



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

[Български](#)

[Català](#)

[Čeština](#)

[Deutsch](#)

[Español](#)

[Français](#)

[한국어](#)

[Italiano](#)

[Magyar](#)

[日本語](#)

[Occitan](#)

[Polski](#)

[Português](#)

[Русский](#)

[Slovenščina](#)

[Svenska](#)

[中文](#)

 [Edit links](#)

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

Article

[Talk](#)

[Read](#)

[Edit](#)

[View history](#)



Lempel–Ziv–Markov chain algorithm

From Wikipedia, the free encyclopedia

"LZMA" redirects here. For the airport with the ICAO code "LZMA", see [Martin Airport \(Slovakia\)](#).



This article's **lead section** may not adequately **summarize key points of its contents**. Please consider expanding the lead to [provide an accessible overview](#) of all important aspects of the article. *(July 2014)*



This article **is written like a manual or guidebook**. Please help [rewrite this article](#) from a descriptive, [neutral point of view](#), and remove advice or instruction. *(July 2014)*

The **Lempel–Ziv–Markov chain algorithm** (**LZMA**) is an [algorithm](#) used to perform [lossless data compression](#). It has been under development either since 1998 or 1996^[1] and was first used in the [7z](#) format of the 7-Zip archiver. This algorithm uses a [dictionary compression](#) scheme somewhat similar to the [LZ77](#) algorithm published by [Abraham Lempel](#) and [Jacob Ziv](#) in 1977 and features a high compression ratio (generally higher than [bzip2](#))^{[2][3]} and a variable compression-dictionary size (up to 4 [GB](#)),^[4] while still maintaining decompression speed similar to other commonly used compression algorithms.^[5]

LZMA2 is a simple container format that can include both uncompressed data and LZMA data, possibly with multiple different LZMA encoding parameters. LZMA2 supports arbitrarily scalable multithreaded compression and decompression and efficient compression of data which is partially incompressible.

Contents [hide]

- Overview
- Compressed format overview
- Decompression algorithm details
 - Range coding of bits
 - Range coding of integers
 - LZMA configuration
 - LZMA coding contexts
 - LZMA2 format
 - xz and 7z formats
- Compression algorithm details
 - Range encoder
 - Dictionary search data structures
 - Hash chains
 - Binary trees
 - Patricia tries
 - LZMA encoder
 - Fast encoder
 - Normal encoder
 - LZMA2 encoder
 - Upper encoding layers
- 7-Zip reference implementation
- Implementations
- See also
- References
- External links

Overview [edit]



This section **needs additional citations for verification**. Please help [improve this article](#) by [adding citations to reliable sources](#). Unsourced material may be challenged and removed. *(July 2010)*

LZMA uses a dictionary compression algorithm (a variant of [LZ77](#) with huge dictionary sizes and special support

for repeatedly used match distances), whose output is then encoded with a [range encoder](#), using a complex model to make a probability prediction of each bit. The dictionary compressor finds matches using sophisticated dictionary data structures, and produces a stream of literal symbols and phrase references, which is encoded one bit at a time by the range encoder: many encodings are possible, and a [dynamic programming](#) algorithm is used to select an optimal one under certain approximations.

Prior to LZMA, most encoder models were purely byte-based (i.e. they coded each bit using only a cascade of contexts to represent the dependencies on previous bits from the same byte). The main innovation of LZMA is that instead of a generic byte-based model, LZMA's model uses contexts specific to the bitfields in each representation of a literal or phrase: this is nearly as simple as a generic byte-based model, but gives much better compression because it avoids mixing unrelated bits together in the same context. Furthermore, compared to classic dictionary compression (such as the one used in [zip](#) and [gzip](#) formats), the dictionary sizes can be and usually are much larger, taking advantage of the large amount of memory available on modern systems.

Compressed format overview [\[edit\]](#)



This section **needs additional citations for verification**. Please help [improve this article](#) by [adding citations to reliable sources](#). Unsourced material may be challenged and removed. *(July 2010)*

In LZMA compression, the compressed stream is a stream of bits, encoded using an adaptive binary range coder. The stream is divided into packets, each packet describing either a single byte, or an LZ77 sequence with its length and distance implicitly or explicitly encoded. Each part of each packet is modeled with independent contexts, so the probability predictions for each bit are correlated with the values of that bit (and related bits from the same field) in previous packets of the same type.

There are 7 types of packets:^{[*[citation needed](#)*]}

packed code (bit sequence)	packet name	packet description
0 + byteCode	LIT	A single byte encoded using an adaptive binary range coder.
1+0 + len + dist	MATCH	A typical LZ77 sequence describing sequence length and distance.
1+1+0+0	SHORTREP	A one-byte LZ77 sequence. Distance is equal to the last used LZ77 distance.
1+1+0+1 + len	LONGREP[0]	An LZ77 sequence. Distance is equal to the last used LZ77 distance.
1+1+1+0 + len	LONGREP[1]	An LZ77 sequence. Distance is equal to the second last used LZ77 distance.
1+1+1+1+0 + len	LONGREP[2]	An LZ77 sequence. Distance is equal to the third last used LZ77 distance.
1+1+1+1+1 + len	LONGREP[3]	An LZ77 sequence. Distance is equal to the fourth last used LZ77 distance.

