# Lossless compression

**Lossless compression** is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data. By contrast, lossy compression permits reconstruction only of an approximation of the original data, though this usually improves compression rates (and therefore reduces file sizes).

Lossless data compression is used in many applications. For example, it is used in the ZIP file format and in the GNU tool gzip. It is also often used as a component within lossy data compression technologies (e.g. lossless mid/side joint stereo preprocessing by MP3 encoders and other lossy audio encoders).

Lossless compression is used in cases where it is important that the original and the decompressed data be identical, or where deviations from the original data would be unfavourable. Typical examples are executable programs, text documents, and source code. Some image file formats, like PNG or GIF, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods. Lossless audio formats are most often used for archiving or production purposes, while smaller lossy audio files are typically used on portable players and in other cases where storage space is limited or exact replication of the audio is unnecessary.

## Contents



## Lossless compression techniques

Most lossless compression programs do two things in sequence: the first step generates a *statistical model* for the input data, and the second step uses this model to map input data to bit sequences in such a way that "probable" (e.g. frequently encountered) data will produce shorter output than "improbable" data.

The primary encoding algorithms used to produce bit sequences are Huffman coding (also used by DEFLATE) and arithmetic coding. Arithmetic coding achieves compression rates close to the best possible for a particular statistical model, which is given by the information entropy, whereas Huffman compression is simpler and faster but produces poor results for models that deal with symbol probabilities close to 1.

There are two primary ways of constructing statistical models: in a *static* model, the data is analyzed and a model is constructed, then this model is stored with the compressed data. This approach is simple and modular, but has the disadvantage that the model itself can be expensive to store, and also that it forces using a single model for all data being compressed, and so performs poorly on files that contain heterogeneous data. *Adaptive* models dynamically update the model as the data is compressed. Both the encoder and decoder begin with a trivial model, yielding poor compression of initial data, but as they learn more about the data, performance improves. Most popular types of compression used in practice now use adaptive coders.

Lossless compression methods may be categorized according to the type of data they are designed to compress. While, in principle, any general-purpose lossless compression algorithm (*general-purpose* meaning that they can accept any bitstring) can be used on any type of data, many are unable to achieve significant compression on data that are not of the form for which they were designed to compress. Many of the lossless compression techniques used for text also work reasonably well for indexed images.

## Multimedia

These techniques take advantage of the specific characteristics of images such as the common phenomenon of contiguous 2-D areas of similar tones. Every pixel but the first is replaced by the difference to its left neighbor. This leads to small values having a much higher probability than large values. This is often also applied to sound files, and can compress files that contain mostly low frequencies and low volumes. For images, this step can be repeated by taking the difference to the top pixel, and then in videos, the difference to the pixel in the next frame can be taken.

A hierarchical version of this technique takes neighboring pairs of data points, stores their difference and sum, and on a higher level with lower resolution continues with the sums. This is called discrete wavelet transform. JPEG2000 additionally uses data points from other pairs and multiplication factors to mix them into the difference. These factors must be integers, so that the result is an integer under all circumstances. So the values are increased, increasing file size, but hopefully the distribution of values is more peaked.

The adaptive encoding uses the probabilities from the previous sample in sound encoding, from the left and upper pixel in image encoding, and additionally from the previous frame in video encoding. In the wavelet transformation, the probabilities are also passed through the hierarchy.

## Historical legal issues

Many of these methods are implemented in open-source and proprietary tools, particularly LZW and its variants. Some algorithms are patented in the United States and other countries and their legal usage requires licensing by the patent holder. Because of patents on certain kinds of LZW compression, and in particular licensing practices by patent holder Unisys that many developers considered abusive, some open source proponents encouraged people to avoid using the Graphics Interchange Format (GIF) for compressing still image files in favor of Portable Network Graphics (PNG), which combines the LZ77-based deflate algorithm with a selection of domain-specific prediction filters. However, the patents on LZW expired on June 20, 2003.[1]

