WIKIPEDIA

# Lempel–Ziv–Welch

**Lempel–Ziv–Welch** (**LZW**) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations.[1] It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.

## Contents

## Algorithm

The scenario described by Welch's 1984 paper[1] encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence with no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary.

The idea was quickly adapted to other situations. In an image based on a color table, for example, the natural character alphabet is the set of color table indexes, and in the 1980s, many images had small color tables (on the order of 16 colors). For such a reduced alphabet, the full 12-bit codes yielded poor compression unless the image was large, so the idea of a **variable-width** code was introduced: codes typically start one bit wider than the symbols being encoded, and as each code size is used up, the code width increases by 1 bit, up to some prescribed maximum (typically 12 bits). When the maximum code value is reached, encoding proceeds using the existing table, but new codes are not generated for addition to the table.

Further refinements include reserving a code to indicate that the code table should be cleared and restored to its initial state (a "clear code", typically the first value immediately after the values for the individual alphabet characters), and a code to indicate the end of data (a "stop code", typically one greater than the clear code). The clear code lets the table be reinitialized after it fills up, which lets the encoding adapt to changing patterns in the input data. Smart encoders can monitor the compression efficiency and clear the table whenever the existing table no longer matches the input well.

Since codes are added in a manner determined by the data, the decoder mimics building the table as it sees the resulting codes. It is critical that the encoder and decoder agree on the variety of LZW used: the size of the alphabet, the maximum table size (and code width), whether variable-width encoding is used, initial code size, and whether to use the clear and stop codes (and what values they have). Most formats that employ LZW build this information into the format specification or provide explicit fields for them in a compression header for the data.

## Encoding

A high level view of the encoding algorithm is shown here:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string W in the dictionary that matches the current input.
3. Emit the dictionary index for W to output and remove W from the input.
4. Add W followed by the next symbol in the input to the dictionary.
5. Go to Step 2.

A dictionary is initialized to contain the single-character strings corresponding to all the possible input characters (and nothing else except the clear and stop codes if they're being used). The algorithm works by scanning through the input string for successively longer substrings until it finds one that is not in the dictionary. When such a string is found, the index for the string without the last character (i.e., the longest substring that *is* in the dictionary) is retrieved from the dictionary and sent to output, and the new string (including the last character) is added to the dictionary with the next available code. The last input character is then used as the next starting point to scan for substrings.

In this way, successively longer strings are registered in the dictionary and available for subsequent encoding as single output values. The algorithm works best on data with repeated patterns, so the initial parts of a message see little compression. As the message grows, however, the compression ratio tends asymptotically to the maximum (i.e., the compression factor or ratio improves on an increasing curve, and not linearly, approaching a theoretical maximum inside a limited time period rather than over infinite time).[2]

## Decoding

A high level view of the decoding algorithm is shown here:

1. Initialize the dictionary to contain all strings of length one.
2. Read the next encoded symbol: Is it encoded in the dictionary?
    1. Yes:

        1. Emit the corresponding string W to output.
        2. Concatenate the previous string emitted to output with the first symbol of W. Add this to the dictionary.
    2. No:
        1. Concatenate the previous string emitted to output with its first symbol. Call this string V.
        2. Add V to the dictionary and emit V to output.
3. Repeat Step 2 until end of input string

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the dictionary. However, the full dictionary is not needed, only the initial dictionary that contains single-character strings (and that is usually hard coded in the program, instead of sent with the encoded data). Instead, the full dictionary is rebuilt during the decoding process the following way: after decoding a value and outputting a string, the decoder concatenates it with the first character of the *next* decoded string (or the first character of current string, if the next one can't be decoded; since if the next value is unknown, then it must be the value added to the dictionary in *this* iteration, and so its first character is the same as the first character of the current string), and updates the dictionary with the new string. The decoder then proceeds to the next input (which was already read in the previous iteration) and processes it as before, and so on until it has exhausted the input stream.

## Variable-width codes

If variable-width codes are being used, the encoder and decoder must be careful to change the width at the same points in the encoded data so they don't disagree on boundaries between individual codes in the stream. In the standard version, the encoder increases the width from $p$ to $p + 1$ when a sequence $\omega + s$ is encountered that is not in the table (so that a code must be added for it) but the next available code in the table is $2^p$ (the first code requiring $p + 1$ bits). The encoder emits the code for $\omega$ at width $p$ (since that code does not require $p + 1$ bits), and then increases the code width so that the next code emitted is $p + 1$ bits wide.

