

Made by: - Guided by: -

Harshit Verma (768948) Mr. Pankaj Anadure

Ritik Arora (768981)

Rohit Singh Bisht (768983)

Table of Contents

[PROBLEM STATEMENT 4](#_Toc7705704)

[APPROACH 4](#_Toc7705705)

[1. INTRODUCTION 5](#_Toc7705706)

[1.1 What is data compression? 5](#_Toc7705707)

[1.2 How compression Works? 5](#_Toc7705708)

[1.3 Why is data compression important? 6](#_Toc7705709)

[2. Data compression methods 7](#_Toc7705710)

[2.1 Lossless Compression 8](#_Toc7705711)

[Lossless Compression Algorithms 8](#_Toc7705712)

[2.1.1. Run length encoding 9](#_Toc7705713)

[2.1.2. Differential pulse code modulation 9](#_Toc7705714)

[2.1.3. Dictionary based encoding 10](#_Toc7705715)

[2.2 Lossy Compression 11](#_Toc7705716)

[2.2.1 Audio Compression 12](#_Toc7705717)

[2.2.2 MP3 14](#_Toc7705718)

[2.3 Difference between Lossless and Lossy 15](#_Toc7705719)

[3. Compression vs. data deduplication 16](#_Toc7705720)

[3.1 Data compression 16](#_Toc7705721)

[3.2 Data deduplication 17](#_Toc7705722)

[3.3 Encryption 18](#_Toc7705723)

[3.4 Understanding how they work together 19](#_Toc7705724)

[3.5 How they diverge 20](#_Toc7705725)

[3.6 How these three intersect 20](#_Toc7705726)

[3.7 How deduplication and compression work with other technologies 21](#_Toc7705727)

[4. Data compression and backup 22](#_Toc7705728)

[5. Pros and cons of compression 23](#_Toc7705729)

[5.1 When Compression Is Good 23](#_Toc7705730)

[5.2 When Compression Is Bad 24](#_Toc7705731)

[5.3 Inline versus Offline 25](#_Toc7705732)

[5.4 Hidden Compression 25](#_Toc7705733)

[5.5 Should I Turn On Inline Compression? 26](#_Toc7705734)

[6. GZIP Compression 27](#_Toc7705735)

[6.1 File Format 28](#_Toc7705736)

[6.2 Implementation 30](#_Toc7705737)

[6.3 Algorithm 31](#_Toc7705738)

[6.3.1 Huffman coding 31](#_Toc7705739)

[6.3.2 LZ77 compression 35](#_Toc7705740)

[6.3.3 Putting it all together 38](#_Toc7705741)

[7. BZip2 Compression 40](#_Toc7705742)

[7.1 File Format 41](#_Toc7705743)

[7.2 Implementation 42](#_Toc7705744)

[7.3 Algorithm 43](#_Toc7705745)

[7.3.1 Initial run-length encoding 44](#_Toc7705746)

[7.3.2 Burrows–Wheeler transform 44](#_Toc7705747)

[7.3.2 Move to front transform 45](#_Toc7705748)

[7.3.4 Run-length encoding of MTF result 46](#_Toc7705749)

[7.3.5 Multiple Huffman tables 47](#_Toc7705750)

[7.3.6 Unary encoding of Huffman table selection 47](#_Toc7705751)

[7.3.7 Delta encoding 48](#_Toc7705752)

[7.3.8 Sparse bit array 48](#_Toc7705753)

[8. Difference between GZIP and BZip2 50](#_Toc7705754)

[8.1 Compression 50](#_Toc7705755)

[8.2 Decompression 51](#_Toc7705756)

[8.3 So what is the best? 52](#_Toc7705757)

# PROBLEM STATEMENT

Develop an application which would use 2 different compression techniques to send and receive the contents (It can be files, API responses etc). Compare the compression techniques and show the report on how much time it has taken to compress, send or receive a particular content. This needs both sender and receiver applications.

# APPROACH

In order to solve the problem asked we have developed Server-Client Application using Angular6 on front-end and python (flask) on backend. Client application is used to receive the compressed file and decompress it. Server application is used to send the file to the client after compressing the original file using two different techniques.

# INTRODUCTION

## 1.1 What is data compression?

Data compression is a reduction in the number of [bits](https://whatis.techtarget.com/definition/bit-binary-digit) needed to represent data. Compressing data can save storage capacity, speed up file transfer, and decrease costs for storage hardware and network [bandwidth](https://searchnetworking.techtarget.com/definition/bandwidth).

## How compression Works?

Compression is performed by a program that uses a formula or [algorithm](https://whatis.techtarget.com/definition/algorithm) to determine how to shrink the size of the data. For instance, an algorithm may represent a string of bits -- or 0s and 1s -- with a smaller string of 0s and 1s by using a dictionary for the conversion between them, or the formula may insert a reference or pointer to a string of 0s and 1s that the program has already seen.

Text compression can be as simple as removing all unneeded [characters](https://whatis.techtarget.com/definition/character), inserting a single repeat character to indicate a string of repeated characters and substituting a smaller bit string for a frequently occurring bit string. Data compression can reduce a text file to 50% or a significantly higher percentage of its original size.

For data transmission, compression can be performed on the data content or on the entire transmission unit, including [header](https://whatis.techtarget.com/definition/header) data. When information is sent or received via the internet, larger files, either singly or with others as part of an [archive](https://searchstorage.techtarget.com/definition/archive) file, may be transmitted in a ZIP, [GZIP](https://searchdatacenter.techtarget.com/definition/gzip-GNU-zip) or other compressed format.

## Why is data compression important?

Data compression can dramatically decrease the amount of storage a file takes up. For example, in a 2:1 compression ratio, a 20 megabyte ([MB](https://searchstorage.techtarget.com/definition/megabyte)) file takes up 10 MB of space. As a result of compression, administrators spend less money and less time on storage.

