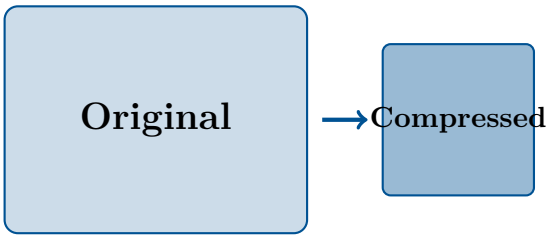
Compression Algorithms

A Comprehensive Guide



RLE • Delta • Huffman • LZ77/LZ78 • LZW
DEFLATE • LZ4 • Zstandard • Brotli • bzip2

Abdessamad JAOUAD
M2 Big Data & IoT
ENSAM Casablanca

January 2025

Contents

Chapter 1

Introduction to Data Compression

1.1 What is Data Compression?

Data compression is the process of reducing the number of bits needed to represent data. This reduction allows for:

- **Storage savings**: Store more data in the same space
- **Faster transmission**: Send data more quickly over networks
- **Reduced costs**: Lower storage and bandwidth expenses

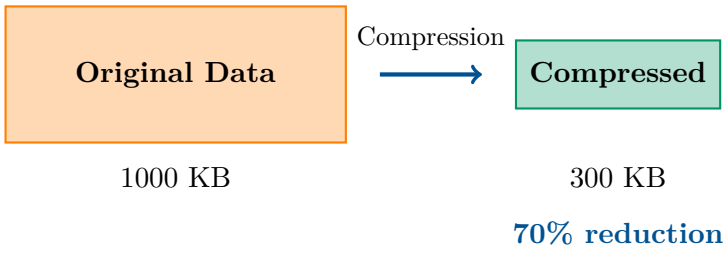


Figure 1.1: The compression process reduces data size significantly

1.2 Types of Compression

1.2.1 Lossless Compression

Data is compressed without any loss of information. The original data can be perfectly reconstructed.

Use cases: Text files, executable programs, archives, databases
Algorithms: Huffman, LZW, DEFLATE, bzip2, LZMA

1.2.2 Lossy Compression

Some information is permanently discarded to achieve higher compression ratios.

Use cases: Images (JPEG), audio (MP3), video (H.264)

Aspect	Lossless	Lossy
Data Recovery	Perfect	Approximate
Compression Ratio	2:1 to 10:1	10:1 to 100:1
Use Case	Text, Code	Media

Table 1.1: Comparison of lossless and lossy compression

1.3 Key Concepts

1.3.1 Compression Ratio

$$\text{Compression Ratio} = \frac{\text{Original Size}}{\text{Compressed Size}} \tag{1.1}$$

A ratio of 3:1 means the compressed file is 1/3 the size of the original.

1.3.2 Entropy

Shannon entropy measures the theoretical minimum bits needed to encode data:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{1.2}$$

where $p(x_i)$ is the probability of symbol x_i.

1.4 Classification of Algorithms

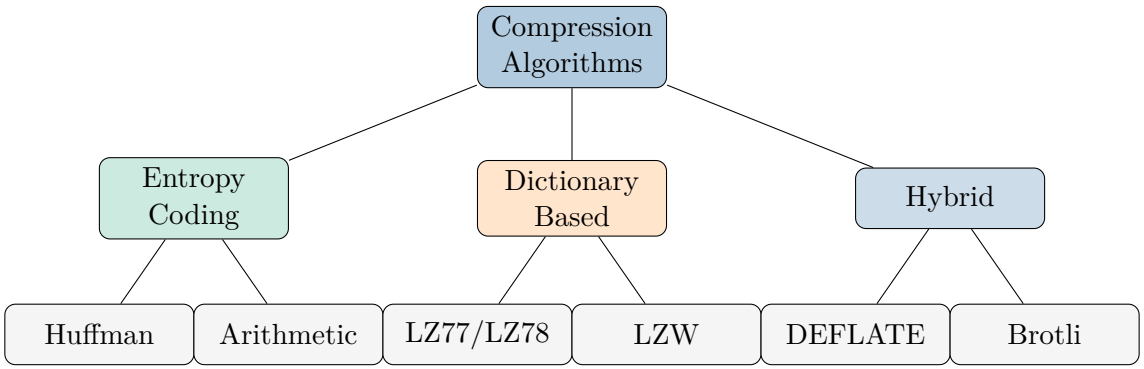


Figure 1.2: Classification of compression algorithms

Chapter 2

Run-Length Encoding (RLE)

2.1 Overview

Run-Length Encoding is one of the simplest compression algorithms. It replaces sequences of identical symbols (runs) with a count and the symbol.

