



WIKIPEDIA
The Free Encyclopedia

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

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

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

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Català
Čeština
Dansk
Deutsch
Eesti
Ελληνικά
Español
فارسی
Français
한국어
Italiano
עברית
Magyar
Nederlands
日本語
Norsk bokmål
Polski
Português
Русский
Српски / srpski
Suomi
Svenska
ไทย
Türkçe
Українська
Tiếng Việt
中文

Edit links

Create account Log in

Article Talk

Read Edit View history

Search

Huffman coding

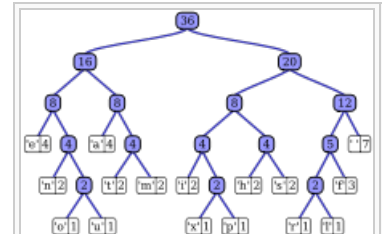
From Wikipedia, the free encyclopedia



This article includes a [list of references](#), but **its sources remain unclear** because it has **insufficient inline citations**. Please help to [improve](#) this article by [introducing](#) more precise citations. *(January 2011)*

In [computer science](#) and [information theory](#), a **Huffman code** is a particular type of optimal [prefix code](#) that is commonly used for [lossless data compression](#). The process of finding and/or using such a code proceeds by means of **Huffman coding**, an algorithm developed by [David A. Huffman](#) while he was a [Ph.D.](#) student at [MIT](#), and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".^[1]

The output from Huffman's algorithm can be viewed as a [variable-length code](#) table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other [entropy encoding](#) methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in [linear time](#) to the number of input weights if these weights are sorted.^[2] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.



Huffman tree generated from the exact frequencies of the text "this is an example of a huffman tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 bits, as opposed to 288 (or 180) bits if 36 characters of 8 (or 5) bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Contents

- History
- Terminology
- Problem definition
 - Informal description
 - Formalized description
 - Example
- Basic technique
 - Compression
 - Decompression
- Main properties
 - Optimality
- Variations
 - n*-ary Huffman coding
 - Adaptive Huffman coding
 - Huffman template algorithm
 - Length-limited Huffman coding/minimum variance Huffman coding
 - Huffman coding with unequal letter costs
 - Optimal alphabetic binary trees (Hu–Tucker coding)
 - The canonical Huffman code
- Applications
- See also
- Notes
- References

History [[edit](#)]

In 1951, [David A. Huffman](#) and his [MIT information theory](#) classmates were given the choice of a term paper or a final [exam](#). The professor, [Robert M. Fano](#), assigned a [term paper](#) on the problem of finding the most efficient

binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted [binary tree](#) and quickly proved this method the most efficient.^[3]

In doing so, the student outdid his professor, who had worked with [information theory](#) inventor [Claude Shannon](#) to develop a similar code. By building the tree from the bottom up instead of the top down, Huffman avoided the major flaw of the suboptimal [Shannon-Fano coding](#).

Terminology [\[edit\]](#)

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol). Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Problem definition [\[edit\]](#)

Informal description [\[edit\]](#)

Given

A set of symbols and their weights (usually [proportional](#) to probabilities).

Find

A [prefix-free binary code](#) (a set of codewords) with minimum [expected](#) codeword length (equivalently, a tree with minimum [weighted path length from the root](#)).

Formalized description [\[edit\]](#)

Input.

Alphabet $A = \{a_1, a_2, \dots, a_n\}$, which is the symbol alphabet of size n .

Set $W = \{w_1, w_2, \dots, w_n\}$, which is the set of the (positive) symbol weights (usually proportional to probabilities), i.e.

$$w_i = \text{weight}(a_i), 1 \leq i \leq n.$$

Output.

Code $C(A, W) = (c_1, c_2, \dots, c_n)$, which is the tuple of (binary) codewords, where c_i is the codeword for a_i , $1 \leq i \leq n$.