Compression optimizes backup storage performance and has recently shown up in [primary storage data reduction](https://searchstorage.techtarget.com/definition/data-reduction-in-primary-storage-DRIPS). Compression will be an important method of data reduction as data continues to grow exponentially.

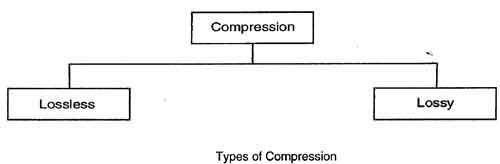
Virtually any type of file can be compressed, but it's important to follow best practices when choosing which ones to compress. For example, some files may already come compressed, so compressing those files would not have a significant impact.

# Data compression methods

Data compression is the function of presentation layer in OSI reference model. Compression is often used to maximize the use of bandwidth across a network or to optimize disk space when saving data.

There are two general types of compression algorithms:

* Lossless compression
* Lossy compression

[](http://ecomputernotes.com/images/Types-of-Compression.jpg)

## **2.1 Lossless Compression**

Lossless compression compresses the data in such a way that when data is decompressed it is exactly the same as it was before compression *i.e.*there is no loss of data.

A lossless compression is used to compress file data such as executable code, text files, and numeric data, because programs that process such file data cannot tolerate mistakes in the data.

Lossless compression will typically not compress file as much as lossy compression techniques and may take more processing power to accomplish the compression.

### Lossless Compression Algorithms

The various algorithms used to implement lossless data compression are:

* Run length encoding
* Differential pulse code modulation
* Dictionary based encoding

### **2.1.1. Run length encoding**

• This method replaces the consecutive occurrences of a given symbol with only one copy of the symbol along with a count of how many times that symbol occurs. Hence the names ‘run length'.

• For example, the string AAABBCDDDD would be encoded as 3A2BIC4D.

• A real life example where run-length encoding is quite effective is the fax machine. Most faxes are white sheets with the occasional black text. So, a run-length encoding scheme can take each line and transmit a code for while then the number of pixels, then the code for black and the number of pixels and so on.

• This method of compression must be used carefully. If there is not a lot of repetition in the data then it is possible the run length encoding scheme would actually increase the size of a file.

### **2.1.2. Differential pulse code modulation**

• In this method first a reference symbol is placed. Then for each symbol in the data, we place the difference between that symbol and the reference symbol used.

• For example, using symbol A as reference symbol, the string AAABBC DDDD would be encoded as AOOOl123333, since A is the same as reference symbol, B has a difference of 1 from the reference symbol and so on.

### **2.1.3. Dictionary based encoding**

• One of the best known dictionary based encoding algorithms is Lempel-Ziv (LZ) compression algorithm.

• This method is also known as substitution coder.

• In this method, a dictionary (table) of variable length strings (common phrases) is built.

• This dictionary contains almost every string that is expected to occur in data.

• When any of these strings occur in the data, then they are replaced with the corresponding index to the dictionary.

• In this method, instead of working with individual characters in text data, we treat each word as a string and output the index in the dictionary for that word.

• For example, let us say that the word "compression" has the index 4978 in one particular dictionary; it is the 4978th word is usr/share/dict/words. To compress a body of text, each time the string "compression" appears, it would be replaced by 4978.

## **2.2 Lossy Compression**

Lossy compression is the one that does not promise that the data received is exactly the same as data send *i.e.*the data may be lost.

This is because a lossy algorithm removes information that it cannot later restore.

Lossy algorithms are used to compress still images, video and audio.

Lossy algorithms typically achieve much better compression ratios than the lossless algorithms.

### 2.2.1 Audio Compression [Techniques of Audio Compression](http://ecomputernotes.com/images/Techniques-of-Audio-compression.jpg)

• Audio compression is used for speech or music.

• For speech, we need to compress a 64-KHz digitized signal; for music, we need to compress a 1.411.MHz signal

 • Two types of techniques are used for audio compression:

* Predictive encoding
* Perceptual encoding

#### **2.2.1.1 Predictive encoding**

• In predictive encoding, the differences between the samples are encoded instead of encoding all the sampled values.

• This type of compression is normally used for speech.

• Several standards have been defined such as GSM (13 kbps), G. 729 (8 kbps), and G.723.3 (6.4 or 5.3 kbps).

#### **2.2.1.2 Perceptual encoding**

• Perceptual encoding scheme is used to create a CD-quality audio that requires a transmission bandwidth of 1.411 Mbps.

• MP3 (MPEG audio layer 3), a part of MPEG standard uses this perceptual encoding.

• Perceptual encoding is based on the science of psychoacoustics, a study of how people perceive sound.

• The perceptual encoding exploits certain flaws in the human auditory system to encode a signal in such a way that it sounds the same to a human listener, even if it looks quite different on an oscilloscope.

• The key property of perceptual coding is that some sounds can mask other sound. For example, imagine that you are broadcasting a live flute concert and all of a sudden someone starts striking a hammer on a metal sheet. You will not be able to hear the flute any more. Its sound has been masked by the hammer.

• Such a technique explained above is called frequency masking-the ability of a loud sound in one frequency band to hide a softer sound in another frequency band that would have been audible in the absence of the loud sound.

• Masking can also be done on the basis of time. For example: Even if the hammer is not striking on a metal sheet, the flute will be inaudible for a short period of time because the ears turn down its gain when they start and take a finite time to turn up again.

• Thus, a loud sound can numb our ears for a short time even after the sound has stopped. This effect is called temporal masking.

### 2.2.2 MP3

• MP3 uses these two phenomena, *i.e.*frequency masking and temporal masking to compress audio signals.