- **Type**: Lossless
- **Complexity**: $O(n)$ time and space
- **Best for**: Data with many consecutive repeated values

2.2 How It Works

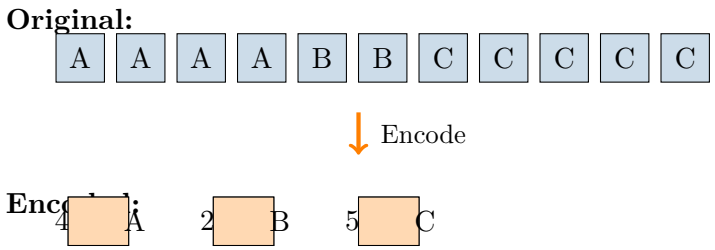


Figure 2.1: RLE encoding: AAAABBCCCCC → 4A2B5C

2.3 Algorithm

Algorithm 1 Run-Length Encoding

Input: String S of length n
Output: Encoded string
$result \leftarrow$ empty string
$count \leftarrow 1$
for $i = 1$ to $n - 1$ **do**
  if $S[i] = S[i-1]$ **then**
    $count \leftarrow count + 1$
  else
    Append $count$ and $S[i-1]$ to $result$
    $count \leftarrow 1$
  end if
end for
Append $count$ and $S[n-1]$ to $result$
return $result$

2.4 Example

Input	Output	Original	Compressed
WWWWWWWWWWWWBWWW	12W1B3W	16 chars	7 chars
AABBBCCCC	2A3B4C	9 chars	6 chars
ABCDEF	1A1B1C1D1E1F	6 chars	12 chars (expansion!)

Table 2.1: RLE encoding examples

2.5 Advantages and Limitations

Advantages	Limitations
• Very simple to implement	• Can expand data with few runs
• Fast encoding/decoding	• Not effective for random data
• No dictionary needed	• Limited compression ratio
• Low memory usage	• Best only for specific data types

2.6 Applications

- **Fax transmission** (CCITT Group 3/4)
- **Bitmap images** (BMP, PCX, TIFF)
- **Icons and simple graphics**
- **PackBits** (used in TIFF)

Chapter 3

Delta Encoding

3.1 Overview

Delta encoding stores differences between consecutive values rather than the values themselves. This is highly effective when consecutive values are similar.

- **Type**: Lossless
- **Complexity**: $O(n)$
- **Best for**: Time series, sensor data, version control

3.2 How It Works

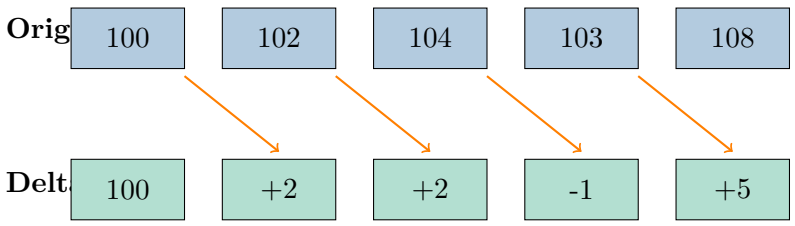


Figure 3.1: Delta encoding stores differences: [100, 102, 104, 103, 108] → [100, +2, +2, -1, +5]

3.3 Algorithm

Algorithm 2 Delta Encoding

Input: Array A of n values
Output: Delta-encoded array D
$D[0] \leftarrow A[0]$ {Store first value as-is}
for $i = 1$ to $n - 1$ **do**
    $D[i] \leftarrow A[i] - A[i-1]$ {Store difference}
end for
return D

3.4 Why It Compresses

When data has small variations, delta values are small numbers requiring fewer bits:

Data Type	Original Bits	Delta Bits
Temperature (20-25°C)	5 bits each	3 bits each
Stock prices	16+ bits each	4-8 bits each
Sensor readings	12 bits each	4-6 bits each

Table 3.1: Bit savings with delta encoding

3.5 Applications

- **Video compression**: Motion vectors are delta-encoded
- **Version control**: Git stores file differences
- **rsync**: Transfers only changed bytes
- **Time-series databases**: InfluxDB, TimescaleDB
- **HTTP delta encoding**: RFC 3229

Chapter 4

Huffman Coding

4.1 Overview

Huffman coding is an entropy encoding algorithm that creates optimal prefix-free codes based on symbol frequencies. More frequent symbols get shorter codes.