LONGREP[*] refers to LONGREP[0-3] packets, *REP refers to both LONGREP and SHORTREP, and *MATCH refers to both MATCH and *REP.

LONGREP[n] packets remove the distance used from the list of the most recent distances and reinsert it at the front, to avoid useless repeated entry, while MATCH just adds the distance to the front even if already present in the list and SHORTREP and LONGREP[0] don't alter the list.

The length is encoded as follows:

Length code (bit sequence)	Description
0+ 3 bits	The length encoded using 3 bits, gives the lengths range from 2 to 9.
1+0+ 3 bits	The length encoded using 3 bits, gives the lengths range from 10 to 17.
1+1+ 8 bits	The length encoded using 8 bits, gives the lengths range from 18 to 273.

As in LZ77, the length is not limited by the distance, because copying from the dictionary is defined as if the copy was performed byte by byte, keeping the distance constant.

Distances are logically 32-bit and distance 0 points to the most recently added byte in the dictionary.

The distance encoding starts with a 6-bit "distance slot", which determines how many further bits are needed. Distances are decoded as a binary concatenation of, from most to least significant, two bits depending on the distance slot, some bits encoded with fixed 0.5 probability, and some context encoded bits, according to the following table (distance slots 0-3 directly encode distances 0-3).

6-bit distance slot	Highest 2 bits	Fixed 0.5 probability bits	Context encoded bits
0	00	0	0
1	01	0	0
2	10	0	0
3	11	0	0
4	10	0	1
5	11	0	1
6	10	0	2
7	11	0	2
8	10	0	3
9	11	0	3
10	10	0	4
11	11	0	4
12	10	0	5
13	11	0	5
14-62 (even)	10	$((\text{slot} / 2) - 5)$	4
15-63 (odd)	11	$((\text{slot} - 1) / 2) - 5)$	4

Decompression algorithm details [\[edit\]](#)



This section **possibly contains original research**. Please [improve it](#) by [verifying](#) the claims made and adding [inline citations](#). Statements consisting only of original research should be removed. *(April 2012)*

No complete natural language specification of the compressed format seems to exist, other than the one attempted in the following text.

The description below is based on the compact [XZ](#) Embedded decoder by Lasse Collin included in the Linux kernel source^[6] from which the LZMA and LZMA2 algorithm details can be relatively easily deduced: thus, while citing source code as reference isn't ideal, any programmer should be able to check the claims below with a few hours of work.

Range coding of bits [\[edit\]](#)

LZMA data is at the lowest level decoded one bit at a time by the range decoder, at the direction of the LZMA decoder.

Context-based range decoding is invoked by the LZMA algorithm passing it a reference to the "context", which consists of the unsigned 11-bit variable *prob* (typically implemented using a 16-bit data type) representing the predicted probability of the bit being 1, which is read and updated by the range decoder (and should be initialized to 2^{10} , representing 0.5 probability).

Fixed probability range decoding instead assumes a 0.5 probability, but operates slightly differently from context-based range decoding.

The range decoder state consists of two unsigned 32-bit variables, *range* (representing the range size), and *code* (representing the encoded point within the range).

Initialization of the range decoder consists of setting *range* to $2^{32} - 1$, and *code* to the 32-bit value starting at the second byte in the stream interpreted as big-endian; the first byte in the stream is completely ignored.

Normalization proceeds in this way:

1. Shift both *range* and *code* left by 8 bits
2. Read a byte from the compressed stream

3. Set the least significant 8 bits of *code* to the byte value read

Context-based range decoding of a bit using the *prob* probability variable proceeds in this way:

1. If *range* is less than 2^{24} , perform normalization
2. Set *bound* to $\text{floor}(\text{range} / 2^{11}) * \text{prob}$
3. If *code* is less than *bound*:
 1. Set *range* to *bound*
 2. Set *prob* to $\text{prob} + \text{floor}((2^{11} - \text{prob}) / 2^5)$
 3. Return bit 0
4. Otherwise (if *code* is greater than or equal to the *bound*):
 1. Set *range* to $\text{range} - \text{bound}$
 2. Set *code* to $\text{code} - \text{bound}$
 3. Set *prob* to $\text{prob} - \text{floor}(\text{prob} / 2^5)$
 4. Return bit 1

Fixed-probability range decoding of a bit proceeds in this way:

1. If *range* is less than 2^{24} , perform normalization
2. Set *range* to $\text{floor}(\text{range} / 2)$
3. If *code* is less than *range*:
 1. Return bit 0
4. Otherwise (if *code* is greater or equal than *range*):
 1. Set *code* to $\text{code} - \text{range}$
 2. Return bit 1

The Linux kernel implementation of fixed-probability decoding in `rc_direct`, for performance reasons, doesn't include a conditional branch, but instead subtracts *range* from *code* unconditionally, and uses the resulting sign bit to both decide the bit to return, and to generate a mask that is combined with *code* and added to *range*.