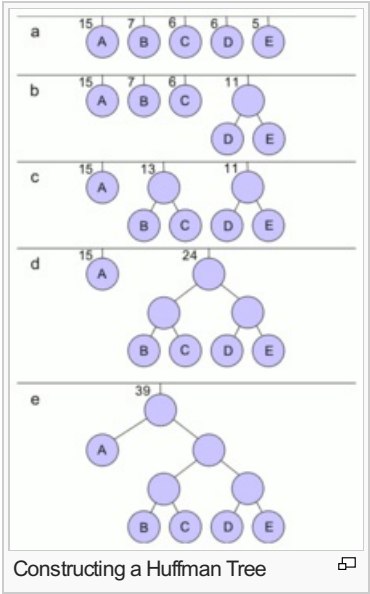
Goal.

Let $L(C) = \sum_{i=1}^n w_i \times \text{length}(c_i)$ be the weighted path length of code C . Condition: $L(C) \leq L(T)$ for any code $T(A, W)$.

Example [\[edit\]](#)

We give an example of the result of Huffman coding for a code with five words and given weights. We will not verify that it minimizes L over all codes (it does of course), but we will compute L and compare it to the Shannon entropy H of the given set of weights; the result is nearly optimal.

Input (A, W)	Symbol (a_i)	a	b	c	d	e	Sum
	Weights (w_i)	0.10	0.15	0.30	0.16	0.29	= 1
Output C	Codewords (c_i)	010	011	11	00	10	$L(C) = 2.25$
	Codeword length (in bits) (l_i)	3	3	2	2	2	
	Contribution to weighted path length ($l_i w_i$)	0.30	0.45	0.60	0.32	0.58	
	Probability budget (2^{-l_i})	1/8	1/8	1/4	1/4	1/4	



Constructing a Huffman Tree

Optimality	Information content (in bits) ($-\log_2 w_i \approx$	3.32	2.74	1.74	2.64	1.79	
	Contribution to entropy ($-w_i \log_2 w_i$)	0.332	0.411	0.521	0.423	0.518	$H(A) = 2.205$

For any code that is *biunique*, meaning that the code is *uniquely decodeable*, the sum of the probability budgets across all symbols is always less than or equal to one. In this example, the sum is strictly equal to one; as a result, the code is termed a *complete* code. If this is not the case, you can always derive an equivalent code by adding extra symbols (with associated null probabilities), to make the code complete while keeping it *biunique*.

As defined by [Shannon \(1948\)](#), the information content h (in bits) of each symbol a_i with non-null probability is

$$h(a_i) = \log_2 \frac{1}{w_i}.$$

The [entropy](#) H (in bits) is the weighted sum, across all symbols a_i with non-zero probability w_i , of the information content of each symbol:

$$H(A) = \sum_{w_i > 0} w_i h(a_i) = \sum_{w_i > 0} w_i \log_2 \frac{1}{w_i} = - \sum_{w_i > 0} w_i \log_2 w_i.$$

(Note: A symbol with zero probability has zero contribution to the entropy, since $\lim_{w \rightarrow 0^+} w \log_2 w = 0$. So for simplicity, symbols with zero probability can be left out of the formula above.)

As a consequence of [Shannon's source coding theorem](#), the entropy is a measure of the smallest codeword length that is theoretically possible for the given alphabet with associated weights. In this example, the weighted average codeword length is 2.25 bits per symbol, only slightly larger than the calculated entropy of 2.205 bits per symbol. So not only is this code optimal in the sense that no other feasible code performs better, but it is very close to the theoretical limit established by Shannon.

In general, a Huffman code need not be unique. Thus the set of Huffman codes for a given probability distribution is a non-empty subset of the codes minimizing $L(C)$ for that probability distribution. (However, for each minimizing codeword length assignment, there exists at least one Huffman code with those lengths.)

Basic technique [\[edit\]](#)

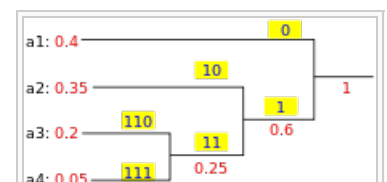
Compression [\[edit\]](#)

The technique works by creating a [binary tree](#) of nodes. These can be stored in a regular [array](#), the size of which depends on the number of symbols, n . A node can be either a [leaf node](#) or an [internal node](#). Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol and optionally, a link to a **parent** node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol **weight**, links to **two child nodes** and the optional link to a **parent** node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n - 1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node (thus not considering them anymore), and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree.