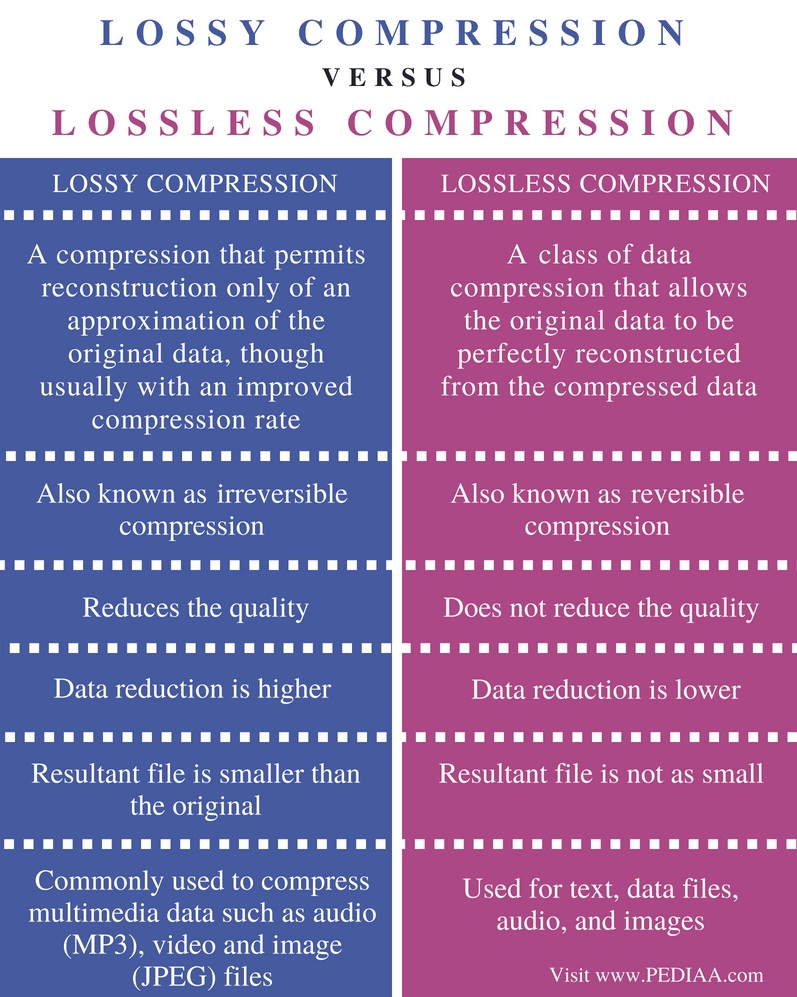
• In such a system, the technique analyzes and divides the spectrum into several groups. Zero bits are allocated to the frequency ranges that are totally masked.

• A small number of bits are allocated to the frequency ranges that are partially masked.

• A larger number. of bits are allocated to the frequency ranges that are not masked.

• Based on the range of frequencies in the original analog audio, MP3 produces three data rates: 96kbps, 128 kbps and 160 kbps.

## Difference between Lossless and Lossy



# 3. Compression vs. data deduplication

As more corporate data is relegated to spinning disk, storage administrators must implement, configure and manage this escalating capacity -- stretching disk space to the limit, while protecting important data against loss or theft.

Compression, deduplication and encryption are common data protection technologies for managing and optimizing disk storage, and it's important to understand the role each one plays in the data center.

## 3.1 Data compression

Compression attempts to reduce the size of a file by removing redundant data within the file. By making files smaller, less disk space is consumed, and more files can be stored on disk. For example, a 100 KB text file might be compressed to 52 KB by removing extra spaces or replacing long character strings with short representations. An algorithm recreates the original data when the file is read. Picture files are also usually compressed. For example, the JPEG image file format uses compression to eliminate redundant pixel data.

Almost any file can be compressed, though files with nonredundant data may compress little, if at all, so compression ratios are a guideline and not a rule. For example, a 2-to-1 compression ratio can ideally allow 400 GB worth of files on a 200 GB disk. It's difficult to determine exactly how much a file can be compressed until a compression algorithm is applied.

## 3.2 Data deduplication

A typical data center may store many copies of the same file. File deduplication -- sometimes called data reductionor single-instance storage -- is another space-saving data protection technology intended to eliminate redundant files on a storage system. By saving only one instance of a file, disk space can be significantly reduced.

For example, suppose the same 10 MB PowerPoint presentation is stored in 10 folders for each sales associate or department. That's 100 MB of disk space consumed to maintain the same 10 MB file. File deduplication ensures that only one complete copy is saved to disk. Subsequent iterations of the file are only saved as references that point to the saved copy, so users still see their own files in place. Similarly, a storage system may retain 200 emails, each with a 1 MB attachment. With deduplication, the 200 MB needed to store each 1 MB attachment is reduced to just 1 MB for one iteration of the file.

Block-level deduplication saves unique iterations of each block in a file. If a user updates a file, only the revised data is saved. Because the changes don't establish a completely new file, block-level deduplication is more efficient than file-level dedupe. Block-level deduplication, though, requires more processing power and uses a bigger index to track individual pieces.

Deduplication can also provide more granular control, removing redundant portions of files, potentially down to the block level. When evaluating a deduplication product, it's important to understand the granularity offered by the vendor's platform.

Vendors frequently tout their deduplication ratio. As an example, a 10-to-1 deduplication ratio means that 10 times more data is protected than the physical space needed to store it.

## 3.3 Encryption

With an increase in government regulations and corporate litigation, data storage managers have to pay close attention to the role of security in enterprise storage. Encryption is a key data protection technology that prevents unauthorized users from accessing information, even if files are hacked and stolen.

Encryption uses a mathematical algorithm with a unique key to encode a file into a form that cannot be read. No one else can access or use the encrypted file until it is decrypted using the identical key. If the encryption key is lost, any data encrypted with that key will be inaccessible.

Encryption is a key element of data protection products. When moving data from one location to another, it's best to encrypt it at the original site, while it's in transit and when it's at rest in the new spot. It's important to properly encrypt backups with a good algorithm and solid key management.

End-to-end encryption prevents data sent between two locations from being viewed by anyone who intercepts the communication channel. Text messaging applications, such as Apple iMessage, often use end-to-end encryption, but law enforcement agencies have criticized vendors because the technology prevents them from accessing information.

