**gzip** is a [software application](http://en.wikipedia.org/wiki/Software_application) used for [file compression and decompression](http://en.wikipedia.org/wiki/Data_compression). The program was created by [Jean-Loup Gailly](http://en.wikipedia.org/wiki/Jean-Loup_Gailly) and [Mark Adler](http://en.wikipedia.org/wiki/Mark_Adler) as a [free software](http://en.wikipedia.org/wiki/Free_software) replacement for the [compress](http://en.wikipedia.org/wiki/Compress) program used in early [Unix](http://en.wikipedia.org/wiki/Unix) systems, and intended for use by the [GNU Project](http://en.wikipedia.org/wiki/GNU_Project) (the "g" is from "GNU"). Version 0.1 was first publicly released on 31 October 1992, and version 1.0 followed in February 1993.

**Contents**

 [[hide](http://en.wikipedia.org/wiki/Gzip)]

* [1 File format](http://en.wikipedia.org/wiki/Gzip#File_format)
* [2 Implementations](http://en.wikipedia.org/wiki/Gzip#Implementations)
* [3 Derivatives and other uses](http://en.wikipedia.org/wiki/Gzip#Derivatives_and_other_uses)
* [4 See also](http://en.wikipedia.org/wiki/Gzip#See_also)
* [5 Notes](http://en.wikipedia.org/wiki/Gzip#Notes)
* [6 References](http://en.wikipedia.org/wiki/Gzip#References)
* [7 External links](http://en.wikipedia.org/wiki/Gzip#External_links)

**File format[[edit](http://en.wikipedia.org/w/index.php?title=Gzip&action=edit&section=1" \o "Edit section: File format)]**

|  |  |
| --- | --- |
| **gzip** | |
| [**Filename extension**](http://en.wikipedia.org/wiki/Filename_extension) | .gz |
| [**Internet media type**](http://en.wikipedia.org/wiki/Internet_media_type) | application/gzip[[2]](http://en.wikipedia.org/wiki/Gzip" \l "cite_note-2) |
| [**Uniform Type Identifier**](http://en.wikipedia.org/wiki/Uniform_Type_Identifier) | org.gnu.gnu-zip-archive |
| **Developed by** | Jean-Loup Gailly and Mark Adler |
| **Type of format** | [data compression](http://en.wikipedia.org/wiki/Data_compression) |
| [**Open format**](http://en.wikipedia.org/wiki/Open_format)**?** | Yes |

gzip is based on the [DEFLATE](http://en.wikipedia.org/wiki/DEFLATE) algorithm, which is a combination of [LZ77](http://en.wikipedia.org/wiki/LZ77_and_LZ78) and [Huffman coding](http://en.wikipedia.org/wiki/Huffman_coding). DEFLATE was intended as a replacement for [LZW](http://en.wikipedia.org/wiki/LZW) and other [patent](http://en.wikipedia.org/wiki/Patent)-encumbered [data compression](http://en.wikipedia.org/wiki/Data_compression) [algorithms](http://en.wikipedia.org/wiki/Algorithm) which, at the time, limited the usability of compress and other popular archivers.

"gzip" is often also used to refer to the gzip file format, which is:

* a 10-byte header, containing a [magic number](http://en.wikipedia.org/wiki/Magic_number_(programming)) (1f 8b), a version number and a timestamp
* optional extra headers, such as the original file name,
* a body, containing a DEFLATE-compressed [payload](http://en.wikipedia.org/wiki/Payload_(communication_and_information_technology))
* an 8-byte footer, containing a [CRC-32](http://en.wikipedia.org/wiki/CRC-32) checksum and the length of the original uncompressed data.