Many of the lossless compression techniques used for text also work reasonably well for indexed images, but there are other techniques that do not work for typical text that are useful for some images (particularly simple bitmaps), and other techniques that take advantage of the specific characteristics of images (such as the common phenomenon of contiguous 2-D areas of similar tones, and the fact that color images usually have a preponderance of a limited range of colors out of those representable in the color space)

As mentioned previously, lossless sound compression is a somewhat specialized area. Lossless sound compression algorithms can take advantage of the repeating patterns shown by the wave-like nature of the data – essentially using autoregressive models to predict the "next" value and encoding the (hopefully small) difference between the expected value and the actual data. If the

difference between the predicted and the actual data (called the *error*) tends to be small, then certain difference values (like 0, +1, −1 etc. on sample values) become very frequent, which can be exploited by encoding them in few output bits.

It is sometimes beneficial to compress only the differences between two versions of a file (or, in video compression, of successive images within a sequence). This is called delta encoding (from the Greek letter Δ, which in mathematics, denotes a difference), but the term is typically only used if both versions are meaningful outside compression and decompression. For example, while the process of compressing the error in the above-mentioned lossless audio compression scheme could be described as delta encoding from the approximated sound wave to the original sound wave, the approximated version of the sound wave is not meaningful in any other context.

# Lossless compression methods

By operation of the pigeonhole principle, no lossless compression algorithm can efficiently compress all possible data. For this reason, many different algorithms exist that are designed either with a specific type of input data in mind or with specific assumptions about what kinds of redundancy the uncompressed data are likely to contain.

Some of the most common lossless compression algorithms are listed below.

## General purpose

- Run-length encoding (RLE) – Simple scheme that provides good compression of data containing lots of runs of the same value
- Huffman coding – Pairs well with other algorithms, used by Unix's `pack` utility
- Prediction by partial matching (PPM) – Optimized for compressing plain text
- bzip2 – Combines Burrows–Wheeler transform with RLE and Huffman coding
- Lempel-Ziv compression (LZ77 and LZ78) – Dictionary-based algorithm that forms the basis for many other algorithms
  - DEFLATE – Combines Lempel-Ziv compression with Huffman coding, used by ZIP, gzip, and PNG images
  - Lempel–Ziv–Markov chain algorithm (LZMA) – Very high compression ratio, used by 7zip and xz
  - Lempel–Ziv–Oberhumer (LZO) – Designed for compression/decompression speed at the expense of compression ratios
  - Lempel–Ziv–Storer–Szymanski (LZSS) – Used by WinRAR in tandem with Huffman coding
  - Lempel–Ziv–Welch (LZW) – Used by GIF images and Unix's `compress` utility

## Audio

- Apple Lossless (ALAC - Apple Lossless Audio Codec)
- Adaptive Transform Acoustic Coding (ATRAC)
- apt-X Lossless
- Audio Lossless Coding (also known as MPEG-4 ALS)
- Direct Stream Transfer (DST)
- Dolby TrueHD
- DTS-HD Master Audio
- Free Lossless Audio Codec (FLAC)
- Meridian Lossless Packing (MLP)
- Monkey's Audio (Monkey's Audio APE)
- MPEG-4 SLS (also known as HD-AAC)
- OptimFROG
- Original Sound Quality (OSQ)
- RealPlayer (RealAudio Lossless)
- Shorten (SHN)
- TTA (True Audio Lossless)
- WavPack (WavPack lossless)

- WMA Lossless (Windows Media Lossless)

### Graphics

- PNG – Portable Network Graphics
- TIFF – Tagged Image File Format
- WebP – (high-density lossless or lossy compression of RGB and RGBA images)
- BPG – Better Portable Graphics (lossless/lossy compression based on HEVC)
- FLIF – Free Lossless Image Format
- JPEG-LS – (lossless/near-lossless compression standard)
- TGA - Truevision TGA
- PCX - PiCture eXchange
- JPEG 2000 – (includes lossless compression method, as proven by Sunil Kumar, Prof San Diego State University)
- JPEG XR – formerly *WMPhoto* and *HD Photo*, includes a lossless compression method
- ILBM – (lossless RLE compression of Amiga IFF images)
- JBIG2 – (lossless or lossy compression of B&W images)
- PGF – Progressive Graphics File (lossless or lossy compression)

### 3D Graphics

- OpenCTM – Lossless compression of 3D triangle meshes

### Video

See this list of lossless video codecs.