Note that:

1. The division by 2^{11} when computing *bound* and floor operation is done before the multiplication, not after (apparently to avoid requiring fast hardware support for 32-bit multiplication with a 64-bit result)
2. Fixed probability decoding is not strictly equivalent to context-based range decoding with any *prob* value, due to the fact that context-based range decoding discards the lower 11 bits of *range* before multiplying by *prob* as just described, while fixed probability decoding only discards the last bit

Range coding of integers [\[edit\]](#)

The range decoder also provides the bit-tree, reverse bit-tree and fixed probability integer decoding facilities, which are used to decode integers, and generalize the single-bit decoding described above. To decode unsigned integers less than *limit*, an array of $(\text{limit} - 1)$ 11-bit probability variables is provided, which are conceptually arranged as the internal nodes of a complete binary tree with *limit* leaves.

Non-reverse bit-tree decoding works by keeping a pointer to the tree of variables, which starts at the root. As long as the pointer doesn't point to a leaf, a bit is decoded using the variable indicated by the pointer, and the pointer is moved to either the left or right children depending on whether the bit is 0 or 1; when the pointer points to a leaf, the number associated with the leaf is returned.

Non-reverse bit-tree decoding thus happens from most significant to least significant bit, stopping when only one value in the valid range is possible (this conceptually allows to have range sizes that are not powers of two, even though LZMA doesn't make use of this).

Reverse bit-tree decoding instead decodes from least significant bit to most significant bits, and thus only supports ranges that are powers of two, and always decodes the same number of bits. It is equivalent to performing non-reverse bittree decoding with a power of two *limit*, and reversing the last $\log_2(\text{limit})$ bits of the result.

Note that in the `rc_bittree` function in the Linux kernel, integers are actually returned in the $[\text{limit}, 2 * \text{limit})$ range (with *limit* added to the conceptual value), and the variable at index 0 in the array is unused, while the one at index 1 is the root, and the left and right children indices are computed as $2i$ and $2i + 1$. The `rc_bittree_reverse` function instead adds integers in the $[0, \text{limit})$ range to a caller-provided variable, where *limit* is implicitly represented by its logarithm, and has its own independent implementation for efficiency reasons.

Fixed probability integer decoding simply performs fixed probability bit decoding repeatedly, reading bits from the most to the least significant.

LZMA configuration [\[edit\]](#)

The LZMA decoder is configured by an *lcppb* "properties" byte and a dictionary size.

The value of the *lcppb* byte is $lc + lp * 9 + pb * 9 * 5$, where:

- *lc* is the number of high bits of the previous byte to use as a context for literal encoding (the default value used by the LZMA SDK is 3)
- *lp* is the number of low bits of the dictionary position to include in *literal_pos_state* (the default value used by the LZMA SDK is 0)
- *pb* is the number of low bits of the dictionary position to include in *pos_state* (the default value used by the LZMA SDK is 2)

In non-LZMA2 streams, *lc* must not be greater than 8, and *lp* and *pb* must not be greater than 4. In LZMA2 streams, (*lc* + *lp*) and *pb* must not be greater than 4.

In the 7-zip LZMA file format, configuration is performed by a header containing the "properties" byte followed by the 32-bit little-endian dictionary size in bytes. In LZMA2, the properties byte can optionally be changed at the start of LZMA2 LZMA packets, while the dictionary size is specified in the LZMA2 header as later described.

LZMA coding contexts [\[edit\]](#)

The LZMA packet format has already been described, and this section specifies how LZMA statistically models the LZ-encoded streams, or in other words which probability variables are passed to the range decoder to decode each bit.

Those probability variables are implemented as multi-dimensional arrays; before introducing them, a few values that are used as indices in these multidimensional arrays are defined.

The *state* value is conceptually based on which of the patterns in the following table match the latest 2-4 packet types seen, and is implemented as a state machine state updated according to the transition table listed in the table every time a packet is output.

The initial state is 0, and thus packets before the beginning are assumed to be LIT packets.

state	previous packets				next state when next packet is			
	4th previous	3rd previous	2nd previous	previous	LIT	MATCH	LONGREP[*]	SHORTREP
0		LIT	LIT	LIT	0	7	8	9
1		MATCH	LIT	LIT	0	7	8	9
2		LONGREP[*]	LIT	LIT	0	7	8	9
	*MATCH	SHORTREP						
3	LIT	SHORTREP	LIT	LIT	0	7	8	9
4			MATCH	LIT	1	7	8	9
5			LONGREP[*]	LIT	2	7	8	9
		*MATCH	SHORTREP					
6		LIT	SHORTREP	LIT	3	7	8	9
7			LIT	MATCH	4	10	11	11
8			LIT	LONGREP[*]	5	10	11	11
9			LIT	SHORTREP	6	10	11	11
10			*MATCH	MATCH	4	10	11	11
11			*MATCH	*REP	5	10	11	11

The *pos_state* and *literal_pos_state* values consist of respectively the *pb* and *lp* (up to 4, from the LZMA header or LZMA2 properties packet) least significant bits of the dictionary position (the number of bytes coded since the last dictionary reset modulo the dictionary size). Note that the dictionary size is normally the multiple of a large power of 2, so these values are equivalently described as the least significant bits of the number of uncompressed bytes seen since the last dictionary reset.