## 3.4 Understanding how they work together

Used together, the three technologies enable increased capacity and protection. A purpose-built backup appliance, for example, will typically incorporate deduplication, compression and encryption, with objectives that include protecting data from breaches and removing redundant data.

Deduplication is often associated with compression, and many storage systems support both data protection technologies because they optimize storage capacity and enable longer retention periods, lower bandwidth consumption and faster recoveries. An organization can use the two together, for example, by deduplicating a storage environment and then compressing the files.

## 3.5 How they diverge

The data protection technologies are similar, but they operate differently.

Deduplication looks for redundant pieces of data, while compression uses an algorithm to reduce the bits required to represent data. Deduplication is effective in organizations that have a lot of redundant data, such as backup systems that have several versions of the same file. Compression is effective in decreasing the size of unique files, such as images, videos and databases. While deduplication typically works at the block level, compression tends to work at the file level.

While deduplication and compression are storage-focused, encryption is more of a security feature.

## 3.6 How these three intersect

If an organization is using all three data protection technologies, it should dedupe and compress data prior to encryption. Encrypted files are tough to dedupe and compress because an organization will likely need the key to unlock the files first.

Compression and deduplication can have a negative effect on storage performance. Both data protection technologies need substantial compute resources and may increase latency. One way to find out how well a storage system incorporates compression and deduplication is to run a test.

In addition, organizations should be careful about resiliency when deduplicating and compressing. With both data protection methods removing redundant data, a storage environment may be left with one complete copy of content, so proper backup is an important element.

## 3.7 How deduplication and compression work with other technologies

Deduplication and compression often combine with other technologies for improved protection.

Deduplication, along with replication, aids disaster recovery. In this case, an organization would use deduplication first to reduce the amount of duplicate data and then replicate the data off-site.

Erasure coding can work with compression and deduplication to conserve storage capacity. Erasure coding reconstructs data that's been corrupted.

# 4. Data compression and backup

Compression is often used for data that's not accessed much, as the process can be intensive and slow down systems. Administrators, though, can seamlessly integrate compression in their backup systems.

Backup is a redundant type of workload, as the process captures the same files frequently. An organization that performs full backups will often have close to the same data from backup to backup.

There are major benefits to compressing data prior to backup:

* Data takes up less space, as a compression ratio can reach 100:1, but between 2:1 and 5:1 is common.
* If compression is done in a server prior to transmission, the time needed to transmit the data and the total network bandwidth are drastically reduced.
* On tape, the compressed, smaller file system image can be scanned faster to reach a particular file, reducing restore [latency](https://whatis.techtarget.com/definition/latency).
* Compression is supported by backup software and tape libraries, so there is a choice of data compression techniques.

# Pros and cons of compression

Simply put, compression is a process which trades CPU cycles for bytes.  But the trade isn't always a good one.  Sometimes you can spend a lot of valuable CPU cycles for little or no gain.

In the context of network data transport, "Should I compress?" is a common question.  But the answer can get complicated, depending on several factors.  The most important thing to remember is that compression can actually make your data move *much slower*, so it should not be used without some consideration.

## 5.1 When Compression Is Good

Compression algorithms try to identify large repeating patterns in a data set and replace them with smaller patterns.  Ideally, this shrinks the size of the data set.  For the purposes of network transport, having less data to move means it should take less time to move it.

Documents and files which consist mostly of plain text or machine executable code tend to compress well.  Examples include word processing documents, HTML files, some .exe files, and some database files.

Combining many small files into a single archive prior to network transfer can often result in faster speeds than transferring each file individually.  This may be true even if the individual files themselves are not compressible.  Many archiving utilities have options to pack files into an archive without compression, such as the "-0" option for "zip".  *ExpeDat* will combine the contents of a folder into a single data stream when you enable Streaming Folders.

## 5.2 When Compression Is Bad

Many data types are not compressible, because the repeating patterns have already been removed.  This includes most images, videos, songs, any data that is already compressed, or any data that has been encrypted.

Trying to compress data that is not compressible wastes CPU time.  When you are trying to move data at high speeds, that CPU time may be critical to feeding the network.  So by taking away processing time with worthless compression, you can actually end up moving your data much more slowly than if you had compression turned off.

If you are using a compression utility only for the purposes of combining many small files, check for options that disable compression.  For example, the "zip" command has a "-0" option which packages files into an archive without spending time trying to compress them.

## 5.3 Inline versus Offline

Many transport mechanisms allow you to apply compression algorithms to data as its being transferred.  This is convenient because the compression and decompression occur seamlessly without the user having to perform extra steps.  But it is also risky because any CPU time spent on compression is time NOT being spent on feeding data through the network.  If the network is very fast, the CPU is very slow, or the compression algorithm is unable to scale, having inline compression turned on may cause your data to move more slowly than if you turn compression off.  Inline compression can be slower than no compression even when the data is compressible!

If you are going to be transferring the same data set multiple times, it pays to compress it first using Zip or Tar-Gzip.  Then you can transfer the compressed archive without taking CPU cycles away from the network processing.  If you are planning to encrypt your data, make sure you *compress it first*, then encrypt second.

## 5.4 Hidden Compression

Devices in your network may be applying compression without you realizing it.  This becomes evident if the "speed" of the network seems to change for different data types.  If the network seems slow when you are transferring data that is already compressed, but fast when you are transferring uncompressed text files, then you can be pretty sure that something out there is making compression decisions for you.

Network compression devices can be helpful in that they take the compression burden away from the end-point CPUs.  But they can also create very inconsistent results since they will not work for all destinations and data types.  Network level compression can also run into the same CPU trade-offs discussed above, resulting in some files moving more slowly than they would if there was no compression.

If you are testing the speed of your network, try using data that is already compressed or encrypted to ensure consistent results.

## 5.5 Should I Turn On Inline Compression?

For compressed data, images, audio, video, or encrypted files: No.