### Cryptography

Cryptosystems often compress data (the "plaintext") *before* encryption for added security. When properly implemented, compression greatly increases the unicity distance by removing patterns that might facilitate cryptanalysis. However, many ordinary lossless compression algorithms produce headers, wrappers, tables, or other predictable output that might instead make cryptanalysis easier. Thus, cryptosystems must utilize compression algorithms whose output does not contain these predictable patterns.

### Genetics

Genetics compression algorithms (not to be confused with genetic algorithms) are the latest generation of lossless algorithms that compress data (typically sequences of nucleotides) using both conventional compression algorithms and specific algorithms adapted to genetic data. In 2012, a team of scientists from Johns Hopkins University published the first genetic compression algorithm that does not rely on external genetic databases for compression. HAPZIPPER was tailored for HapMap data and achieves over 20-fold compression (95% reduction in file size), providing 2- to 4-fold better compression and in much faster time than the leading general-purpose compression utilities.[2]

### Executables

Self-extracting executables contain a compressed application and a decompressor. When executed, the decompressor transparently decompresses and runs the original application. This is especially often used in demo coding, where competitions are held for demos with strict size limits, as small as 1k. This type of compression is not strictly limited to binary executables, but can also be applied to scripts, such as JavaScript.

## Lossless compression benchmarks

Lossless compression algorithms and their implementations are routinely tested in head-to-head benchmarks. There are a number of better-known compression benchmarks. Some benchmarks cover only the data compression ratio, so winners in these benchmarks may be unsuitable for everyday use due to the slow speed of the top performers. Another drawback of some benchmarks is that their data files are known, so some program writers may optimize their programs for best performance on a particular data set. The winners on these benchmarks often come from the class of context-mixing compression software.

The benchmarks listed in the 5th edition of the *Handbook of Data Compression* (Springer, 2009) are:[3]

- The Maximum Compression benchmark, started in 2003 and updated until November 2011, includes over 150 programs. Maintained by Werner Bergmans, it tests on a variety of data sets, including text, images, and executable code. Two types of results are reported: single file compression (SFC) and multiple file compression (MFC). Not surprisingly, context mixing programs often win here; programs from the PAQ series and WinRK often are in the top. The site also has a list of pointers to other benchmarks.[4]
- UCLC (the ultimate command-line compressors) benchmark by Johan de Bock is another actively maintained benchmark including over 100 programs. The winners in most tests usually are PAQ programs and WinRK, with the exception of lossless audio encoding and grayscale image compression where some specialized algorithms shine.
- Squeeze Chart by Stephan Busch is another frequently updated site.
- The EmilCont benchmarks by Berto Destasio are somewhat outdated having been most recently updated in 2004. A distinctive feature is that the data set is not public, to prevent optimizations targeting it specifically. Nevertheless, the best ratio winners are again the PAQ family, SLIM and WinRK.
- The Archive Comparison Test (ACT) by Jeff Gilchrist included 162 DOS/Windows and 8 Macintosh lossless compression programs, but it was last updated in 2002.
- The Art Of Lossless Data Compression by Alexander Ratushnyak provides a similar test performed in 2003.