Although its file format also allows for multiple such streams to be [concatenated](http://en.wikipedia.org/wiki/Concatenation) (zipped files are simply decompressed concatenated as if they were originally one file[[3]](http://en.wikipedia.org/wiki/Gzip" \l "cite_note-3)), gzip is normally used to compress just single files.[[4]](http://en.wikipedia.org/wiki/Gzip#cite_note-4) Compressed archives are typically created by assembling collections of files into a single [tar](http://en.wikipedia.org/wiki/Tar_(file_format)) archive, and then compressing that archive with gzip. The final .tar.gz or .tgz file is usually called a [tarball](http://en.wikipedia.org/wiki/Tar_(file_format)).[[5]](http://en.wikipedia.org/wiki/Gzip#cite_note-5)

gzip is not to be confused with the [ZIP](http://en.wikipedia.org/wiki/ZIP_(file_format)) archive format, which also uses DEFLATE. The ZIP format can hold collections of files without an external archiver, but is less compact than compressed [tarballs](http://en.wikipedia.org/wiki/Tar_(file_format)) holding the same data, because it compresses files individually and cannot take advantage of redundancy between files ([solid compression](http://en.wikipedia.org/wiki/Solid_compression)).

**Implementations[[edit](http://en.wikipedia.org/w/index.php?title=Gzip&action=edit&section=2" \o "Edit section: Implementations)]**

|  |  |
| --- | --- |
| NetBSD Gzip / FreeBSD Gzip | |
| [**Developer(s)**](http://en.wikipedia.org/wiki/Software_developer) | The NetBSD Foundation |
| **Written in** | [C](http://en.wikipedia.org/wiki/C_(programming_language)) |
| [**Operating system**](http://en.wikipedia.org/wiki/Operating_system) | [Cross-platform](http://en.wikipedia.org/wiki/Cross-platform) |
| [**Type**](http://en.wikipedia.org/wiki/List_of_software_categories) | [data compression](http://en.wikipedia.org/wiki/Data_compression) |
| [**License**](http://en.wikipedia.org/wiki/Software_license) | [Simplified BSD License](http://en.wikipedia.org/wiki/BSD_licenses) |

Various implementations of the program have been written. The most commonly known is the GNU Project's implementation using [Lempel-Ziv](http://en.wikipedia.org/wiki/LZ77) coding (LZ77). [OpenBSD](http://en.wikipedia.org/wiki/OpenBSD)'s version of gzip is actually the compress program, to which support for the gzip format was added in OpenBSD 3.4. The 'g' in this specific version stands for [*gratis*](http://en.wiktionary.org/wiki/gratis).[[6]](http://en.wikipedia.org/wiki/Gzip#cite_note-6) [FreeBSD](http://en.wikipedia.org/wiki/FreeBSD), [DragonFlyBSD](http://en.wikipedia.org/wiki/DragonFlyBSD) and [NetBSD](http://en.wikipedia.org/wiki/NetBSD) use a BSD-licensed implementation instead of the GNU version; it is actually a [command-line interface](http://en.wikipedia.org/wiki/Command-line_interface) for [zlib](http://en.wikipedia.org/wiki/Zlib) intended to be compatible with the GNU implementation's options.[[7]](http://en.wikipedia.org/wiki/Gzip#cite_note-7) These implementations originally come from [NetBSD](http://en.wikipedia.org/wiki/NetBSD), and supports decompression of [bzip2](http://en.wikipedia.org/wiki/Bzip2) and the Unix pack(1) format.

**Derivatives and other uses[[edit](http://en.wikipedia.org/w/index.php?title=Gzip&action=edit&section=3" \o "Edit section: Derivatives and other uses)]**

When gzip is invoked as gunzip, it decompresses the data (a file or stdin). gunzip is equivalent to gzip -d.

When gzip is invoked as zcat, it also decompresses the data, but behaves similarly to [cat](http://en.wikipedia.org/wiki/Cat_(Unix)). It decompresses individual files and [concatenates](http://en.wikipedia.org/wiki/Concatenate) them to standard output. zcat is equivalent to gzip -d -c.[[8]](http://en.wikipedia.org/wiki/Gzip#cite_note-man-8)

[zlib](http://en.wikipedia.org/wiki/Zlib) is an abstraction of the DEFLATE algorithm in library form which includes support both for the gzip file format and a lightweight stream format in its API. The zlib stream format, DEFLATE, and the gzip file format were standardized respectively as [RFC 1950](http://tools.ietf.org/html/rfc1950), [RFC 1951](http://tools.ietf.org/html/rfc1951), and [RFC 1952](http://tools.ietf.org/html/rfc1952).