For other types of data, test it both ways to see which is faster.

If the network is very fast (hundreds of megabits per second or faster), consider turning off inline compression and instead compress the data before you move it.

# GZIP Compression



GZip is a form of data compression -- ie it takes a chunk of data and makes it smaller. The original data can be restored by un-zipping the compressed file.

It is relevant to web apps and web sites because the HTTP protocol includes the ability to gzip data that is being sent.

This means that when it is in use, your bandwidth costs for serving the site will be lower because people visiting the site will be downloading smaller files.

There are a few caveats to using GZip, but overall it's usually better to use gzip than not to -- for example, it does take time and processor power to zip and unzip the files, but typically this is not a problem because the time it takes to do that is often less than the time that is saved by downloading a smaller file. Therefore the overall effect is a time saving, despite the browser having to unzip the file.

GZip can compress all files; it doesn't make any difference what the file type is or the encoding. Obviously some files can be compressed more effectively than others, so the bandwidth saving will vary - text files like HTML give the best results; images are not compressed so much by gzip because they already have some compression built-in. Some files (eg those that are already heavily compressed like .zip files) may actually get slightly bigger when gzipped, because they can't be compressed any futher but gzip still needs to add its Meta data to the file. But these are edge cases, and don't make much difference.

GZip across HTTP normally happens completely transparently. The end user should be completely unaware that it is happening; the browser would do it behind the scenes for them. And from the web server end it is simply a matter of turning on a config setting in your web server software. From your perspective, that's really all you need to know; just set the gzip setting on your server (or ask your ISP to do it). It's quite possible it may already be active on your site without you even knowing.

## 6.1 File Format

Gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE was intended as a replacement for LZW and other patent-encumbered data compression algorithms 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 (1f 8b), compression ID (08 for DEFLATE), file flags, a 32-bit timestamp, compression flags and operating system ID.
* optional extra headers denoted by file flags, such as the original filename
* a body, containing a DEFLATE-compressed payload
* An 8-byte footer, containing a CRC-32 checksum and the length of the original uncompressed data, modulo{\displaystyle 2^{32}}.

[Targzip.svg](https://en.wikipedia.org/wiki/File:Targzip.svg)

Although its file format also allows for multiple such streams to be concatenated (gzipped files are simply decompressed concatenated as if they were originally one file), gzip is normally used to compress just single files. Compressed archives are typically created by assembling collections of files into a single tar archive (also called tarball), and then compressing that archive with gzip. The final compressed file usually has the extension .tar.gz or .tgz.

gzip is not to be confused with the ZIP 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 holding the same data, because it compresses files individually and cannot take advantage of redundancy between files (solid compression)

## 6.2 Implementation

Various implementations of the program have been written. The most commonly known is the GNU Project's implementation using Lempel-Ziv coding (LZ77). 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*](https://en.wiktionary.org/wiki/gratis).[[7]](https://en.wikipedia.org/wiki/Gzip#cite_note-7) FreeBSD, DragonFly BSD and NetBSD use a BSD-licensed implementation instead of the GNU version; it is actually a command-line interface for zlib intended to be compatible with the GNU implementation's options.[[8]](https://en.wikipedia.org/wiki/Gzip#cite_note-8) These implementations originally come from NetBSD, and support decompression of bzip2 and the Unix pack format.

An alternative compression program achieving 3-8% better compression is Zopfli. It achieves gzip-compatible compression using more exhaustive algorithms, at the expense of compression time required. It does not affect decompression time.

pigz, written by Mark Adler, is compatible to gzip and speeds up compression by using all available CPU cores and threads.

## 6.3 Algorithm

It's important before trying to understand DEFLATE to understand the other two compression strategies that make it up -- Huffman coding and LZ77 compression.

### 6.3.1 Huffman coding

Huffman coding is a form of prefix coding, which you may not think you know. But you've almost certainly used a prefix code -- when using the phone. Starting from the dial tone, you press a sequence of what may be five, seven, eight, eleven, twelve, or some other number of keys -- and each sequence of keys reaches another specific phone line.

Now suppose you're in an office setting with an internal switchboard, as many large companies do. All other phones within the bank only require five numbers to dial, instead of seven -- that's because it's expected that you'll be calling those numbers more often. You may still need to call other numbers, though -- so all of those numbers have a `9' added to the front.

That's a prefix code. Each element that you could want to specify has a code made up of numbers, and because no code for one element begins with the code for any other element, you can type in that code and there will be no ambiguity about that being the one you mean.

A Huffman code is a prefix code prepared by a special algorithm. Here, instead of each code being a series of numbers between 0 and 9, each code is a series of bits, either 0 or 1. Instead of each code representing a phone, each code represents an element in a specific ``alphabet'' (such as the set of ASCII characters, which is the primary but not the only use of Huffman coding in DEFLATE).

A Huffman algorithm starts by assembling the elements of the ``alphabet,'' each one being assigned a ``weight'' -- a number that represents its relative frequency within the data to be compressed. These weights may be guessed at beforehand, or they may be measured exactly from passes through the data, or some combination of the two. In any case, the elements are selected two at a time, the elements with the lowest weights being chosen. The two elements are made to be leaf nodes of a node with two branches (I really hope you know nodes and trees...) Anyway, suppose we had a set of elements and weights that looked like this:

A 16

B 32

C 32

D 8

E 8

We would pick D and E first, and make them branches of a single node -- one being the `0' branch, and one the `1' branch.

( )

0 / \ 1

D E

At this point, no element has been given its complete code yet, but we now know that the codes for D and E will be exactly the same, except for the last binary digit: D will end in 0 and E in 1.

The combined node D-and-E is placed back with the other (as yet) uncombined elements, and given a weight equal to the sum of its leaf nodes: in this case, 8 + 8 = 16. Again, we take the two nodes with lowest weight, which are A, and D-and-E, and join them into a larger node.