Matt Mahoney, in his February 2010 edition of the free booklet *Data Compression Explained*, additionally lists the following:[5]

- The Calgary Corpus dating back to 1987 is no longer widely used due to its small size. Matt Mahoney currently maintains the Calgary Compression Challenge[1], created and maintained from May 21, 1996 through May 21, 2016 by Leonid A. Broukhis[2].
- The Large Text Compression Benchmark[6] and the similar Hutter Prize both use a trimmed Wikipedia XML UTF-8 data set.
- The Generic Compression Benchmark[7], maintained by Mahoney himself, test compression on random data.
- Sami Runsas (author of NanoZip) maintains Compression Ratings, a benchmark similar to Maximum Compression multiple file test, but with minimum speed requirements. It also offers a calculator that allows the user to weight the importance of speed and compression ratio. The top programs here are fairly different due to speed requirement. In January 2010, the top programs were NanoZip followed by FreeArc, CCM, flashzip, and 7-Zip.
- The Monster of Compression benchmark by N. F. Antonio tests compression on 1Gb of public data with a 40-minute time limit. As of Dec. 20, 2009 the top ranked archiver is NanoZip 0.07a and the top ranked single file compressor is ccmx 1.30c, both context mixing.

The Compression Ratings website published a chart summary of the "frontier" in compression ratio and time.[8]

The Compression Analysis Tool[9] is a Windows application that enables end users to benchmark the performance characteristics of streaming implementations of LZF4, DEFLATE, ZLIB, GZIP, BZIP2 and LZMA using their own data. It produces measurements and charts with which users can compare the compression speed, decompression speed and compression ratio of the different compression methods and to examine how the compression level, buffer size and flushing operations affect the results.

The Squash Compression Benchmark uses the Squash library to compare more than 25 compression libraries in many different configurations using numerous different datasets on several different machines, and provides a web interface to help explore the results. There are currently over 50,000 results to compare.

# Limitations

Lossless data compression algorithms cannot guarantee compression for all input data sets. In other words, for any lossless data compression algorithm, there will be an input data set that does not get smaller when processed by the algorithm, and for any lossless data compression algorithm that makes at least one file smaller, there will be at least one file that it makes larger. This is easily proven with elementary mathematics using a counting argument, as follows:

- Assume that each file is represented as a string of bits of some arbitrary length.

- Suppose that there is a compression algorithm that transforms every file into an output file that is no longer than the original file, and that at least one file will be compressed into an output file that is shorter than the original file.
- Let $M$ be the least number such that there is a file $F$ with length $M$ bits that compresses to something shorter. Let $N$ be the length (in bits) of the compressed version of $F$.
- Because $N<M$, **every** file of length $N$ keeps its size during compression. There are $2^N$ such files. Together with $F$, this makes $2^N+1$ files that all compress into one of the $2^N$ files of length $N$.
- But $2^N$ is smaller than $2^N+1$, so by the pigeonhole principle there must be some file of length $N$ that is simultaneously the output of the compression function on two different inputs. That file cannot be decompressed reliably (which of the two originals should that yield?), which contradicts the assumption that the algorithm was lossless.
- We must therefore conclude that our original hypothesis (that the compression function makes no file longer) is necessarily untrue.

Any lossless compression algorithm that makes some files shorter must necessarily make some files longer, but it is not necessary that those files become *very much* longer. Most practical compression algorithms provide an "escape" facility that can turn off the normal coding for files that would become longer by being encoded. In theory, only a single additional bit is required to tell the decoder that the normal coding has been turned off for the entire input; however, most encoding algorithms use at least one full byte (and typically more than one) for this purpose. For example, DEFLATE compressed files never need to grow by more than 5 bytes per 65,535 bytes of input.

In fact, if we consider files of length N, if all files were equally probable, then for any lossless compression that reduces the size of some file, the expected length of a compressed file (averaged over all possible files of length N) must necessarily be *greater* than N. So if we know nothing about the properties of the data we are compressing, we might as well not compress it at all. A lossless compression algorithm is useful only when we are more likely to compress certain types of files than others; then the algorithm could be designed to compress those types of data better

Thus, the main lesson from the argument is not that one risks big losses, but merely that one cannot always win. To choose an algorithm always means implicitly to select a *subset* of all files that will become usefully shorter. This is the theoretical reason why we need to have different compression algorithms for different kinds of files: there cannot be any algorithm that is good for all kinds of data.