The simplest construction algorithm uses a [priority queue](#) where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.



A source generates 4 different symbols $\{a_1, a_2, a_3, a_4\}$ with probability $\{0.4; 0.35; 0.2; 0.05\}$. A binary tree is generated from left to right taking the two least probable symbols and putting them together to form another equivalent symbol having a probability that equals the sum of the two symbols. The process is repeated until there is just one symbol. The tree can then be read backwards, from right to left, assigning different bits to different branches. The final Huffman code is:

Symbol	Code
a1	0
a2	10
a3	110
a4	111

The standard way to represent a signal made of 4 symbols is by using 2 bits/symbol, but the [entropy](#) of the

2. While there is more than one node in the queue:
 1. Remove the two nodes of highest priority (lowest probability) from the queue
 2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
 3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

source is 1.74 bits/symbol. If this Huffman code is used to represent the signal, then the average length is lowered to 1.85 bits/symbol; it is still far from the theoretical limit because the probabilities of the symbols are different from negative powers of two.

Since efficient priority queue data structures require $O(\log n)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in $O(n \log n)$ time, where n is the number of symbols.

If the symbols are sorted by probability, there is a **linear-time** ($O(n)$) method to create a Huffman tree using two **queues**, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
 1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
 2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
 3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

Although linear-time given sorted input, in the general case of arbitrary input, using this algorithm requires pre-sorting. Thus, since sorting takes $O(n \log n)$ time in the general case, both methods have the same overall complexity.

In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

Here's an example of optimized Huffman coding using the French subject string "j'aime aller sur le bord de l'eau les jeudis ou les jours impairs". Note that the original Huffman coding tree structure would be different from the given example:

File :

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

Decompression [\[edit\]](#)

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using [canonical encoding](#), the compression model can be precisely reconstructed with just $B2^B$ bits of information (where B is the number of bits per symbol). Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).

Main properties [\[edit\]](#)

The probabilities used can be generic ones for the application domain that are based on average experience, or they can be the actual frequencies found in the text being compressed. This requires that a [frequency table](#) must be stored with the compressed text. See the Decompression section above for more information about the various techniques employed for this purpose.

Optimality [\[edit\]](#)

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e., a stream of unrelated

symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the [probability mass functions](#) are unknown. Also, if symbols are not [independent and identically distributed](#), a single code may be insufficient for optimality. Other methods such as [arithmetic coding](#) and [LZW](#) coding often have better compression capability: Both of these methods can combine an arbitrary number of symbols for more efficient coding, and generally adapt to the actual input statistics, useful when input probabilities are not precisely known or vary significantly within the stream. However, these methods have higher computational complexity. Also, both arithmetic coding and LZW were historically a subject of some concern over [patent](#) issues. However, as of mid-2010, the most commonly used techniques for these alternatives to Huffman coding have passed into the public domain as the early patents have expired.

However, the limitations of Huffman coding should not be overstated; it can be used adaptively, accommodating unknown, changing, or context-dependent probabilities. In the case of known independent and identically distributed random variables, combining symbols ("blocking") reduces inefficiency in a way that approaches optimality as the number of symbols combined increases. Huffman coding is optimal when each input symbol is a known independent and identically distributed random variable having a probability that is an the inverse of a power of two.

Prefix codes tend to have inefficiency on small alphabets, where probabilities often fall between these optimal points. The worst case for Huffman coding can happen when the probability of a symbol exceeds $2^{-1} = 0.5$, making the upper limit of inefficiency unbounded. These situations often respond well to a form of blocking called [run-length encoding](#); for the simple case of [Bernoulli processes](#), [Golomb coding](#) is a probably optimal run-length code.

For a set of symbols with a uniform probability distribution and a number of members which is a [power of two](#), Huffman coding is equivalent to simple binary [block encoding](#), e.g., [ASCII](#) coding. This reflects the fact that compression is not possible with such an input.