The *prev_byte_lc_msbs* value is set to the *lc* (up to 4, from the LZMA header or LZMA2 properties packet) most significant bits of the previous uncompressed byte.

The *is_REP* value denotes whether a packet that includes a length is a LONGREP rather than a MATCH.

The *match_byte* value is the byte that would have been decoded if a SHORTREP packet had been used (in other words, the byte found at the dictionary at the last used distance); it is only used just after a *MATCH packet.

literal_bit_mode is an array of 8 values in the 0-2 range, one for each bit position in a byte, which are 1 or 2 if the previous packet was a *MATCH and it is either the most significant bit position or all the more significant bits in the literal to encode/decode are equal to the bits in the corresponding positions in *match_byte*, while otherwise it is 0; the choice between the 1 or 2 values depends on the value of the bit at the same position in *match_byte*.

The literal/Literal set of variables can be seen as a "pseudo-bit-tree" similar to a bit-tree but with 3 variables instead of 1 in every node, chosen depending on the *literal_bit_mode* value at the bit position of the next bit to decode after the bit-tree context denoted by the node.

The claim, found in some sources, that literals after a *MATCH are coded as the XOR of the byte value with *match_byte* is incorrect; they are instead coded simply as their byte value, but using the pseudo-bit-tree just described and the additional context listed in the table below.

The probability variable groups used in LZMA are those:

XZ name	LZMA SDK name	parameterized by	used when	coding mode	if bit 0 then	if bit 1 then
is_match	IsMatch	<i>state, pos_state</i>	packet start	bit	LIT	*MATCH
is_rep	IsRep	<i>state</i>	after bit sequence 1	bit	MATCH	*REP
is_rep0	IsRepG0	<i>state</i>	after bit sequence 11	bit	SHORTREP/ LONGREP[0]	LONGREP[1-3]
is_rep0_long	IsRep0Long	<i>state, pos_state</i>	after bit sequence 110	bit	SHORTREP	LONGREP[0]
is_rep1	IsRepG1	<i>state</i>	after bit sequence 111	bit	LONGREP[1]	LONGREP[2/3]
is_rep2	IsRepG2	<i>state</i>	after bit sequence 1111	bit	LONGREP[2]	LONGREP[3]
literal	Literal	<i>prev_byte_lc_msbs, literal_pos_state, literal_bit_mode</i> [bit position], bit-tree context	after bit sequence 0	256 values pseudo-bit-tree	literal byte value	
dist_slot	PosSlot	$\min(\text{match_length}, 5)$, bit-tree context	distance: start	64 values bit-tree	distance slot	
dist_special	SpecPos	<i>distance_slot</i> , reverse bit-tree context	distance: 4-13 distance slots	$((\text{distance_slot} \gg 1) - 1)$ -bit reverse bit-tree	low bits of distance	
dist_align	Align	reverse bit-tree context	distance: 14+ distance slots, after fixed probability bits	4-bit reverse bit-tree	low bits of distance	
len_dec.choice	LenChoice	<i>is_REP</i>	match length: start	bit	2-9 length	10+ length

len_dec.choice2	LenChoice2	<i>is_REP</i>	match length: after bit sequence 1	bit	10-17 length	18+ length
len_dec.low	LenLow	<i>is_REP</i> , <i>pos_state</i> , bit-tree context	match length: after bit sequence 0	8 values bit-tree	low bits of length	
len_dec.mid	LenMid	<i>is_REP</i> , <i>pos_state</i> , bit-tree context	match length: after bit sequence 10	8 values bit-tree	middle bits of length	
len_dec.high	LenHigh	<i>is_REP</i> , bit-tree context	match length: after bit sequence 11	256 values bit-tree	high bits of length	

LZMA2 format [\[edit\]](#)

The LZMA2 container supports multiple runs of compressed LZMA data and uncompressed data. Each LZMA compressed run can have a different LZMA configuration and dictionary. This improves the compression of partially or completely incompressible files and allows multithreaded compression and multithreaded decompression by breaking the file into runs that can be compressed or decompressed independently in parallel.

The LZMA2 header consists of a byte indicating the dictionary size:

- 40 indicates a 4 GB - 1 dictionary size
- Even values less than 40 indicate a $2^{(v/2 + 12)}$ bytes dictionary size
- Odd values less than 40 indicate a $3 \cdot 2^{((v - 1)/2 + 11)}$ bytes dictionary size
- Values higher than 40 are invalid

LZMA2 data consists of packets starting with a control byte, with the following values:

- 0 denotes the end of the file
- 1 denotes a dictionary reset followed by an uncompressed chunk
- 2 denotes an uncompressed chunk without a dictionary reset
- 3-0x7f are invalid values
- 0x80-0xff denotes an LZMA chunk, where the lowest 5 bits are used as bit 16-20 of the uncompressed size minus one, and bit 5-6 indicates what should be reset

Bits 5-6 for LZMA chunks can be:

- 0: nothing reset
- 1: state reset
- 2: state reset, properties reset using properties byte
- 3: state reset, properties reset using properties byte, dictionary reset