The decoder is always one code behind the encoder in building the table, so when it sees the code for $\omega$, it generates an entry for code $2^p - 1$. Since this is the point where the encoder increases the code width, the decoder must increase the width here as well—at the point where it generates the largest code that fits in $p$ bits.

Unfortunately, some early implementations of the encoding algorithm increase the code width and *then* emit $\omega$ at the new width instead of the old width, so that to the decoder it looks like the width changes one code too early. This is called "early change"; it caused so much confusion that Adobe now allows both versions in PDF files, but includes an explicit flag in the header of each LZW-compressed stream to indicate whether early change is being used. Of the graphics file formats that support LZW compression, TIFF uses early change, while GIF and most others don't.

When the table is cleared in response to a clear code, both encoder and decoder change the code width after the clear code back to the initial code width, starting with the code immediately following the clear code.

## Packing order

Since the codes emitted typically do not fall on byte boundaries, the encoder and decoder must agree on how codes are packed into bytes. The two common methods are *LSB-first* ("least significant bit first") and *MSB-first* ("most significant bit first"). In LSB-first packing, the first code is aligned so that the least significant bit of the code falls in the least significant bit of the first stream byte, and if the code has more than 8 bits, the high-order bits left over are aligned with the least significant bits of the next byte; further codes are packed with LSB going into the least significant bit not yet used in the current stream byte, proceeding into further bytes as necessary. MSB-first packing aligns the first code so that its *most* significant bit falls in the MSB of the first stream byte, with overflow aligned with the MSB of the next byte; further codes are written with MSB going into the most significant bit not yet used in the current stream byte.

GIF files use LSB-first packing order. TIFF files and PDF files use MSB-first packing order.

## Example

The following example illustrates the LZW algorithm in action, showing the status of the output and the dictionary at every stage, both in encoding and decoding the data. This example has been constructed to give reasonable compression on a very short message. In real text data, repetition is generally less pronounced, so longer input streams are typically necessary before the compression builds up efficiency.

The plaintext to be encoded (from an alphabet using only the capital letters) is:

```
TOBEORNOTTOBEORTOBEORNOT#
```

The **#** is a marker used to show that the end of the message has been reached. There are thus 26 symbols in the plaintext alphabet (the 26 capital letters *A* through *Z*), and the # character represents a stop code. We arbitrarily assign these the values 1 through 26 for the letters, and 0 for '#'. (Most flavors of LZW would put the stop code *after* the data alphabet, but nothing in the basic algorithm requires that. The encoder and decoder only have to agree what value it has.)

A computer renders these as strings of bits. Five-bit codes are needed to give sufficient combinations to encompass this set of 27 values. The dictionary is initialized with these 27 values. As the dictionary grows, the codes must grow in width to accommodate the additional entries. A 5-bit code gives $2^5 = 32$ possible combinations of bits, so when the 33rd dictionary word is created, the algorithm must switch at that point from 5-bit strings to 6-bit strings (for *all* code values, including those previously output with only five bits). Note that since the all-zero code 00000 is used, and is labeled "0", the 33rd dictionary entry is labeled **32**. (Previously generated output is not affected by the code-width change, but once a 6-bit value is generated in the dictionary, it could conceivably be the next code emitted, so the width for subsequent output shifts to 6 bits to accommodate that.)

The initial dictionary, then, consists of the following entries:

| Symbol | Binary | Decimal |
|---:|---|---:|
| # | 00000 | 0 |
| A | 00001 | 1 |
| B | 00010 | 2 |
| C | 00011 | 3 |
| D | 00100 | 4 |
| E | 00101 | 5 |
| F | 00110 | 6 |
| G | 00111 | 7 |
| H | 01000 | 8 |
| I | 01001 | 9 |
| J | 01010 | 10 |
| K | 01011 | 11 |
| L | 01100 | 12 |
| M | 01101 | 13 |
| N | 01110 | 14 |
| O | 01111 | 15 |
| P | 10000 | 16 |
| Q | 10001 | 17 |
| R | 10010 | 18 |
| S | 10011 | 19 |
| T | 10100 | 20 |
| U | 10101 | 21 |
| V | 10110 | 22 |
| W | 10111 | 23 |
| X | 11000 | 24 |
| Y | 11001 | 25 |
| Z | 11010 | 26 |