Variations [\[edit\]](#)

Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others of which find optimal prefix codes (while, for example, putting different restrictions on the output). Note that, in the latter case, the method need not be Huffman-like, and, indeed, need not even be [polynomial time](#). An exhaustive list of papers on Huffman coding and its variations is given by "Code and Parse Trees for Lossless Source Encoding"[\[1\]](#)[↗](#).

***n*-ary Huffman coding** [\[edit\]](#)

The ***n*-ary Huffman** algorithm uses the $\{0, 1, \dots, n - 1\}$ alphabet to encode message and build an *n*-ary tree. This approach was considered by Huffman in his original paper. The same algorithm applies as for binary (*n* equals 2) codes, except that the *n* least probable symbols are taken together, instead of just the 2 least probable. Note that for *n* greater than 2, not all sets of source words can properly form an *n*-ary tree for Huffman coding. In this case, additional 0-probability place holders must be added. This is because the tree must form an *n* to 1 contractor; for binary coding, this is a 2 to 1 contractor, and any sized set can form such a contractor. If the number of source words is congruent to 1 modulo *n*-1, then the set of source words will form a proper Huffman tree.

Adaptive Huffman coding [\[edit\]](#)

A variation called [adaptive Huffman coding](#) involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates. It is used rarely in practice, since the cost of updating the tree makes it slower than optimized [adaptive arithmetic coding](#), that is more flexible and has a better compression.

Huffman template algorithm [\[edit\]](#)

Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only that the weights form a [totally ordered commutative monoid](#), meaning a way to order weights and to add them. The **Huffman template algorithm** enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition). Such algorithms can solve other minimization problems, such as minimizing $\max_i [w_i + \text{length}(c_i)]$, a problem first applied to circuit design.

Length-limited Huffman coding/minimum variance Huffman coding [\[edit\]](#)

Length-limited Huffman coding is a variant where the goal is still to achieve a minimum weighted path length,

but there is an additional restriction that the length of each codeword must be less than a given constant. The [package-merge algorithm](#) solves this problem with a simple [greedy](#) approach very similar to that used by Huffman's algorithm. Its time complexity is $O(nL)$, where L is the maximum length of a codeword. No algorithm is known to solve this problem in [linear or linearithmic](#) time, unlike the presorted and unsorted conventional Huffman problems, respectively.

Huffman coding with unequal letter costs [\[edit\]](#)

In the standard Huffman coding problem, it is assumed that each symbol in the set that the code words are constructed from has an equal cost to transmit: a code word whose length is N digits will always have a cost of N , no matter how many of those digits are 0s, how many are 1s, etc. When working under this assumption, minimizing the total cost of the message and minimizing the total number of digits are the same thing.

Huffman coding with unequal letter costs is the generalization without this assumption: the letters of the encoding alphabet may have non-uniform lengths, due to characteristics of the transmission medium. An example is the encoding alphabet of [Morse code](#), where a 'dash' takes longer to send than a 'dot', and therefore the cost of a dash in transmission time is higher. The goal is still to minimize the weighted average codeword length, but it is no longer sufficient just to minimize the number of symbols used by the message. No algorithm is known to solve this in the same manner or with the same efficiency as conventional Huffman coding, though it has been solved by [Karp](#) <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1057615&newsearch=true&queryText=Minimum-redundancy%20coding%20for%20the%20discrete%20noiseless%20channel>> whose solution has been refined for the case of integer costs by [Golin](#) <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=705558&queryText=dynamic%20programming%20golin%20constructing%20optimal%20prefix-free&newsearch=true>>.

Optimal alphabetic binary trees (Hu–Tucker coding) [\[edit\]](#)

In the standard Huffman coding problem, it is assumed that any codeword can correspond to any input symbol. In the alphabetic version, the alphabetic order of inputs and outputs must be identical. Thus, for example, $A = \{a, b, c\}$ could not be assigned code $H(A, C) = \{00, 1, 01\}$, but instead should be assigned either $H(A, C) = \{00, 01, 1\}$ or $H(A, C) = \{0, 10, 11\}$. This is also known as the **Hu–Tucker** problem, after [T. C. Hu](#) and [Alan Tucker](#), the authors of the paper presenting the first [linearithmic](#) solution to this optimal binary alphabetic problem,^[4] which has some similarities to Huffman algorithm, but is not a variation of this algorithm. These optimal alphabetic binary trees are often used as [binary search trees](#).

