**LZW Compression**

LZW compression is named after its developers, A. Lempel and J. Ziv, with later modifications by Terry A. Welch. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size. LZW also performs well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals. Compression ratios of 5:1 are common for these cases. LZW is the basis of several personal computer utilities that claim to *"double the capacity of your hard drive."*

LZW compression is always used in GIF image files, and offered as an option in TIFF and PostScript. LZW compression is protected under U.S. patent number 4,558,302, granted December 10, 1985 to Sperry Corporation (now the Unisys Corporation). For information on commercial licensing, contact: Welch Licensing Department, Law Department, M/SC2SW1, Unisys Corporation, Blue Bell, Pennsylvania, 19424-0001.

LZW compression uses a **code table**, as illustrated in Fig. 27-6. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12 bit codes, each referring to one of the entries in the code table. Uncompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During uncompression, each 12 bit code would be translated via the code table back into the single bytes. Of course, this wouldn't be a useful situation.



The LZW method achieves compression by using codes 256 through 4095 to represent *sequences* of bytes. For example, code 523 may represent the sequence of three bytes: 231 124 234. Each time the compression algorithm encounters this sequence in the input file, code 523 is placed in the encoded file. During uncompression, code 523 is translated via the code table to recreate the true 3 byte sequence. The longer the sequence assigned to a single code, and the more often the sequence is repeated, the higher the compression achieved.

Although this is a simple approach, there are two major obstacles that need to be overcome: (1) how to determine what sequences should be in the code table, and (2) how to provide the uncompression program the same code table used by the compression program. The LZW algorithm exquisitely solves both these problems.

When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first codes going into the compressed file are simply the single bytes from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the uncompression program to *reconstruct* the code table directly from the compressed data, without having to transmit the code table separately.



Figure 27-7 shows a flowchart for LZW compression. Table 27-3 provides the step-by-step details for an example input file consisting of 45 bytes, the ASCII text string: *the/rain/in/Spain/falls/mainly/on/the/plain*. When we say that the LZW algorithm reads the character "a" from the input file, we mean it reads the value: 01100001 (97 expressed in 8 bits), where 97 is "a" in ASCII. When we say it writes the character "a" to the encoded file, we mean it writes: 000001100001 (97 expressed in 12 bits).



The compression algorithm uses two variables: *CHAR* and *STRING*. The variable, *CHAR*, holds a single character, i.e., a single byte value between 0 and 255. The variable, *STRING*, is a variable length string, i.e., a group of one or more characters, with each character being a single byte. In box 1 of Fig. 27-7, the program starts by taking the first byte from the input file, and placing it in the variable, *STRING*. Table 27-3 shows this action in line 1. This is followed by the algorithm looping for each additional byte in the input file, controlled in the flow diagram by box 8. Each time a byte is read from the input file (box 2), it is stored in the variable, *CHAR*. The data table is then searched to determine if the concatenation of the two variables, *STRING+CHAR*, has already been assigned a code (box 3).

If a match in the code table is *not* found, three actions are taken, as shown in boxes 4, 5 & 6. In box 4, the 12 bit code corresponding to the contents of the variable, *STRING*, is written to the compressed file. In box 5, a new code is created in the table for the concatenation of *STRING+CHAR*. In box 6, the variable, *STRING*, takes the value of the variable, *CHAR*. An example of these actions is shown in lines 2 through 10 in Table 27-3, for the first 10 bytes of the example file.

When a match in the code table *is* found (box 3), the concatenation of *STRING+CHAR*is stored in the variable, *STRING*, without any other action taking place (box 7). That is, if a matching sequence is found in the table, no action should be taken before determining if there is a *longer* matching sequence also in the table. An example of this is shown in line 11, where the sequence: *STRING+CHAR = in*, is identified as already having a code in the table. In line 12, the next character from the input file, /, is added to the sequence, and the code table is searched for: *in/*. Since this longer sequence is not in the table, the program *adds* it to the table, outputs the code for the shorter sequence that *is* in the table (code 262), and starts over searching for sequences beginning with the character, /. This flow of events is continued until there are no more characters in the input file. The program is wrapped up with the code corresponding to the current value of *STRING* being written to the compressed file (as illustrated in box 9 of Fig. 27-7 and line 45 of Table 27-3).

A flowchart of the LZW uncompression algorithm is shown in Fig. 27-8. Each code is read from the compressed file and compared to the code table to provide the translation. As each code is processed in this manner, the code table is updated so that it continually matches the one used during the compression. However, there is a small complication in the uncompression routine. There are certain combinations of data that result in the uncompression algorithm receiving a code that does not yet exist in its code table. This contingency is handled in boxes 4,5 & 6.

Only a few dozen lines of code are required for the most elementary LZW programs. The real difficulty lies in the efficient management of the code table. The brute force approach results in large memory requirements and a slow program execution. Several tricks are used in commercial LZW programs to improve their performance. For instance, the memory problem



arises because it is not know beforehand how long each of the character strings for each code will be. Most LZW programs handle this by taking advantage of the redundant nature of the code table. For example, look at line 29 in Table 27-3, where code 278 is defined to be *ainl*. Rather than storing these four bytes, code 278 could be stored as: *code 269 + l*, where code 269 was previously defined as *ain* in line 17. Likewise, code 269 would be stored as: *code 261 + n*, where code 261 was previously defined as *ai* in line 7. This pattern always holds: every code can be expressed as a previous code plus one new character.

The execution time of the compression algorithm is limited by searching the code table to determine if a match is present. As an analogy, imagine you want to find if a friend's name is listed in the telephone directory. The catch is, the only directory you have is arranged by telephone number, not alphabetical order. This requires you to search page after page trying to find the name you want. This inefficient situation is exactly the same as searching all 4096 codes for a match to a specific character string. The answer: organize the code table so that what you are looking for tells you where to look (like a partially alphabetized telephone directory). In other words, don't assign the 4096 codes to sequential locations in memory. Rather, divide the memory into sections based on what sequences will be stored there. For example, suppose we want to find if the sequence: *code 329 + x,* is in the code table. The code table should be organized so that the "*x*" indicates where to starting looking. There are many schemes for this type of code table management, and they can become quite complicated.

This brings up the last comment on LZW and similar compression schemes: *it is a very competitive field*. While the basics of data compression are relatively simple, the kinds of programs sold as commercial products are extremely sophisticated. Companies make money by selling you programs that perform compression, and jealously protect their trade-secrets through patents and the like. Don't expect to achieve the same level of performance as these programs in a few hours work.