## Encoding

Buffer input characters in a sequence ω until ω + next character is not in the dictionary. Emit the code for ω, and add ω + next character to the dictionary. Start buffering again with the next character. (The string to be encoded is "TOBEORNOTTOBEORTOBEORNOT#".)

| Current Sequence | Next Char | Output | | Extended Dictionary | Comments |
|---|---|---|---|---|---|
| | | Code | Bits | | |
| NULL | T | | | | |
| T | O | 20 | `10100` | 27: TO | 27 = first available code after 0 through 26 |
| O | B | 15 | `01111` | 28: OB | |
| B | E | 2 | `00010` | 29: BE | |
| E | O | 5 | `00101` | 30: EO | |
| O | R | 15 | `01111` | 31: OR | |
| R | N | 18 | `10010` | 32: RN | 32 requires 6 bits, so for next output use 6 bits |
| N | O | 14 | `001110` | 33: NO | |
| O | T | 15 | `001111` | 34: OT | |
| T | T | 20 | `010100` | 35: TT | |
| TO | B | 27 | `011011` | 36: TOB | |
| BE | O | 29 | `011101` | 37: BEO | |
| OR | T | 31 | `011111` | 38: ORT | |
| TOB | E | 36 | `100100` | 39: TOBE | |
| EO | R | 30 | `011110` | 40: EOR | |
| RN | O | 32 | `100000` | 41: RNO | |
| OT | # | 34 | `100010` | | # stops the algorithm; send the cur seq |
| | | 0 | `000000` | | and the stop code |

Unencoded length = 25 symbols × 5 bits/symbol = 125 bits
Encoded length = (6 codes × 5 bits/code) + (11 codes × 6 bits/code) = 96 bits.

Using LZW has saved 29 bits out of 125, reducing the message by more than 23%. If the message were longer, then the dictionary words would begin to represent longer and longer sections of text, sending repeated words very compactly.

## Decoding

To decode an LZW-compressed archive, one needs to know in advance the initial dictionary used, but additional entries can be reconstructed as they are always simply concatenations of previous entries.

| Input | | Output Sequence | New Dictionary Entry | | Comments |
|---|---|---|---|---|---|
| Bits | Code | | Full | Conjecture | |
| 10100 | 20 | T | | 27: T? | |
| 01111 | 15 | O | 27: TO | 28: O? | |
| 00010 | 2 | B | 28: OB | 29: B? | |
| 00101 | 5 | E | 29: BE | 30: E? | |
| 01111 | 15 | O | 30: EO | 31: O? | |
| 10010 | 18 | R | 31: OR | 32: R? | created code 31 (last to fit in 5 bits) |
| 001110 | 14 | N | 32: RN | 33: N? | so start reading input at 6 bits |
| 001111 | 15 | O | 33: NO | 34: O? | |
| 010100 | 20 | T | 34: OT | 35: T? | |
| 011011 | 27 | TO | 35: TT | 36: TO? | |
| 011101 | 29 | BE | 36: TOB | 37: BE? | 36 = TO + 1st symbol (B) of |
| 011111 | 31 | OR | 37: BEO | 38: OR? | next coded sequence received (BE) |
| 100100 | 36 | TOB | 38: ORT | 39: TOB? | |
| 011110 | 30 | EO | 39: TOBE | 40: EO? | |
| 100000 | 32 | RN | 40: EOR | 41: RN? | |
| 100010 | 34 | OT | 41: RNO | 42: OT? | |
| 000000 | 0 | # | | | |

At each stage, the decoder receives a code X; it looks X up in the table and outputs the sequence χ it codes, and it conjectures χ + ? as the entry the encoder just added – because the encoder emitted X for χ precisely because χ + ? was not in the table, and the encoder goes ahead and adds it. But what is the missing letter? It is the first letter in the sequence coded by the *next* code Z that the decoder receives. So the decoder looks up Z, decodes it into the sequence ω and takes the first letter z and tacks it onto the end of χ as the next dictionary entry.