- **Inventor**: David Huffman (1952, MIT)
- **Type**: Entropy coding (lossless)
- **Complexity**: $O(n \log n)$ for tree construction
- **Property**: Optimal for symbol-by-symbol encoding

4.2 How It Works

1. Count frequency of each symbol
2. Create leaf nodes for each symbol
3. Build tree by repeatedly combining lowest-frequency nodes
4. Assign codes: left = 0, right = 1

4.3 Building the Huffman Tree

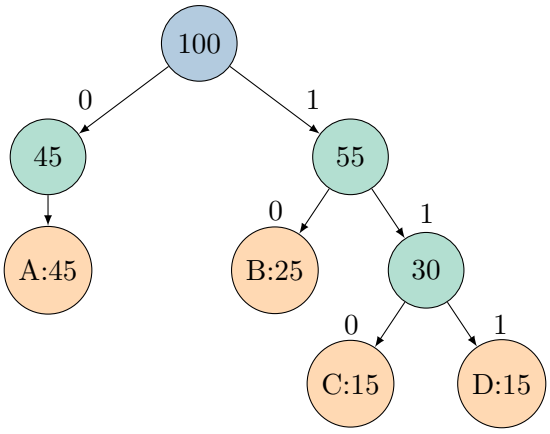


Figure 4.1: Huffman tree for symbols A(45%), B(25%), C(15%), D(15%)

4.4 Resulting Codes

Symbol	Frequency	Huffman Code	Fixed Code
A	45%	0 (1 bit)	00 (2 bits)
B	25%	10 (2 bits)	01 (2 bits)
C	15%	110 (3 bits)	10 (2 bits)
D	15%	111 (3 bits)	11 (2 bits)
Average		**1.75 bits**	**2 bits**

Table 4.1: Huffman codes vs fixed-length codes

4.5 Algorithm

Algorithm 3 Huffman Tree Construction

```
Create a leaf node for each symbol with its frequency
Insert all nodes into a min-priority queue Q
while |Q| > 1 do
    Remove two nodes x, y with lowest frequencies
    Create new node z with freq(z) = freq(x) + freq(y)
    Set x as left child, y as right child of z
    Insert z into Q
end while
return remaining node as root
```

4.6 Properties

- **Prefix-free**: No code is a prefix of another
- **Optimal**: Minimum expected code length for symbol-by-symbol encoding
- **Average length**: $L = \sum p_i \cdot l_i$ where l_i is code length
- **Bounded**: $H(X) \leq L < H(X) + 1$ (within 1 bit of entropy)

4.7 Applications

- **DEFLATE** (gzip, PNG, ZIP)
- **JPEG** image compression
- **MP3** audio compression
- **Fax machines** (Modified Huffman)

Chapter 5

Arithmetic Coding

5.1 Overview

Arithmetic coding encodes an entire message as a single fractional number between 0 and 1. It can achieve compression very close to the theoretical entropy limit.

- **Inventors**: Jorma Rissanen (IBM), Richard Pasco (1976)
- **Type**: Entropy coding (lossless)
- **Advantage**: Can achieve fractional bits per symbol

5.2 How It Works

The algorithm progressively narrows an interval based on symbol probabilities:

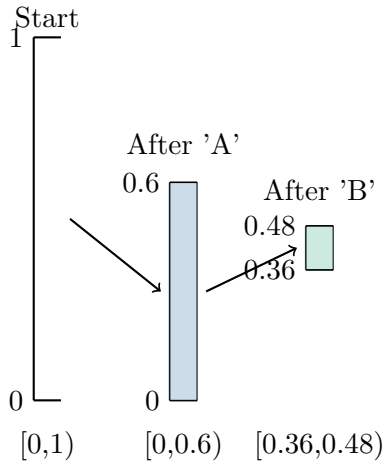


Figure 5.1: Arithmetic coding narrows the interval with each symbol

5.3 Example: Encoding "AB"

Given probabilities: A=60%, B=20%, C=10%, D=10%

1. Start with interval [0, 1)
2. Symbol A: New interval = [0, 0.6)

3. Symbol B: Within [0, 0.6), B occupies [0.6×0.6, 0.6×0.8) = [0.36, 0.48)
4. Output: Any number in [0.36, 0.48), e.g., 0.4