( )

0 / \ 1

( ) A

0 / \ 1

D E

This time, when the node A-D-E is put back into the list of elements, all remaining elements have the same weight, 32. Which two of the three are selected to be combined first is not important, at least not in the classical Huffman algorithm.

When all nodes have been recombined into a single ``Huffman tree,'' then by starting at the root and selecting 0 or 1 at each step, you can reach any element in the tree. Each element now has a Huffman code, which is the sequence of 0's and 1's that represents that path through the tree.

Now, it should be fairly easy to see how such a tree, and such a set of codes, could be used for compression. If compressing ordinary text, for example, probably more than half of the ASCII character set could be left out of the tree altogether. Frequently used characters, like `E' and `T' and `A,' will probably get much shorter codes, and even if some codes are actually made longer, they will be the ones that are used less often.

However, there is also the question: how do you pass the tree along with the encoded data? It turns out that there is a fairly simple way, if you modify slightly the algorithm used to generate the tree.

In the classic Huffman algorithm, a single set of elements and weights could generate multiple trees. In the variation used by the Deflate standard, there are two additional rules: elements that have shorter codes are placed to the left of those with longer codes. (In our previous example, D and E wind up with the longest codes, and so they would be all the way to the right.) Among elements with codes of the same length, those that come first in the element set are placed to the left. (If D and E end up being the only elements with codes of that length, then D will get the 0 branch and E the 1 branch, as D comes before E.)

It turns out that when these two restrictions are placed upon the trees, there is at most one possible tree for every set of elements and their respective codelengths. The codelengths are all that we need to reconstruct the tree, and therefore all that we need to transmit.

### 6.3.2 LZ77 compression

LZ77 compression works by finding sequences of data that are repeated. The term ``sliding window'' is used; all it really means is that at any given point in the data, there is a record of what characters went before. A 32K sliding window means that the compressor (and decompressor) have a record of what the last 32768 (32 \* 1024) characters were. When the next sequence of characters to be compressed is identical to one that can be found within the sliding window, the sequence of characters is replaced by two numbers: a distance, representing how far back into the window the sequence starts, and a length, representing the number of characters for which the sequence is identical.

I realize this is a lot easier to see than to just be told. Let's look at some highly compressible data:

Blah blah blah blah blah!

Our datastream starts by receiving the following characters: `B,' `l,' `a,' `h,' ` ,' and `b.' However, look at the next five characters:

vvvvv

Blah blah blah blah blah!

^^^^^

There is an exact match for those five characters in the characters that have already gone into the datastream, and it starts exactly five characters behind the point where we are now. This being the case, we can output special characters to the stream that represent a number for length, and a number for distance.

The data so far:

Blah blah b

The compressed form of the data so far:

Blah b[D=5,L=5]

The compression can still be increased, though to take full advantage of it requires a bit of cleverness on the part of the compressor. Look at the two strings that we decided were identical. Compare the character that follows each of them. In both cases, it's `l' -- so we can make the length 6, and not just five. But if we continue checking, we find the next characters, and the next characters, and the next characters, are still identical -- even if the so-called ``previous'' string is overlapping the string we're trying to represent in the compressed data!

It turns out that the 18 characters that start at the second character are identical to the 18 characters that start at the seventh character. It's true that when we're decompressing, and read the length, distance pair that describes this relationship, we don't know what all those 18 characters will be yet -- but if we put in place the ones that we know, we will know more, which will allow us to put down more... or, knowing that any length-and-distance pair where length > distance is going to be repeating (distance) characters again and again, we can set up the decompressor to do just that.

It turns out our highly compressible data can be compressed down to just this:

Blah b[D=5, L=18]!

### 6.3.3 Putting it all together

The deflate compressor is given a great deal of flexibility as to *how* to compress the data. The programmer must deal with the problem of designing smart algorithms to make the *right* choices, but the compressor *does* have choices about how to compress data.

There are three modes of compression that the compressor has available:

1. Not compressed at all. This is an intelligent choice for, say, data that's already been compressed. Data stored in this mode will expand slightly, but not by as much as it would if it were already compressed and one of the other compression methods was tried upon it.
2. Compression, first with LZ77 and then with Huffman coding. The trees that are used to compress in this mode are defined by the Deflate specification itself, and so no extra space needs to be taken to store those trees.
3. Compression, first with LZ77 and then with Huffman coding with trees that the compressor creates and stores along with the data.

The data is broken up in ``blocks,'' and each block uses a single mode of compression. If the compressor wants to switch from non-compressed storage to compression with the trees defined by the specification, or to compression with specified Huffman trees, or to compression with a different pair of Huffman trees, the current block must be ended and a new one begun.

# BZip2 Compression

**bzip2** is a free and open-source file compression program that uses the Burrows–Wheeler algorithm. It only compresses single files and is not a file archiver.

bzip2 compresses most files more effectively than the older LZW (.Z) and Deflate (.zip and .gz) compression algorithms, but is considerably slower. LZMA is generally more space-efficient than bzip2 at the expense of even slower compression speed, while having much faster decompression

bzip2 compresses data in blocks of size between 100 and 900 [kB](https://en.wikipedia.org/wiki/Kilobyte) and uses the Burrows–Wheeler transform to convert frequently-recurring character sequences into strings of identical letters. It then applies move-to-front transform and Huffman coding. bzip2's ancestor **bzip** used arithmetic coding instead of Huffman. The change was made because of a software patent restriction.

bzip2 performance is asymmetric, as decompression is relatively fast. Motivated by the large CPU time required for compression, a modified version was created in 2003 called pbzip2 that supported multi-threading, giving almost linear speed improvements on multi-CPU and multi-core computers. As of May 2010, this functionality has not been incorporated into the main project.

Like gzip, bzip2 is only a data compressor. It is not an archiver like tar or ZIP; the program itself has no facilities for multiple files, encryption or archive-splitting, but, in the UNIX tradition, relies instead on separate external utilities such as tar and GnuPG for these tasks.