The "Content-Encoding"/"Accept-Encoding" and "Transfer-Encoding"/"TE" headers in [HTTP](http://en.wikipedia.org/wiki/HTTP)/1.1 allow clients to optionally receive compressed HTTP responses and (less commonly) to send compressed requests. The specification for HTTP/1.1 ([RFC 2616](http://tools.ietf.org/html/rfc2616)) specifies three compression methods: "gzip" ([RFC 1952](http://tools.ietf.org/html/rfc1952); the content wrapped in a gzip stream), "deflate" ([RFC 1950](http://tools.ietf.org/html/rfc1950); the content wrapped in a zlib-formatted stream), and "compress" (explained in [RFC 2616](http://tools.ietf.org/html/rfc2616) section 3.5 as "*The encoding format produced by the common UNIX file compression program compress. This format is an adaptive Lempel-Ziv-Welch coding (LZW).*"). Many client libraries, browsers, and server platforms (including [Apache](http://en.wikipedia.org/wiki/Apache) and [Microsoft IIS](http://en.wikipedia.org/wiki/Microsoft_IIS)) support gzip. Many agents also support deflate, although several important players incorrectly implement deflate support using the format specified by [RFC 1951](http://tools.ietf.org/html/rfc1951) instead of the correct format specified by [RFC 1950](http://tools.ietf.org/html/rfc1950) (which encapsulates [RFC 1951](http://tools.ietf.org/html/rfc1951)). Notably, Internet Explorer versions 6, 7, and 8 report deflate support but do not actually accept [RFC 1950](http://tools.ietf.org/html/rfc1950) format, making actual use of deflate highly unusual. Many clients accept both [RFC 1951](http://tools.ietf.org/html/rfc1951) and [RFC 1950](http://tools.ietf.org/html/rfc1950)-formatted data for the "deflate" compressed method, but a server has no way to detect whether a client will correctly handle [RFC 1950](http://tools.ietf.org/html/rfc1950) format.

Since the late 1990s, [bzip2](http://en.wikipedia.org/wiki/Bzip2), a file compression utility based on a block-sorting algorithm, has gained some popularity as a gzip replacement. It produces considerably smaller files (especially for source code and other structured text), but at the cost of memory and processing time (up to a factor of 4)[*[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*]. bzip2-compressed tarballs are conventionally named either .tar.bz2 or simply .tbz.

[AdvanceCOMP](http://en.wikipedia.org/wiki/AdvanceCOMP) and [7-Zip](http://en.wikipedia.org/wiki/7-Zip) can produce gzip-compatible files, using an internal DEFLATE implementation with better compression ratios than gzip itself—at the cost of more processor time compared to the reference implementation.

**See also[[edit](http://en.wikipedia.org/w/index.php?title=Gzip&action=edit&section=4" \o "Edit section: See also)]**

What is gzip?

=============

Gzip reduces the size of the named files using Lempel-Ziv coding (LZ77)

How does it work?

=================

1. Compression algorithm (deflate)

The deflation algorithm used by gzip (also zip and zlib) is a variation of

LZ77 (Lempel-Ziv 1977, see reference below). It finds duplicated strings in

the input data. The second occurrence of a string is replaced by a

pointer to the previous string, in the form of a pair (distance,

length). Distances are limited to 32K bytes, and lengths are limited

to 258 bytes. When a string does not occur anywhere in the previous

32K bytes, it is emitted as a sequence of literal bytes. (In this

description, `string' must be taken as an arbitrary sequence of bytes,

and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and

match distances are compressed with another tree. The trees are stored

in a compact form at the start of each block. The blocks can have any

size (except that the compressed data for one block must fit in

available memory). A block is terminated when deflate() determines that

it would be useful to start another block with fresh trees. (This is

somewhat similar to the behavior of LZW-based \_compress\_.)

Duplicated strings are found using a hash table. All input strings of

length 3 are inserted in the hash table. A hash index is computed for

the next 3 bytes. If the hash chain for this index is not empty, all

strings in the chain are compared with the current input string, and

the longest match is selected.

The hash chains are searched starting with the most recent strings, to

favor small distances and thus take advantage of the Huffman encoding.