The canonical Huffman code [\[edit\]](#)

If weights corresponding to the alphabetically ordered inputs are in numerical order, the Huffman code has the same lengths as the optimal alphabetic code, which can be found from calculating these lengths, rendering Hu–Tucker coding unnecessary. The code resulting from numerically (re-)ordered input is sometimes called the [canonical Huffman code](#) and is often the code used in practice, due to ease of encoding/decoding. The technique for finding this code is sometimes called **Huffman-Shannon-Fano coding**, since it is optimal like Huffman coding, but alphabetic in weight probability, like [Shannon-Fano coding](#). The Huffman-Shannon-Fano code corresponding to the example is $\{000, 001, 01, 10, 11\}$, which, having the same codeword lengths as the original solution, is also optimal. But in [canonical Huffman code](#), the result is $\{110, 111, 00, 01, 10\}$.

Applications [\[edit\]](#)

[Arithmetic coding](#) can be viewed as a generalization of Huffman coding, in the sense that they produce the same output when every symbol has a probability of the form $1/2^k$; in particular it tends to offer significantly better compression for small alphabet sizes. Huffman coding nevertheless remains in wide use because of its simplicity and high speed. Intuitively, arithmetic coding can offer better compression than Huffman coding because its "code words" can have effectively non-integer bit lengths, whereas code words in Huffman coding can only have an integer number of bits. Therefore, there is an inefficiency in Huffman coding where a code word of length k only optimally matches a symbol of probability $1/2^k$ and other probabilities are not represented as optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol.

Huffman coding today is often used as a "back-end" to some other compression methods. [DEFLATE](#) (PKZIP's algorithm) and multimedia [codecs](#) such as [JPEG](#) and [MP3](#) have a front-end model and [quantization](#) followed by Huffman coding (or variable-length prefix-free codes with a similar structure, although perhaps not necessarily designed by using Huffman's algorithm^{[\[clarification needed\]](#)}).

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

- Adaptive Huffman coding
- Data compression
- Group 4 compression
- Huffyuv
- Lempel–Ziv–Welch
- Modified Huffman coding - used in fax machines
- Shannon-Fano coding
- Varicode


Notes [\[edit\]](#)

1. [^] [Huffman, D.](#) (1952). "A Method for the Construction of Minimum-Redundancy Codes" [\[PDF\]](#). *Proceedings of the IRE* **40** (9): 1098–1101. doi:10.1109/JRPROC.1952.273898 [↗](#).

2. [^] [Van Leeuwen, Jan](#) (1976). "On the construction of Huffman trees" [\[PDF\]](#). *ICALP*: 382–410. Retrieved 20 February 2014.

3. [^] see [Ken Huffman](#) (1991)

4. [^] [Hu, T. C.](#); [Tucker, A. C.](#) (1971). "Optimal Computer Search Trees and Variable-Length Alphabetical Codes". *SIAM Journal on Applied Mathematics* **21** (4): 514. doi:10.1137/0121057 [↗](#). JSTOR 2099603 [↗](#).



Wikimedia Commons has media related to [Huffman coding](#).

References [\[edit\]](#)



• [D.A. Huffman](#), "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the I.R.E.*, September 1952, pp 1098–1102. Huffman's original article.

• [Ken Huffman](#). [Profile: David A. Huffman](#) [↗](#), *Scientific American*, September 1991, pp. 54–58

• [Thomas H. Cormen](#), [Charles E. Leiserson](#), [Ronald L. Rivest](#), and [Clifford Stein](#). *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 16.3, pp. 385–392.

v · t · e

Data compression methods

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

Categories: [1952 in computer science](#) | [Lossless compression algorithms](#) | [Binary trees](#)