This works as long as the codes received are in the decoder's dictionary, so that they can be decoded into sequences. What happens if the decoder receives a code Z that is not yet in its dictionary? Since the decoder is always just one code behind the encoder, Z can be in the encoder's dictionary only if the encoder *just* generated it, when emitting the previous code X for χ. Thus Z codes some ω that is χ + ?, and the decoder can determine the unknown character as follows:

1. The decoder sees X and then Z, where X codes the sequence χ and Z codes some unknown sequence ω.
2. The decoder knows that the encoder just added Z as a code for χ + some unknown character *c*, so ω = χ + *c*.
3. Since *c* is the first character in the input stream after χ, and since ω is the string appearing immediately after χ, *c* must be the first character of the sequence ω.
4. Since χ is an initial substring of ω, *c* must also be the first character of χ.
5. So even though the Z code is not in the table, the decoder is able to infer the unknown sequence and adds χ + (the first character of χ) to the table as the value of Z.

This situation occurs whenever the encoder encounters input of the form *cScSc*, where *c* is a single character, *S* is a string and *cS* is already in the dictionary, but *cSc* is not. The encoder emits the code for *cS*, putting a new code for *cSc* into the dictionary. Next it sees *cSc* in the input (starting at the second *c* of *cScSc*) and emits the new code it just inserted. The argument above shows that whenever the decoder receives a code not in its dictionary, the situation must look like this.

Although input of form *cScSc* might seem unlikely, this pattern is fairly common when the input stream is characterized by significant repetition. In particular, long strings of a single character (which are common in the kinds of images LZW is often used to encode) repeatedly generate patterns of this sort.

## Further coding

The simple scheme described above focuses on the LZW algorithm itself. Many applications apply further encoding to the sequence of output symbols. Some package the coded stream as printable characters using some form of binary-to-text encoding; this increases the encoded length and decreases the compression rate. Conversely, increased compression can often be achieved with an *adaptive entropy encoder*. Such a coder estimates the probability distribution for the value of the next symbol, based on the observed frequencies of values so far. A standard entropy encoding such as Huffman coding or arithmetic coding then uses shorter codes for values with higher probabilities.

## Uses

LZW compression became the first widely used universal data compression method on computers. A large English text file can typically be compressed via LZW to about half its original size.

LZW was used in the public-domain program compress, which became a more or less standard utility in Unix systems around 1986. It has since disappeared from many distributions, both because it infringed the LZW patent and because gzip produced better compression ratios using the LZ77-based DEFLATE algorithm, but as of 2008 at least FreeBSD includes both compress and uncompress as a part of the distribution. Several other popular compression utilities also used LZW or closely related methods.

LZW became very widely used when it became part of the GIF image format in 1987. It may also (optionally) be used in TIFF and PDF files. (Although LZW is available in Adobe Acrobat software, Acrobat by default uses DEFLATE for most text and color-table-based image data in PDF files.)

## Patents

Various patents have been issued in the United States and other countries for LZW and similar algorithms. LZ78 was covered by U.S. Patent 4,464,650 (https://patents.google.com/patent/US4464650) by Lempel, Ziv, Cohn, and Eastman, assigned to Sperry Corporation, later Unisys Corporation, filed on August 10, 1981. Two US patents were issued for the LZW algorithm: U.S. Patent 4,814,746 (https://patents.google.com/patent/US4814746) by Victor S. Miller and Mark N. Wegman and assigned to IBM, originally filed on June 1, 1983, and U.S. Patent 4,558,302 (https://patents.google.com/patent/US4558302) by Welch, assigned to Sperry Corporation, later Unisys Corporation, filed on June 20, 1983.

In addition to the above patents, Welch's 1983 patent also includes citations to several other patents that influenced it, including two 1980 Japanese patents (JP9343880A (https://patents.google.com/patent/JPS5719857A/en) and JP17790880A (https://patents.google.com/patent/JPS57101937A/en)) from NEC's Jun Kanatsu, U.S. Patent 4,021,782 (https://patents.google.com/patent/US4021782) (1974) from John S.