The hash chains are singly linked. There are no deletions from the

hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily

truncated at a certain length, determined by a runtime option (level

parameter of deflateInit). So deflate() does not always find the longest

possible match but generally finds a match which is long enough.

deflate() also defers the selection of matches with a lazy evaluation

mechanism. After a match of length N has been found, deflate() searches for

a longer match at the next input byte. If a longer match is found, the

previous match is truncated to a length of one (thus producing a single

literal byte) and the process of lazy evaluation begins again. Otherwise,

the original match is kept, and the next match search is attempted only N

steps later.

The lazy match evaluation is also subject to a runtime parameter. If

the current match is long enough, deflate() reduces the search for a longer

match, thus speeding up the whole process. If compression ratio is more

important than speed, deflate() attempts a complete second search even if

the first match is already long enough.

The lazy match evaluation is not performed for the fastest compression

modes (level parameter 1 to 3). For these fast modes, new strings

are inserted in the hash table only when no match was found, or

when the match is not too long. This degrades the compression ratio

but saves time since there are both fewer insertions and fewer searches.

2. Decompression algorithm (inflate)

2.1 Introduction

The real question is, given a Huffman tree, how to decode fast. The most

important realization is that shorter codes are much more common than

longer codes, so pay attention to decoding the short codes fast, and let

the long codes take longer to decode.

inflate() sets up a first level table that covers some number of bits of

input less than the length of longest code. It gets that many bits from the

stream, and looks it up in the table. The table will tell if the next

code is that many bits or less and how many, and if it is, it will tell

the value, else it will point to the next level table for which inflate()

grabs more bits and tries to decode a longer code.

How many bits to make the first lookup is a tradeoff between the time it

takes to decode and the time it takes to build the table. If building the

table took no time (and if you had infinite memory), then there would only

be a first level table to cover all the way to the longest code. However,

building the table ends up taking a lot longer for more bits since short

codes are replicated many times in such a table. What inflate() does is

simply to make the number of bits in the first table a variable, and set it

for the maximum speed.

inflate() sends new trees relatively often, so it is possibly set for a

smaller first level table than an application that has only one tree for

all the data. For inflate, which has 286 possible codes for the

literal/length tree, the size of the first table is nine bits. Also the

distance trees have 30 possible values, and the size of the first table is

six bits. Note that for each of those cases, the table ended up one bit

longer than the ``average'' code length, i.e. the code length of an

approximately flat code which would be a little more than eight bits for

286 symbols and a little less than five bits for 30 symbols. It would be

interesting to see if optimizing the first level table for other

applications gave values within a bit or two of the flat code size.

2.2 More details on the inflate table lookup

Ok, you want to know what this cleverly obfuscated inflate tree actually

looks like. You are correct that it's not a Huffman tree. It is simply a

lookup table for the first, let's say, nine bits of a Huffman symbol. The

symbol could be as short as one bit or as long as 15 bits. If a particular

symbol is shorter than nine bits, then that symbol's translation is duplicated

in all those entries that start with that symbol's bits. For example, if the

symbol is four bits, then it's duplicated 32 times in a nine-bit table. If a

symbol is nine bits long, it appears in the table once.

If the symbol is longer than nine bits, then that entry in the table points

to another similar table for the remaining bits. Again, there are duplicated

entries as needed. The idea is that most of the time the symbol will be short

and there will only be one table look up. (That's whole idea behind data

compression in the first place.) For the less frequent long symbols, there

will be two lookups. If you had a compression method with really long

symbols, you could have as many levels of lookups as is efficient. For

inflate, two is enough.

So a table entry either points to another table (in which case nine bits in

the above example are gobbled), or it contains the translation for the symbol

and the number of bits to gobble. Then you start again with the next

ungobbled bit.

You may wonder: why not just have one lookup table for how ever many bits the

longest symbol is? The reason is that if you do that, you end up spending

more time filling in duplicate symbol entries than you do actually decoding.

At least for deflate's output that generates new trees every several 10's of

kbytes. You can imagine that filling in a 2^15 entry table for a 15-bit code

would take too long if you're only decoding several thousand symbols. At the

other extreme, you could make a new table for every bit in the code. In fact,

that's essentially a Huffman tree. But then you spend two much time

traversing the tree while decoding, even for short symbols.

So the number of bits for the first lookup table is a trade of the time to

fill out the table vs. the time spent looking at the second level and above of

the table.

Here is an example, scaled down:

The code being decoded, with 10 symbols, from 1 to 6 bits long: