally likely, and a simple parse tree displays how the compression is achieved.

The four basic subparts of a practical data compression algorithm based on the LZ78 algorithm (such as LZW) are (1) the dictionary, (2) the initial state, (3) the creation of new phrases for the dictionary, and (4) the response when the dictionary runs out of space.

1. A dictionary or table of phrases that can contain a large number (eg., 4096) of entries.

2. The initial state of the dictionary. For example,
   - The dictionary need not be initialized, symbols learned as seen.
   - In LZW, address 0 has the EOF (End of File) symbol, and indices (i.e., table addresses) 1 through N have the N symbols of the alphabet. Thus, the dictionary is "seeded" with all the 1-symbol phrases.
   - In the GIF version of LZW, indices 0 through N-1 are the 1-symbol phrases, symbol N means clear (restart) the dictionary, and N+1 means EOF.

3. The subpart that creates new phrases.
   - Following the identification of a phrase by its dictionary index the next item of the code string is called the innovative symbol. Both the encoder and decoder concatenate the innovative symbol to the end of the just-encoded and just-decoded phrase, and add this new phrase to the next available index in the dictionary.
   - Another method to add phrases is to concatenate the first symbol of the next (following) phrase to the just-decoded and just-encoded phrase. The "innovative symbol by implication" method has some subtleties, and is beyond the scope of the present treatment.

4. The subpart that deals with No More Space in the dictionary.
- A simple “No More Space” strategy fills the dictionary and compresses the rest of the file with the static dictionary.
- A sophisticated method fills the dictionary while monitoring the performance. If the average “bits per symbol” increases too much, the dictionary is re-initialized and recreated.
- Another approach maintains phrases in an LRU stack, and when full makes room by deleting the least recently used phrase.

8.4.1 LZ78: an example encoder

Consider the following sequence $S$ of symbols:

$$S = a b a b b a b a b b a b x$$

where $x$ is an “end of file” (EOF) symbol that is one of the phrases of the initial dictionary.

Following initialization, we show the generation of phrases from sequence $S$ in the example below.

The initial state of the dictionary is all one-symbol phrases $a$, $b$, and $c$, and the EOF symbol $x$. The row **Comma** shows the comma follows the recognized phrase, i.e., is located above the left parenthesis “(” that follows the phrase. The row $S$ shows the sequence of data symbols. The rows **Phrase** and **Innov** respectively have dashes ‘-’. **New** shows the recognized phrase, and the parenthesized innovative symbol.

Rows **New** and **Indx** are respectively the new phrase and its index in the dictionary. After the insertion of 6 commas, the dictionary consists of the original symbols, plus the underlined new phrases. The innovative symbol is enclosed in parentheses. Symbol $x$ is the End of File (EOF).

Creation of Phrases: Example

<table>
<thead>
<tr>
<th>Initial values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: $x$, 1: a, 2: b, 3: c.</td>
</tr>
</tbody>
</table>

The next phrase will have index 4.

<table>
<thead>
<tr>
<th>Comma:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S:</td>
<td>a(b) ab(b) a(a) b(b) abb(a) c(b) x()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phrase:</td>
<td>-</td>
<td>--</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Innov:</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>New:</td>
<td>ab abb aa bb abba cb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Indx:</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

In the above table, we have not related how the phrases were formed to the algorithm that does the work. We now describe the steps.

Step 1. Initialization.

- Initialize the dictionary to one-symbol phrases, beginning with index 0 for the EOF symbol, index 1 for the first alphabet symbol, etc. Due to the EOF, the initial dictionary contains one more symbol than the alphabet. Use the number of bits in the largest index in the dictionary to identify all indices.

Thus, if the initial dictionary contains 257 entries, (eg., an index for each 8-bit byte plus the EOF), then index 256 (binary 1 0000 0000), is its largest dictionary entry, so each phrase in the initial dictionary is encoded with 9 bits until the dictionary reaches a size of 513 (largest index 10 0000 0000), at which time 10-bit indices are used. In the example with an alphabet 256 symbols, whenever an innovative symbol is encoded, it could use its normal 8-bit code. (If the EOF could be an innovative symbol then a 9-bit code would be needed.)

Step 2. Encoding Recursion

2.1. Find the longest phrase in the dictionary that matches the yet-to-be encoded part of the input data.

2.2. Concatenate the dictionary index of this phrase, as a k-bit number per largest dictionary entry, to the code string.

2.3. Concatenate the “innovative symbol” to the code-string unless there are no more symbols. Note that in our example, we can use the original 2-bit symbol value, since only one symbol (including EOF) is expected.

2.4. If the phrase is “$x$”, or if the innovative symbol is EOF: stop. Otherwise,

(a) Concatenate the innovative symbol of (2.3) to the phrase of (2.2) to form a new phrase.

(b) add this new phrase to the Table (dictionary) at the next available index (table entry) unless table is full, and

(c) return to step 2.1.

Example encoding of $S$ with this version of LZ78.

$$S = a b a b b a b a b b a b x$$

$x$: 00, a: 01, b: 10, c: 11. *-- Initial Table.*

In the following table, we show how the phrases in the previous table were created, and encode them with the bit-vectors that represent the indices. The initial dictionary shows 2 bits per initial phrase, but as subsequent phrases are created, more leading bits are needed. The code is fixed-length, so after the fifth phrase is added the initial four phrases are respective coded with 3 bits as: 000, 001, 010, 011. Similarly, at such time the number of phrases exceeds 7, an additional 0-bit is prefixed to all phrases in the dictionary.