LZMA state resets cause a reset of all LZMA state except the dictionary, and specifically:

- The range coder
- The *state* value
- The last distances for repeated matches
- All LZMA probabilities

Uncompressed chunks consist of:

- A 16-bit big-endian value encoding the data size minus one
- The data to be copied verbatim into the dictionary and the output

LZMA chunks consist of:

- A 16-bit big-endian value encoding the low 16-bits of the uncompressed size minus one
- A 16-bit big-endian value encoding the compressed size minus one
- A properties/lclppb byte if bit 6 in the control byte is set
- The LZMA compressed data, starting with the 5 bytes (of which the first is ignored) used to initialize the range coder (which are included in the compressed size)

xz and 7z formats [\[edit\]](#)

The `.xz` format, which can contain LZMA2 data, is documented at tukaani.org,^[7] while the `.7z` file format, which can contain either LZMA or LZMA2 data, is documented in the `7zformat.txt` file contained in the LZMA SDK.^[8]

Compression algorithm details [\[edit\]](#)

Similar to the decompression format situation, no complete natural language specification of the encoding techniques in `7-zip` or `xz` seems to exist, other than the one attempted in the following text.

The description below is based on the `XZ` for Java encoder by Lasse Collin,^[9] which appears to be the most readable among several rewrites of the original 7-zip using the same algorithms: again, while citing source code as reference isn't ideal, any programmer should be able to check the claims below with a few hours of work.

Range encoder [\[edit\]](#)

The range encoder cannot make any interesting choices, and can be readily constructed based on the decoder description.

Initialization and termination are not fully determined; the `xz` encoder outputs 0 as the first byte which is ignored by the decompressor, and encodes the lower bound of the range (which matters for the final bytes).

The `xz` encoder uses an unsigned 33-bit variable called *low* (typically implemented as a 64-bit integer, initialized to 0), an unsigned 32-bit variable called *range* (initialized to $2^{32} - 1$), an unsigned 8-bit variable called *cache* (initialized to 0), and an unsigned variable called *cache_size* which needs to be large enough to store the uncompressed size (initialized to 1, typically implemented as a 64-bit integer).

The *cache*/*cache_size* variables are used to properly handle carries, and represent a number defined by a big-endian sequence starting with the *cache* value, and followed by *cache_size* 0xff bytes, which has been shifted out of the *low* register, but hasn't been written yet, because it could be incremented by one due to a carry.

Note that the first byte output will always be 0 due to the fact that *cache* and *low* are initialized to 0, and the encoder implementation; the `xz` decoder ignores this byte.

Normalization proceeds in this way:

1. If *low* is less than $(2^{32} - 2^{24})$:
 1. Output the byte stored in *cache* to the compressed stream
 2. Output *cache_size* - 1 bytes with 0xff value
 3. Set *cache* to bits 24-31 of *low*
 4. Set *cache_size* to 0
2. If *low* is greater or equal than 2^{32} :
 1. Output the byte stored in *cache* plus one to the compressed stream
 2. Output *cache_size* - 1 bytes with 0 value
 3. Set *cache* to bits 24-31 of *low*
 4. Set *cache_size* to 0
3. Increment *cache_size*
4. Set *low* to the lowest 24 bits of *low* shifted left by 8 bits
5. Set *range* to *range* shifted left by 8 bits

Context-based range encoding of a bit using the *prob* probability variable proceeds in this way:

1. If *range* is less than 2^{24} , perform normalization
2. Set *bound* to $\text{floor}(\text{range} / 2^{11}) * \text{prob}$
3. If encoding a 0 bit:
 1. Set *range* to *bound*
 2. Set *prob* to $\text{prob} + \text{floor}((2^{11} - \text{prob}) / 2^5)$
4. Otherwise (if encoding a 1 bit):
 1. Set *range* to *range* - *bound*
 2. Set *low* to *low* + *bound*

3. Set *prob* to *prob* - floor(*prob* / 2⁵)

Fixed-probability range encoding of a bit proceeds in this way:

1. If *range* is less than 2²⁴, perform normalization
2. Set *range* to floor(*range* / 2)
3. If encoding a 1 bit:
 1. Set *low* to *low* + *range*

Termination proceeds this way:

1. Perform normalization 5 times

Bit-tree encoding is performed like decoding, except that bit values are taken from the input integer to be encoded rather than from the result of the bit decoding functions.

For algorithms that try to compute the encoding with the shortest post-range-encoding size, the encoder also needs to provide an estimate of that.

Dictionary search data structures [\[edit\]](#)

The encoder needs to be able to quickly locate matches in the dictionary. Since LZMA uses very large dictionaries (potentially on the order of gigabytes) to improve compression, simply scanning the whole dictionary would result in an encoder too slow to be practically usable, so sophisticated data structures are needed to support fast match searches.

Hash chains [\[edit\]](#)

The simplest approach, called "hash chains", is parameterized by a constant *N* which can be either 2, 3 or 4, which is typically chosen so that 2^(8×*N*) is greater than or equal to the dictionary size.