## 7.1 File Format

No formal specification for bzip2 exists, although an informal specification has been reverse engineered from the reference implementation.

As an overview, a .bz2 stream consists of a 4-byte header, followed by zero or more compressed blocks, immediately followed by an end-of-stream marker containing a 32-bit CRC for the plaintext whole stream processed. The compressed blocks are bit-aligned and no padding occurs.

Because of the first-stage RLE compression (see above), the maximum length of plaintext that a single 900 kB bzip2 block can contain is around 46 MB (45,899,236 bytes). This can occur if the whole plaintext consists entirely of repeated values (the resulting .bz2 file in this case is 46 bytes long). An even smaller file of 40 bytes can be achieved by using an input containing entirely values of 251, an apparent compression ratio of 1147480.9:1.

The compressed blocks in bzip2 can be independently decompressed, without having to process earlier blocks. This means that bzip2 files can be decompressed in parallel, making it a good format for use in big data applications with cluster computing frameworks like Hadoop and Apache Spark.

## 7.2 Implementation

In addition to Julian Seward's original reference implementation, the following programs support bzip2 format.

* 7-Zip: Written by Igor Pavlov in C++, the 7-Zip suite contains a bzip2 encoder/decoder which is freely licensed. 7-Zip comes with multi-threading support.
* micro-bzip2: A version by Rob Landley designed for reduced compiled code size and available under the GNU LGPL.
* PBZIP2: Parallel pthreads-based implementation in C++ by Jeff Gilchrist (and Windows version).
* bzip2smp: A modification to libbzip2 that has SMP parallelisation "hacked in" by Konstantin Isakov.
* smpbzip2: Another go at parallel bzip2, by Niels Werensteijn.
* pyflate: A pure-Python stand-alone bzip2 and DEFLATE (gzip) decoder by Paul Sladen. Probably useful for research and prototyping, made available under the BSD/GPL/LGPL, or any other DFSG-compatible license.
* bz2: Python 3 module for supporting bzip2 compression (Python Standard Library).

## 7.3 Algorithm

Bzip2 uses several layers of compression techniques stacked on top of each other, which occur in the following order during compression and the reverse order during decompression:

1. Run-length encoding (RLE) of initial data
2. Burrows–Wheeler transform (BWT) or block sorting
3. Move to front (MTF) transform
4. Run-length encoding (RLE) of MTF result
5. Huffman coding
6. Selection between multiple Huffman tables
7. Unary base 1 encoding of Huffman table selection
8. Delta encoding (Δ) of Huffman code bit-lengths
9. Sparse bit array showing which symbols are used

### 7.3.1 Initial run-length encoding

Any sequence of 4 to 255 consecutive duplicate symbols is replaced by the first four symbols and a repeat length between 0 and 251. Thus the sequence "AAAAAAABBBBCCCD" is replaced with "AAAA\3BBBB\0CCCD", where "\3" and "\0" represent byte values 3 and 0 respectively. Runs of symbols are always transformed after four consecutive symbols, even if the run-length is set to zero, to keep the transformation reversible.

In the worst case, it can cause an expansion of 1.25 and best case a reduction to <0.02 . While the specification theoretically allows for runs of length 256–259 to be encoded, the reference encoder will not produce such output.

The author of bzip2 has stated that the RLE step was a historical mistake and was only intended to protect the original BWT implementation from pathological cases.

### 7.3.2 Burrows–Wheeler transform

This is the reversible block-sort that is at the core of bzip2. The block is entirely self-contained, with input and output buffers remaining the same size—in bzip2, the operating limit for this stage is 900 kB. For the block-sort, a (notional) matrix is created in which row {\displaystyle i} contains the whole of the buffer, rotated to start from the {\displaystyle i^{\mathrm {th} }} symbol. Following rotation, the rows of the matrix are sorted into alphabetic (numerical) order. A 24-bit pointer is stored marking the *starting position*for when the block is untransformed. In practice, it is not necessary to construct the full matrix; rather, the sort is performed using pointers for each position in the buffer. The output buffer is the last column of the matrix; this contains the whole buffer, but reordered so that it is likely to contain large runs of identical symbols.

### 7.3.2 Move to front transform

Again, this transform does not alter the size of the processed block. Each of the symbols in use in the document is placed in an array. When a symbol is processed, it is replaced by its location (index) in the array and that symbol is shuffled to the front of the array. The effect is that immediately recurring symbols are replaced by zero symbols (long runs of *any* arbitrary symbol thus become runs of zero symbols), while other symbols are remapped according to their local frequency.

Much "natural" data contains identical symbols that recur within a limited range (text is a good example). As the MTF transform assigns low values to symbols that reappear frequently, this results in a data stream which contains many symbols in the low integer range, many of them being identical (different recurring input symbols can actually map to the same output symbol). Such data can be very efficiently encoded by any legacy compression method.

### 7.3.4 Run-length encoding of MTF result

Long strings of zeros in the output of the move-to-front transform (which come from repeated symbols in the output of the BWT) are replaced by a sequence of two special codes, RUNA and RUNB, which represent the run-length as a binary number. Actual zeros are never encoded in the output; a lone zero becomes RUNA. (This step in fact is done at the same time as MTF is; whenever MTF would produce zero, it instead increases a counter to then encode with RUNA and RUNB.)