The "trick" that allows lossless compression algorithms, used on the type of data they were designed for, to consistently compress such files to a shorter form is that the files the algorithms are designed to act on all have some form of easily modeled redundancy that the algorithm is designed to remove, and thus belong to the subset of files that that algorithm can make shorter, whereas other files would not get compressed or even get bigger. Algorithms are generally quite specifically tuned to a particular type of file: for example, lossless audio compression programs do not work well on text files, and vice versa.

In particular, files of random data cannot be consistently compressed by any conceivable lossless data compression algorithm: indeed this result is used to *define* the concept of randomness in algorithmic complexity theory.

It's provably impossible to create an algorithm that can losslessly compress any data.[10] While there have been many claims through the years of companies achieving "perfect compression" where an arbitrary number $N$ of random bits can always be compressed to $N - 1$ bits, these kinds of claims can be safely discarded without even looking at any further details regarding the purported compression scheme. Such an algorithm contradicts fundamental laws of mathematics because, if it existed, it could be applied repeatedly to losslessly reduce any file to length 0. Allegedly "perfect" compression algorithms are often derisively referred to as "magic" compression algorithms for this reason.

On the other hand, it has also been proven that there is no algorithm to determine whether a file is incompressible in the sense of Kolmogorov complexity. Hence it's possible that any particular file, even if it appears random, may be significantly compressed, even including the size of the decompressor. An example is the digits of the mathematical constant *pi*, which appear random but can be generated by a very small program. However, even though it cannot be determined whether a particular file is incompressible, a simple theorem about incompressible strings shows that over 99% of files of any given length cannot be compressed by more than one byte (including the size of the decompressor).

## Mathematical background

Abstractly, a compression algorithm can be viewed as a function on sequences (normally of octets). Compression is successful if the resulting sequence is shorter than the original sequence (and the instructions for the decompression map). For a compression algorithm to be lossless, the compression map must form an injection from "plain" to "compressed" bit sequences.

The pigeonhole principle prohibits a bijection between the collection of sequences of length *N* and any subset of the collection of sequences of length *N*−1. Therefore, it is not possible to produce a lossless algorithm that reduces the size of every possible input sequence.

## Psychological background

Most everyday files are relatively 'sparse' in an information entropy sense, and thus, most lossless algorithms a layperson is likely to apply on regular files compress them relatively well. This may, through misapplication of intuition, lead some individuals to conclude that a well-designed compression algorithm can compress *any* input, thus, constituting a *magic compression algorithm*.

## Points of application in real compression theory

Real compression algorithm designers accept that streams of high information entropy cannot be compressed, and accordingly, include facilities for detecting and handling this condition. An obvious way of detection is applying a raw compression algorithm and testing if its output is smaller than its input. Sometimes, detection is made by heuristics; for example, a compression application may consider files whose names end in ".zip", ".arj" or ".lha" uncompressible without any more sophisticated detection. A common way of handling this situation is quoting input, or uncompressible parts of the input in the output, minimizing the compression overhead. For example, the zip data format specifies the 'compression method' of 'Stored' for input files that have been copied into the archive verbatim.[11]

## The Million Random Number Challenge

Mark Nelson, in response to claims of magic compression algorithms appearing in comp.compression, has constructed a 415,241 byte binary file of highly entropic content, and issued a public challenge of $100 to anyone to write a program that, together with its input, would be smaller than his provided binary data yet be able to reconstitute it without error.[12]