It consists of creating, for each *k* less than *N*, a hash table indexed by tuples of *k* bytes, where each of the buckets contains the last position where the first *k* bytes hashed to the hash value associated with that hash table bucket.

Chaining is achieved by an additional array which stores, for every dictionary position, the last seen previous position whose first *N* bytes hash to the same value of the first *N* bytes of the position in question.

To find matches of length *N* or higher, a search is started using the *N*-sized hash table, and continued using the hash chain array; the search stop after a pre-defined number of hash chain nodes has been traversed, or when the hash chains "wraps around", indicating that the portion of the input that has been overwritten in the dictionary has been reached.

Matches of size less than *N* are instead found by simply looking at the corresponding hash table, which either contains the latest such match, if any, or a string that hashes to the same value; in the latter case, the encoder won't be able to find the match. This issue is mitigated by the fact that for distant short matches using multiple literals might require less bits, and having hash conflicts in nearby strings is relatively unlikely; using larger hash tables or even direct lookup tables can reduce the problem at the cost of higher cache miss rate and thus lower performance.

Note that all matches need to be validated to check that the actual bytes match currently at that specific dictionary position match, since the hashing mechanism only guarantees that at some past time there were characters hashing to the hash table bucket index (some implementations may not even guarantee that, because they don't initialize the data structures).

LZMA uses [Markov chains](#), as implied by "M" in its name.

Binary trees [\[edit\]](#)

The [binary tree](#) approach follows the hash chain approach, except that it logically uses a binary tree instead of a linked list for chaining.

The binary tree is maintained so that it is always both a search tree relative to the suffix lexicographic ordering, and a max-heap for the dictionary position^[10] (in other words, the root is always the most recent string, and a child cannot have been added more recently than its parent): assuming all strings are lexicographically ordered, these conditions clearly uniquely determine the binary tree (this is trivially provable by induction on the size of the tree).

Since the string to search for and the string to insert are the same, it is possible to perform both dictionary search and insertion (which requires to rotate the tree) in a single tree traversal.

Patricia tries [\[edit\]](#)

Some old LZMA encoders also supported a data structure based on [Patricia tries](#), but such support has since been dropped since it was deemed inferior to the other options.^[10]

LZMA encoder [\[edit\]](#)

LZMA encoders can freely decide which match to output, or whether to ignore the presence of matches and output literals anyway.

The ability to recall the 4 most recently used distances means that, in principle, using a match with a distance that will be needed again later may be globally optimal even if it is not locally optimal, and as a result of this, optimal LZMA compression probably requires knowledge of the whole input and might require algorithms too slow to be usable in practice.

Due to this, practical implementations tend to employ non-global heuristics.

The [xz](#) encoders use a value called *nice_len* (the default is 64): when any match of length at least *nice_len* is found, the encoder stops the search and outputs it, with the maximum matching length.

Fast encoder [\[edit\]](#)

The XZ fast encoder ^[11] (derived from the 7-zip fast encoder) is the shortest LZMA encoder in the [xz](#) source tree.

It works like this:

1. Perform combined search and insertion in the dictionary data structure
2. If any repeated distance matches with length at least *nice_len*:
 - Output the most frequently used such distance with a REP packet
3. If a match was found of length at least *nice_len*:
 - Output it with a MATCH packet
4. Set the main match to the longest match
5. Look at the nearest match of every length in decreasing length order, and until no replacement can be made:
 - Replace the main match with a match which is one character shorter, but whose distance is less than 1/128 the current main match distance
6. Set the main match length to 1 if the current main match is of length 2 and distance at least 128
7. If a repeated match was found, and its length is at least 1 character less than the main match:
 - Output the repeated match with a REP packet
8. If a repeated match was found, and its length is at least 2 character less than the main match, and the main match distance is at least 512:
 - Output the repeated match with a REP packet
9. If a repeated match was found, and its length is at least 3 character less than the main match, and the main match distance is at least 32768:
 - Output the repeated match with a REP packet
10. If the main match size is less than 2 (or there isn't any match):
 - Output a LIT packet
11. Perform a dictionary search for the next byte
12. If the next byte has a match with length at least one less than the main match length, with distance less than 1/128 times the main match distance, and if the main match length is at least 3:
 - Output a LIT packet
13. If the next byte has a match at least as long as the main match, and with less distance than the main match:
 - Output a LIT packet
14. If the next byte has a match at least one character longer than the main match, and such that 1/128 of its distance is less or equal than the main match distance:
 - Output a LIT packet
15. If the next byte has a match more than one character longer than the main match:
 - Output a LIT packet
16. If any repeated match has length at least one less than the main match length:
 - Output the most frequently used such distance with a REP packet
17. Output the main match with a MATCH packet

Normal encoder [\[edit\]](#)

The XZ normal encoder^[12] (derived from the 7-zip normal encoder) is the other LZMA encoder in the **xz** source tree, which adopts a more sophisticated approach that tries to minimize the post-range-encoding size of the generated packets.

Specifically, it encodes portions of the input using the result of a dynamic programming algorithm, where the subproblems are finding the approximately optimal encoding (the one with minimal post-range-encoding size) of the substring of length *L* starting at the byte being compressed.