CE 263 Winter 1998 88

Lempel-Ziv Coding Algorithms
S: a, b, ab, a, b, abb, c, x

<table>
<thead>
<tr>
<th>Codeword</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C: 01, (10)</td>
</tr>
<tr>
<td></td>
<td>New Phrase ab: 100</td>
</tr>
<tr>
<td>2</td>
<td>New concatenation to C: 100(10)</td>
</tr>
<tr>
<td>3</td>
<td>New Phrase abb: 101</td>
</tr>
<tr>
<td>4</td>
<td>New concatenation to C: 001(01)</td>
</tr>
<tr>
<td>5</td>
<td>New Phrase aa: 110</td>
</tr>
<tr>
<td>6</td>
<td>New concatenation to C: 010(10)</td>
</tr>
<tr>
<td>7</td>
<td>New Phrase bb: 111</td>
</tr>
<tr>
<td>8</td>
<td>New concatenation to C: 101(01)</td>
</tr>
<tr>
<td>9</td>
<td>New Phrase abba: 1000</td>
</tr>
<tr>
<td>10</td>
<td>New concatenation to C: 0011(10)</td>
</tr>
<tr>
<td>11</td>
<td>New Phrase cb: 1001</td>
</tr>
<tr>
<td>12</td>
<td>New concatenation to C: 0000</td>
</tr>
</tbody>
</table>

Notes: recognition of “x” 0000, or innovation “x” (0000) terminates the code. Note we have added leading 0s to the table indices as the dictionary growth requires a longer fixed-length codetable.

Below the codewords are separated by spaces. The first part is the index and the second part in parentheses is the innovative symbol (or a “one-symbol phrase”).

\[ C(S) = 01(10) \ 100(10) \ 001(01) \ 010(10) \ 101(01) \ 0011(10) \ 0000 \]

8.4.2 An Easily Readable Shorthand for LZ78

C(S) = 1b 4b 1a 2b 5a 3b 0

For readability we have created the notation above in three steps: (1) replace each phrase part with the decimal number that represents its index to the dictionary, (2) replace innovative symbol with its letter, and (3) leave a space between abbreviated codewords.

8.4.3 Decoder Algorithm

1. Initialization: Same as for encoder
2. Decoder Recursion:
   (a) Deconcatenate the appropriate length index, look up the phrase, and copy to the data string.
   (b) Deconcatenate the innovative symbol (it is in its original form) and copy to the data string.
   (c) Concatenate innovative symbol to the phrase and add the new phrase to the dictionary.

Decoding the Example LZ78 Codestring: In Easily Readable Shorthand: C(S) = 1b 4b 1a 2b 5a 3b 0

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Result</th>
<th>New Phrase Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1b</td>
<td>Phrase 1: a</td>
</tr>
</tbody>
</table>
|           | Inov: b, new phrase: ab <- 4
|           | Decoded so far: a b |
| 2         | 4b         | Phrase 4: ab      |
|           | Inov: b, new phrase: abb <- 5
|           | Decoded so far: ab abb |
| 3         | 1a         | Phrase 1: a       |
|           | Inov: a, new phrase: aa <- 6
|           | Decoded so far: ab abb aa |
| 4         | 2b         | Phrase 2: b       |
|           | Inov: b, new phrase: bb <- 7
|           | Decoded so far: ab abb aa bb |
| 5         | 5a         | Phrase 5: abb      |
|           | Inov: a, new phrase: abba <- 8
|           | Decoded so far: ab abb aa bb abba |
| 6         | 3b         | Phrase 3: c       |
|           | Inov: b, new phrase: cb <- 9
|           | Decoded so far: ab abb aa bb abba cb |
| 7         | 0          | Phrase 0: EOF     |

In Figure 4, see the state of the parse tree after encoding all but the last three symbols of string S; i.e., after encoding S = a b a b b a a b a b b.

![Parse Tree](image)

The final tree after encoding the final part of S (a c b x) is shown in Figure 5. Note that the final phrase used a 4-bit number. In practice, at this point both the encoder and decoder would go from 3-bit codes to
4-bit codes. (A similar operation would occur when the first 3-bit code was created.)

\[
\begin{array}{c}
\text{Parse Tree} \\
\begin{array}{c}
\text{X-00} \\
\text{a-01} \\
\text{b-10} \\
\text{c-11} \\
\text{aa-110} \\
\text{ab-100} \\
\text{bb-111} \\
\text{cb-1001} \\
\text{abb-101} \\
\text{abba-1000}
\end{array}
\end{array}
\]

Figure 5. The Final Parse Tree.

Additional Reading

The book by Mark Nelson [Nel92] discusses LZ77 in the chapter titled “Sliding Window Compression”, and LZ78 in the chapter titled “LZ78 Compression”. Listings in the C language are included in the book. The programs have also appeared in issues of the magazine “Dr Dobbs Journal”, published by M&T Publishing, and have been placed on their ftp site. The second edition of the book adds a co-author and an additional chapter on Fractal compression.

References


