



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export

Create a book
Download as PDF
Printable version

Languages


العربية
Català
Čeština
Deutsch
Eesti
Español
فارسی
Français
한국어
Italiano
Magyar
Nederlands
日本語
Polski
Português
Русский
Svenska
Українська
Tiếng Việt
中文

 Edit links

[Create account](#) [Log in](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)



Lempel–Ziv–Welch

From Wikipedia, the free encyclopedia

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 was the algorithm of the widely used [Unix](#) file compression utility [compress](#), and is used in the [GIF](#) image format.

Contents [hide]

- Algorithm
 - 1.1 Encoding
 - 1.2 Decoding
 - 1.3 Variable-width codes
 - 1.4 Packing order
- Example
 - 2.1 Encoding
 - 2.2 Decoding
- Further coding
- Uses
- Patents
- Variants
- See also
- References
- External links

Algorithm [\[edit\]](#)

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 for which there is 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).

Further refinements include reserving a code to indicate that the code table should be cleared (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 allows the table to 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 the 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 which variety of LZW is being used: the size of the alphabet, the maximum code width, whether variable-width encoding is being used, the initial code size, 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 [\[edit\]](#)

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 made available for subsequent encoding as single output values. The algorithm works best on data with repeated patterns, so the initial parts of a message will see little compression. As the message grows, however, the [compression ratio](#) tends asymptotically to the maximum. ^[2][\[clarification needed\]](#)

Decoding [\[edit\]](#)

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the [concatenation](#) of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it must be the value that will be added to the dictionary this iteration, and so its first character must be the same as the first character of the current string being sent to decoded output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

In this way the decoder builds up a dictionary which is identical to that used by the encoder, and uses it to decode subsequent input values. Thus the full dictionary does not need be sent with the encoded data; just the initial dictionary containing the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than being explicitly sent with the encoded data.)

Variable-width codes [\[edit\]](#)

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, or they will disagree about where the boundaries between individual codes fall 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 ω at width p (since that code does not require $p + 1$ bits), and then increases the code width so that the next code emitted will be $p + 1$ bits wide.

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

Unfortunately some early implementations of the encoding algorithm increase the code width and *then* emit ω 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. Out of graphics file formats capable of using 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.

[edit]

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

[edit]

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

TOBEORNOTTOBEORTOBEORNOT#

The initial dictionary, then, will consist 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 [\[edit\]](#)

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 almost 22%. If the message were longer, then the dictionary words would begin to represent longer and longer sections of text, allowing repeated words to be sent very compactly.

Decoding [\[edit\]](#)

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 $\chi + ?$ as the entry the encoder just added – because the encoder emitted X for χ precisely because $\chi + ?$ 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 $\chi + ?$, and the decoder can determine the unknown character as follows:

1. The decoder sees X and then Z .
2. It knows X codes the sequence χ and Z codes some unknown sequence ω .
3. It knows the encoder just added Z to code $\chi +$ some unknown character,
4. and it knows that the unknown character is the first letter z of ω .
5. But the first letter of ω ($= \chi + ?$) must then also be the first letter of χ .
6. So ω must be $\chi + x$, where **x is the first letter of χ** .
7. So the decoder figures out what Z codes even though it's not in the table,
8. and upon receiving Z , the decoder decodes it as $\chi + x$, and adds $\chi + x$ 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 [\[edit\]](#)

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 will increase the encoded length and decrease 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 [[edit](#)]

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 circa 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 [[edit](#)]

Main article: [Graphics Interchange Format § Unisys and LZW patent enforcement](#)

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](#) [[↗](#)] 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](#) [[↗](#)] 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](#) [[↗](#)] by Welch, assigned to Sperry Corporation, later Unisys Corporation, filed on June 20, 1983.

In 1993-1994, 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 (Compuserve being the creator of the GIF format) controversy engendered 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,^{[[3](#)]} 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,^{[[3](#)]} likewise 20 years after they had been filed.

Variants [[edit](#)]

- LZMW (1985, by V. Miller, M. Wegman)^{[[4](#)]} – 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)^{[[5](#)]} – 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 [[edit](#)]

- [LZ77 and LZ78](#)
- [LZMA](#)
- [Lempel–Ziv–Storer–Szymanski](#)
- [LZJB](#)
- [Context tree weighting](#)



References [[edit](#)]

- ↑ ^{*a*} ^{*b*} Welch, Terry (1984). "A Technique for High-Performance Data Compression" [[↗](#)] (PDF). *Computer* **17** (6): 8–19. doi:10.1109/MC.1984.1659158[[↗](#)].
- ↑ Ziv, J.; Lempel, A. (1978). "Compression of individual sequences via variable-rate coding" [[↗](#)] (PDF). *IEEE Transactions on Information Theory* **24** (5): 530. doi:10.1109/TIT.1978.1055934[[↗](#)].

3. [^] ^a ^b "LZW Patent Information" [↗]. *About Unisys*. Unisys. Archived from [the original](#) [↗] on June 26, 2009. Retrieved March 6, 2014.
4. [^] David Salomon, *Data Compression – The complete reference*, 4th ed., page 209
5. [^] David Salomon, *Data Compression – The complete reference*, 4th ed., page 212

External links ^{[[edit](#)]}

- [Rosettacode wiki, algorithm in various languages](#) [↗]
- [U.S. Patent 4,558,302](#) [↗], Terry A. Welch, *High speed data compression and decompression apparatus and method*
- [SharpLZW - C# open source implementation](#) [↗]
- MIT OpenCourseWare: [Lecture including LZW algorithm](#) [↗]
- [Mark Nelson, LZW Data Compression on Dr. Dobbs Journal \(Oct 1989\)](#) [↗]

v · t · e		Data compression methods	[hide]
Lossless	Entropy type	Unary · Arithmetic · Golomb · Huffman (Adaptive · Canonical · Modified) · Range · Shannon · Shannon–Fano · Shannon–Fano–Elias · Tunstall · Universal (Exp-Golomb · Fibonacci · Gamma · Levenshtein)	
	Dictionary type	Byte pair encoding · DEFLATE · Lempel–Ziv (LZ77 / LZ78 (LZ1 / LZ2) · LZJB · LZMA · LZO · LZRW · LZS · LZSS · LZW · LZWL · LZX · LZ4 · Statistical)	
	Other types	BWT · CTW · Delta · DMC · MTF · PAQ · PPM · RLE	
Audio	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Companding · Convolution · Dynamic range · Latency · Nyquist–Shannon theorem · Sampling · Sound quality · Speech coding · Sub-band coding	
	Codec parts	A-law · μ-law · ACELP · ADPCM · CELP · DPCM · Fourier transform · LPC (LAR · LSP) · MDCT · Psychoacoustic model · WLPC	
Image	Concepts	Chroma subsampling · Coding tree unit · Color space · Compression artifact · Image resolution · Macroblock · Pixel · PSNR · Quantization · Standard test image	
	Methods	Chain code · DCT · EZW · Fractal · KLT · LP · RLE · SPIHT · Wavelet	
Video	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Display resolution · Frame · Frame rate · Frame types · Interlace · Video characteristics · Video quality	
	Codec parts	Lapped transform · DCT · Deblocking filter · Motion compensation	
Theory	Entropy · Kolmogorov complexity · Lossy · Quantization · Rate–distortion · Redundancy · Timeline of information theory		
<div><div> Compression formats ·  Compression software (codecs)</div></div>			

Categories: [Lossless compression algorithms](#)