The size of the portion of the input processed in the dynamic programming algorithm is determined to be the maximum between the longest dictionary match and the longest repeated match found at the start position (which is capped by the maximum LZMA match length, 273); furthermore, if a match longer than *nice_len* is found at any point in the range just defined, the dynamic programming algorithm stops, the solution for the subproblem up to that point is output, the *nice_len*-sized match is output, and a new dynamic programming problem instance is started at the byte after the match is output.

Subproblem candidate solutions are incrementally updated with candidate encodings, constructed taking the solution for a shorter substring of length *L'*, extended with all possible "tails", or sets of 1-3 packets with certain constraints that encode the input at the *L'* position. Once the final solution of a subproblem is found, the LZMA state and least used distances for it are computed, and are then used to appropriately compute post-range-encoding sizes of its extensions.

At the end of the dynamic programming optimization, the whole optimal encoding of the longest substring considered is output, and encoding continues at the first uncompressed byte not already encoded, after updating the LZMA state and least used distances.

Each subproblem is extended by a packet sequence which we call "tail", which must match one of the following patterns:

1st packet	2nd packet	3rd packet
any		
LIT	LONGREP[0]	
*MATCH	LIT	LONGREP[0]

The reason for not only extending with single packets is that subproblems only have the substring length as the parameter for performance and algorithmic complexity reasons, while an optimal dynamic programming approach would also require to have the last used distances and LZMA *state* as parameter; thus, extending with multiple packets allows to better approximate the optimal solution, and specifically to make better use of LONGREP[0] packets.

The following data is stored for each subproblem (of course, the values stored are for the candidate solution with minimum *price*), where by "tail" we refer to the packets extending the solution of the smaller subproblem, which are described directly in the following structure:

XZ for Java member name	description
price	quantity to be minimized: number of post-range-encoding bits needed to encode the string
optPrev	uncompressed size of the substring encoded by all packets except the last one
backPrev	-1 if the last packet is LIT, 0-3 if it is a repetition using the last used distance number 0-3, 4 + <i>distance</i> if it is a MATCH (this is always 0 if prev1IsLiteral is true, since the last packet can only be a LONGREP[0] in that case)
prev1IsLiteral	true if the "tail" contains more than one packet (in which case the one before the last is a LIT)
hasPrev2	true if the "tail" contains 3 packets (only valid if prev1IsLiteral is true)
optPrev2	uncompressed size of the substring encoded by all packets except the "tail" (only valid if prev1IsLiteral and hasPrev2 are true)
backPrev2	-1 if the first packet in the "tail" is LIT, 0-3 if it is a repetition using the last used distance number 0-3, 4 + <i>distance</i> if it is a MATCH (only valid if prev1IsLiteral and hasPrev2 are true)
reps[4]	the values of the 4 last used distances after the packets in the solution (computed only after the best subproblem solution has been determined)
	the LZMA <i>state</i> value after the packets in the solution (computed only after the best

state	subproblem solution has been determined)
-------	------------------------------------------

Note that in the XZ for Java implementation, the *optPrev* and *backPrev* members are reused to store a forward single-linked list of packets as part of outputting the final solution.

LZMA2 encoder [\[edit\]](#)

The XZ LZMA2 encoder processes the input in chunks (of up to 2 MB uncompressed size or 64 KB compressed size, whichever is lower), handing each chunk to the LZMA encoder, and then deciding whether to output an LZMA2 LZMA chunk including the encoded data, or to output an LZMA2 uncompressed chunk, depending on which is shorter (LZMA, like any other compressor, will necessarily expand rather than compress some kinds of data).

The LZMA state is reset only in the first block, if the caller requests a change of properties and every time a compressed chunk is output. The LZMA properties are changed only in the first block, or if the caller requests a change of properties. The dictionary is only reset in the first block.

Upper encoding layers [\[edit\]](#)

Before LZMA2 encoding, depending on the options provided, xz can apply the BCJ filter, which filters executable code to replace relative offsets with absolute ones that are more repetitive, or the delta filter, which replaces each byte with the difference between it and the byte *N* bytes before it.

Parallel encoding is performed by dividing the file in chunks which are distributed to threads, and ultimately each encoded (using, for instance, xz block encoding) separately, resulting in a dictionary reset between chunks in the output file.

7-Zip reference implementation [\[edit\]](#)

The LZMA implementation extracted from 7-Zip is available as LZMA SDK. It was originally dual-licensed under both the [GNU LGPL](#) and [Common Public License](#),^[13] with an additional special exception for linked binaries, but was placed by Igor Pavlov in the [public domain](#) on December 2, 2008, with the release of version 4.62.^[8]

LZMA2 compression, which is an improved version of LZMA,^[14] has been introduced in version 9.04 beta, of May 30, 2009.^[15]

The reference [open source](#) LZMA compression library is written in C++ and has the following properties:

- Compression speed: approximately 1 MB/s on a 2 GHz CPU.
- Decompression speed: 10–20 MB/s on a 2 GHz CPU.
- Support for [multithreading](#).