The sequence 0,0,0,0,0,1 would be represented as RUNA,RUNB,1; RUNA,RUNB represents the value 5 as described below. The run-length code is terminated by reaching another normal symbol. This RLE process is more flexible than the initial RLE step, as it is able to encode arbitrarily long integers (in practice, this is usually limited by the block size, so that this step does not encode a run of more than 900,000 bytes). The run-length is encoded in this fashion: assigning place values of 1 to the first bit, 2 to the second, 4 to the third, etc. in the sequence, multiply each place value in a RUNB spot by 2, and add all the resulting place values (for RUNA and RUNB values alike) together. This is similar to base-2 bijective numeration. Thus, the sequence RUNA,RUNB results in the value (1 + 2 × 2) = 5. As a more complicated example:

RUNA RUNB RUNA RUNA RUNB (ABAAB)

1 2 4 8 16

1 4 4 8 32 = 49

### 7.3.5 Multiple Huffman tables

Several identically-sized Huffman tables can be used with a block if the gain from using them is greater than the cost of including the extra table. At least two (2) and up to six (6) tables can be present, with the most appropriate table being reselected before every 50 symbols processed. This has the advantage of having very responsive Huffman dynamics without having to continuously supply new tables, as would be required in DEFLATE. Run-length encoding in the previous step is designed to take care of codes that have an inverse probability of use higher than the shortest code Huffman code in use.

### 7.3.6 Unary encoding of Huffman table selection

If multiple Huffman tables are in use, the selection of each table (numbered 0 to 5) is done from a list by a zero-terminated bit run between one (1) and six (6) bits in length. The selection is into a MTF list of the tables. Using this feature results in a maximum expansion of around 1.015, but generally less. This expansion is likely to be greatly over-shadowed by the advantage of selecting more appropriate Huffman tables and the common-case of continuing to use the same Huffman table is represented as a single bit. Rather than unary encoding, effectively this is an extreme form of a Huffman tree where each code has half the probability of the previous code.

### 7.3.7 Delta encoding

Huffman code bit-lengths are required to reconstruct each of the used canonical Huffman tables. Each bit-length is stored as an encoded difference against the previous code bit-length. A zero-bit (0) means that the previous bit-length should be duplicated for the current code, whilst a one-bit (1) means that a further bit should be read and the bit-length incremented or decremented based on that value.

In the common case a single bit is used per symbol per table and the worst case—going from length one (1) to length twenty (20)—would require approximately 37 bits. As a result of the earlier MTF encoding, code lengths would start at 2–3 bits long (very frequently used codes) and gradually increase, meaning that the delta format is fairly efficient—requiring around 300 bits (38 bytes) per full Huffman table.

### 7.3.8 Sparse bit array

A bitmap is used to show which symbols are used inside the block and should be included in the Huffman trees. Binary data is likely to use all 256 symbols representable by a byte, whereas textual data may only use a small subset of available values, perhaps covering the ASCII range between 32 and 126. Storing 256 zero bits would be inefficient if they were mostly unused. A *sparse* method is used: the 256 symbols are divided up into 16 ranges and only if symbols are used within that block is a 16-bit array included. The presence of each of these 16 ranges is indicated by an additional 16-bit bit array at the front.

The total bitmap uses between 32 and 272 bits of storage (4–34 bytes). For contrast, the DEFLATE algorithm would show the absence of symbols by encoding the symbols as having a (zero) bit-length with Run Length Encoding and additional Huffman coding.

# Difference between GZIP and BZip2

## 8.1 Compression

When there's need for a very fast compression, gzip is the clear winner. It has also very small memory footprint, making it ideal for systems with limited memory.

bzip2 creates about 15% smaller files than gzip. bzip2 compresses somewhat slower than gzip, but seems that it hasn't prevented bzip2 from getting popular. Nowadays most source code is available as both gzip and bzip2 compressed tar archives.

"lzmash -3" and "lzmash -4" seem to be almost as fast (or slow); same can be said for "lzmash -5", "lzmash -6" and "lzmash -7". However the memory requirements increase with every option meaning that "lzmash -3", "lzmash -5" and "lzmash -6" are usually useful only if you (or the recipient) do not have enough memory for "lzmash -4" or "lzmash -7".

"lzmash -8" and "lzmash -9" require lots of memory and are practical only on newer computers; the files compressed with them are probably a pain to decompress on systems with less than 32 MB or 64 MB of memory.

The extreme mode ("lzmash -e") roughly doubles the compression time, but especially with small files can lead to even worse compression ratio than normal the mode. The extereme mode might be worth trying if you want make as small files as possible, but in that case forgetting lzmash wrapper script and playing with command line options of "lzma" directly can lead to better results.

## 8.2 Decompression

In terms of speed, gzip is the winner again. lzma comes right behind it two to three times slower than gzip. bzip2 is a lot slower taking usually two to six times more time than lzma, that is, four to twelve times more than gzip. One interesting thing is that gzip and lzma decompress the faster the smaller the compressed size is, while bzip2 gets slower when the compression ratio gets better.

The memory usage of lzma stays competitive with bzip2 when files have been compressed with "lzmash -6" or with a smaller option. The files compressed with the default "lzmash -7" can still be decompressed, even on machines with only 16 MB of RAM, but sometimes you don't have even that much memory available. If you compress with "lzmash -8" or "lzmash -9", you should think if the users need to be able to decompress your files also on "ancient" computers.

## 8.3 So what is the best?

Of course, it depends on the intended application. gzip is very fast and has small memory footprint. According to this benchmark, neither bzip2 nor lzma can compete with gzip in terms of speed or memory usage. bzip2 has notably better compression ratio than gzip, which has to be the reason for the popularity of bzip2; it is slower than gzip especially in decompression and uses more memory. However the memory requirements of bzip2 should be nowadays no problem even on older hardware.

Both gzip and bzip2 are bundled with practically all GNU/\*/Linux distributions and \*BSDs. Because everybody has the tools to handle gzip and bzip2 compressed files, they are by far the most commonly used formats to distribute e.g. source code of free software. However, the situation might change because better free (as in freedom) alternatives have become available.

LZMA clearly has potential to become the third commonly used general purporse compression format on \*NIX systems. It mainly competes with bzip2 by offering significantly better compression ratio while still keeping decompressing speed relatively close to that of gzip. Its excellence has been already seen in Tukaani Linux package management system, and in software installers such as **Nullsoft Scriptable Install System** (NSIS), **Inno setup** and installers of MS-Windows versions of **Mozilla** products, including Firefox and Thunderbird.