Hoerning, U.S. Patent 4,366,551 (https://patents.google.com/patent/US4366551) (1977) from Klaus E. Holtz, and a 1981 German patent (DE19813118676 (https://patents.google.com/patent/DE3118676C2/en)) from Karl Eckhart Heinz.[3]

In 1993–94, and again in 1999, Unisys Corporation received widespread condemnation when it attempted to enforce licensing fees for LZW in GIF images. The 1993–1994 Unisys-CompuServe controversy (CompuServe being the creator of the GIF format) prompted a Usenet comp.graphics discussion *Thoughts on a GIF-replacement file format*, which in turn fostered an email exchange that eventually culminated in the creation of the patent-unencumbered Portable Network Graphics (PNG) file format in 1995.

Unisys's US patent on the LZW algorithm expired on June 20, 2003,[4] 20 years after it had been filed. Patents that had been filed in the United Kingdom, France, Germany, Italy, Japan and Canada all expired in 2004,[4] likewise 20 years after they had been filed.

## Variants

- LZMW (1985, by V. Miller, M. Wegman)[5] – Searches input for the longest string already in the dictionary (the "current" match); adds the concatenation of the previous match with the current match to the dictionary. (Dictionary entries thus grow more rapidly; but this scheme is much more complicated to implement.) Miller and Wegman also suggest deleting low-frequency entries from the dictionary when the dictionary fills up.
- LZAP (1988, by James Storer)[6] – modification of LZMW: instead of adding just the concatenation of the previous match with the current match to the dictionary, add the concatenations of the previous match with each initial substring of the current match ("AP" stands for "all prefixes"). For example, if the previous match is "wiki" and current match is "pedia", then the LZAP encoder adds 5 new sequences to the dictionary: "wikip", "wikipe", "wikiped", "wikipedi", and "wikipedia", where the LZMW encoder adds only the one sequence "wikipedia". This eliminates some of the complexity of LZMW, at the price of adding more dictionary entries.
- LZWL is a syllable-based variant of LZW.

## See also

- LZ77 and LZ78
- LZMA
- Lempel–Ziv–Storer–Szymanski
- LZJB
- Context tree weighting
- Discrete cosine transform (DCT), a lossy compression algorithm used in JPEG and MPEG coding standards

## References

1. Welch, Terry (1984). "A Technique for High-Performance Data Compression" (http://www.cs.duke.edu/courses/spring03/cps296.5/papers/welch_1984_technique_for.pdf) (PDF). *Computer*. **17** (6): 8–19. doi:10.1109/MC.1984.1659158 (https://doi.org/10.1109%2FMC.1984.1659158).

2. Ziv, J.; Lempel, A. (1978). "Compression of individual sequences via variable-rate coding" (http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1978_variable-rate.pdf) (PDF). *IEEE Transactions on Information Theory*. **24** (5): 530. CiteSeerX 10.1.1.14.2892 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2892). doi:10.1109/TIT.1978.1055934 (https://doi.org/10.1109%2FTIT.1978.1055934).
3. U.S. Patent 4,558,302 (https://patents.google.com/patent/US4558302)
4. "LZW Patent Information" (https://web.archive.org/web/20090626052026/http://www.unisys.com/about__unisys/lzw/). *About Unisys*. Unisys. Archived from the original (http://www.unisys.com/about__unisys/lzw/) on June 26, 2009. Retrieved March 6, 2014.
5. David Salomon, *Data Compression – The complete reference*, 4th ed., page 209.
6. David Salomon, *Data Compression – The complete reference*, 4th ed., page 212.

## External links

- Rosettacode wiki, algorithm in various languages (http://rosettacode.org/wiki/LZW_compression)
- U.S. Patent 4,558,302 (https://patents.google.com/patent/US4558302), Terry A. Welch, *High speed data compression and decompression apparatus and method*
- SharpLZW – C# open source implementation (http://sourceforge.net/projects/sharplzw/)
- MIT OpenCourseWare: Lecture including LZW algorithm (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-050j-information-and-entropy-spring-2008/videos-homework-and-readings/unit-2-lecture-1/)
- Mark Nelson, *LZW Data Compression* on Dr. Dobbs Journal (October 1, 1989) (https://marknelson.us/posts/1989/10/01/lzw-data-compression.html/)
- Shrink, Reduce, and Implode: The Legacy Zip Compression Methods (https://www.hanshq.net/zip2.html) explains LZW and how it was used in PKZIP