The FAQ for the comp.compression newsgroup contains a challenge by Mike Goldman offering $5,000 for a program that can compress random data. Patrick Craig took up the challenge, but rather than compressing the data, he split it up into separate files all of which ended in the number 5, which was not stored as part of the file. Omitting this character allowed the resulting files (plus, in accordance with the rules, the size of the program that reassembled them) to be smaller than the original file. However, no actual compression took place, and the information stored in the names of the files was necessary to reassemble them in the correct order in the original file, and this information was not taken into account in the file size comparison. The files themselves are thus not sufficient to reconstitute the original file; the file names are also necessary. Patrick Craig agreed that no meaningful compression had taken place, but argued that the wording of the challenge did not actually require this. A full history of the event, including discussion on whether or not the challenge was technically met, is on Patrick Craig's web site.[13]

# See also

- Comparison of file archivers
- Data compression
- David A. Huffman
- Entropy (information theory)
- Grammar induction
- Information theory
- Kolmogorov complexity
- List of codecs

- Lossless Transform Audio Compression (LTAC)
- Lossy compression
- Precompressor
- Universal code (data compression)
- Normal number

# References


1. Unisys | LZW Patent and Software Information (http://www.unisys.com/about__unisys/lzw) Archived (https://web.archive.org/web/20090602212118/http://www.unisys.com/about__unisys/lzw) 2009-06-02 at the Wayback Machine
2. Chanda, Elhaik, and Bader (2012). "HapZipper: sharing HapMap populations just got easier" (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3488212) *Nucleic Acids Res.* **40** (20): 1–7. PMC 3488212 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3488212). PMID 22844100 (https://www.ncbi.nlm.nih.gov/pubmed/22844100). doi:10.1093/nar/gks709 (https://doi.org/10.1093%2Fnar%2Fgks709)
3. David Salomon, Giovanni Motta, (with contributions by David Bryant), *Handbook of Data Compression*, 5th edition, Springer, 2009, ISBN 1-84882-902-7, pp. 16–18.
4. Lossless Data Compression Benchmarks (links and spreadsheets) (http://www.maximumcompression.com/benchmarks/benchmarks.php)
5. Matt Mahoney (2010). "Data Compression Explained" (http://nishi.dreamhosters.com/u/dce2010-02-26.pdf) (PDF). pp. 3–5.
6. http://mattmahoney.net/dc/text.html
7. http://mattmahoney.net/dc/uiq/
8. Visualization of compression ratio and time (https://web.archive.org/web/20160831012928/https://compressionratings.com/rating_sum.html)
9. https://www.noemax.com/free-tools/compression-analysis-tool.asp
10. comp.compression FAQ list entry #9: Compression of random data (WEB, Gilbert and others) (http://www.faqs.org/faqs/compression-faq/part1/section-8.html)
11. ZIP file format specification (http://www.pkware.com/documents/casestudies/APPNOTE.TXT) by PKWARE, Inc., chapter V, section J
12. Nelson, Mark (2006-06-20). "The Million Random Digit Challenge Revisited" (http://marknelson.us/2006/06/20/million-digit-challenge/)
13. Craig, Patrick. "The $5000 Compression Challenge" (http://www.patrickcraig.co.uk/other/compression.htm). Retrieved 2009-06-08.


# External links


- "Lossless comparison". *Hydrogenaudio Knowledgebase*. 2015-01-05. Retrieved 2017-10-17.
- "Lossless and lossy audio formats for music.". *Bobulous Central*. 2003-11-06. Retrieved 2017-10-17.
- Phamdo, Nam. "Theory of Data Compression". *Data Compression*. Retrieved 2017-10-17.
- "comp.compression Frequently Asked Questions (part 2/3) Section - [73] What is the theoretical compression limit?" *faqs.org*. 2014-03-27. Retrieved 2017-10-17.
- "Image Compression Benchmark". *Image Compression Benchmark*. Retrieved 2017-10-17. overview of
  - US patent #7,096,360, "[a]n "Frequency-Time Based Data Compression Method" supporting the compression, encryption, decompression, and decryption and persistence of many binary digits through frequencies where each frequency represents many bits."
- "LZF compression format". *github*. Retrieved 2017-10-17.