In addition to the original C++, the LZMA SDK contains reference implementations of LZMA compression and decompression ported to [ANSI C](#), [C#](#), and [Java](#).^[8] There are also third-party [Python](#) bindings for the C++ library, as well as ports of LZMA to [Pascal](#) and [Go](#).^{[16][17][18]}

The 7-Zip implementation uses several variants of [hash chains](#), [binary trees](#) and [Patricia tries](#) as the basis for its dictionary search algorithm.

Decompression-only code for LZMA generally compiles to around 5 KB, and the amount of RAM required during decompression is principally determined by the size of the [sliding window](#) used during compression. Small code size and relatively low memory overhead, particularly with smaller dictionary lengths, and free source code make the LZMA decompression algorithm well-suited to [embedded](#) applications.

Implementations [\[edit\]](#)

In addition to the 7-Zip reference implementation, the following support the LZMA format.

- [DotNetCompression](#) : a streaming implementation of LZMA in managed C# that is based on the LZMA SDK, conforms to the API of System.IO.Compression and includes assemblies for [.NET Framework](#), [.NET Compact Framework](#), [Xamarin.iOS](#), [Xamarin.Android](#), [Xamarin.Mac](#), [Windows Phone](#), [Xbox 360](#), [Silverlight](#), [Mono](#) and as a Portable Class Library.

See also [\[edit\]](#)

- [LZHAM](#) (LZ, Huffman, Arithmetic, Markov), an LZMA-like implementation that trades compression throughput for very high ratios and higher decompression throughput.^[19]
- [lzip](#), a LZMA implementation

- [xz](#), a file format incorporating LZMA2

References [\[edit\]](#)

1. [^] Igor Pavlov has asserted multiple times on [SourceForge](#) that the algorithm is his own creation. (2004-02-19). "LZMA spec?" [↗](#). Archived from [the original](#) [↗](#) on 2009-08-25. Retrieved 2013-06-16.
2. [^] Collin, Lasse (2005-05-31). "A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA" [↗](#). *The Tukaani Project* [↗](#). Retrieved 2013-06-16.
3. [^] Klausmann, Tobias (2008-05-08). "Gzip, Bzip2 and Lzma compared" [↗](#). *Blog of an Alpha animal* [↗](#). Retrieved 2013-06-16.
4. [^] Igor Pavlov (2013). "7z Format" [↗](#). Retrieved 2013-06-16.
5. [^] Mahoney, Matt. "Data Compression Explained" [↗](#). Retrieved 2013-11-13.
6. [^] Collin, Lasse; Pavlov, Igor. "lib/xz/xz_dec_lzma2.c" [↗](#). Retrieved 2013-06-16.
7. [^] "The .xz File Format" [↗](#). 2009-08-27. Retrieved 2013-06-16.
8. [^] ^a ^b ^c Igor Pavlov (2013). "LZMA SDK (Software Development Kit)" [↗](#). Retrieved 2013-06-16.
9. [^] "XZ in Java" [↗](#). Retrieved 2013-06-16.
10. [^] ^a ^b Solomon, David (2006-12-19). *Data Compression: The Complete Reference* (4 ed.). [Springer Publishing](#). p. 245. ISBN 978-1846286025.
11. [^] Collin, Lasse; Pavlov, Igor. "LZMAEncoderFast.java" [↗](#). Retrieved 2013-06-16.
12. [^] Collin, Lasse; Pavlov, Igor. "LZMAEncoderNormal.java" [↗](#). Retrieved 2013-06-16.
13. [^] "Browse /LZMA SDK/4.23" [↗](#). [Sourceforge](#). Retrieved 2014-02-12.
14. [^] "Inno Setup Help" [↗](#). [jrsoftware.org](#). Retrieved 2013-06-16. "LZMA2 is a modified version of LZMA that offers a better compression ratio for uncompressible data (random data expands about 0.005%, compared to 1.35% with original LZMA), and optionally can compress multiple parts of large files in parallel, greatly increasing compression speed but with a possible reduction in compression ratio."
15. [^] "HISTORY of the 7-Zip" [↗](#). 2012-10-26. Retrieved 2013-06-16.
16. [^] Bauch, Joachim (2010-04-07). "PyLZMA - Platform independent python bindings for the LZMA compression library." [↗](#). Retrieved 2013-06-16.
17. [^] Birtles, Alan (2006-06-13). "Programming Help: Pascal LZMA SDK" [↗](#). Retrieved 2013-06-16.
18. [^] Vieru, Andrei (2012-06-28). "compress/lzma package for Go 1" [↗](#). Retrieved 2013-06-16.
19. [^] "LZHAM - Lossless Data Compression Codec" [↗](#). Richard Geldreich. "LZHAM is a lossless data compression codec written in C/C++ with a compression ratio similar to LZMA but with 1.5x-8x faster decompression speed."

External links [edit]

- [Official home page](#)
- [Lzip format specification](#)
- [XZ format specification](#)
- [LZMA SDK \(Software Development Kit\)](#)
- [LZMA Utils = XZ Utils](#)
- [Windows Binaries for XZ Utils](#)

v t e	Archive formats	[show]
v t e	Data compression methods	[show]

Categories: [Lossless compression algorithms](#)

This page was last modified on 19 August 2015, at 10:22.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