5.4 Comparison with Huffman

Aspect	**Huffman**	**Arithmetic**
Bits per symbol	Integer only	Fractional
Efficiency	Good	Optimal
Complexity	Lower	Higher
Adaptation	Needs rebuild	Easy
Patent status	Free	Historically patented

Table 5.1: Huffman vs Arithmetic coding comparison

5.5 Applications

- **JPEG** (arithmetic mode, rarely used)
- **H.264/AVC** video (CABAC)
- **JPEG XL**
- **Research** applications

Chapter 6

LZ77 and LZ78

6.1 Overview

The **Lempel-Ziv** algorithms are dictionary-based compression methods that form the foundation of most modern compressors.

- **Inventors**: Abraham Lempel, Jacob Ziv
- **LZ77**: Published 1977 (sliding window)
- **LZ78**: Published 1978 (explicit dictionary)
- **IEEE Milestone**: 2004

6.2 LZ77: Sliding Window

LZ77 uses a sliding window to find repeated patterns and replaces them with (offset, length) pairs.

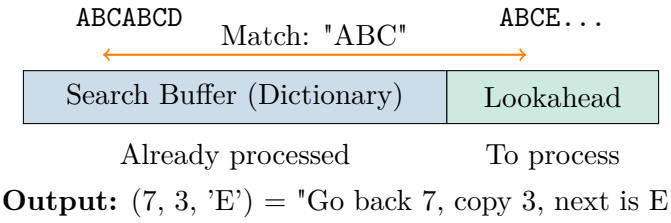


Figure 6.1: LZ77 sliding window compression

6.3 LZ77 Output Format

Each output token contains:

- **Offset**: Distance back to the match
- **Length**: Number of characters to copy
- **Next**: The character following the match

Input	Offset	Length	Next
A	0	0	A
B	0	0	B
ABAB	2	2	A

Table 6.1: LZ77 encoding example for "ABABAB"

6.4 LZ78: Explicit Dictionary

LZ78 builds an explicit dictionary of phrases encountered during compression.

Algorithm 4 LZ78 Encoding

```
Initialize dictionary with empty string at index 0
phrase ← empty
while input not empty do
    Read next symbol c
    if phrase + c is in dictionary then
        phrase ← phrase + c
    else
        Output (index of phrase, c)
        Add phrase + c to dictionary
        phrase ← empty
    end if
end while
```

6.5 Comparison

Aspect	LZ77	LZ78
Dictionary	Implicit (window)	Explicit
Memory	Fixed window size	Grows with input
Decompression	Must start at beginning	Can be random access
Foundation for	DEFLATE, gzip	LZW, GIF

Table 6.2: LZ77 vs LZ78 comparison

Chapter 7

LZW (Lempel-Ziv-Welch)

7.1 Overview

LZW is an improvement of LZ78 that doesn't need to output the next character, making it more efficient.

- **Inventor**: Terry Welch (1984)
- **Type**: Dictionary-based lossless
- **Patent**: Expired 2003 (US), 2004 (international)

7.2 How It Works

1. Initialize dictionary with all single characters
2. Find longest string W in dictionary that matches input
3. Output dictionary index for W
4. Add W + next character to dictionary
5. Repeat from step 2

7.3 Example: Encoding "ABABAB"

Step	Current	Next	Output	Add to Dict
1	A	B	65 (A)	256 = AB
2	B	A	66 (B)	257 = BA
3	AB	A	256 (AB)	258 = ABA
4	AB	-	256 (AB)	-

Table 7.1: LZW encoding steps for "ABABAB"

Initial dictionary: A=65, B=66, ..., Z=90
Output: 65, 66, 256, 256 (4 codes instead of 6 characters)

7.4 Decoding

The decoder rebuilds the dictionary in the same way:

1. Initialize dictionary with single characters
2. Read code, output corresponding string
3. Add previous string + first char of current string to dictionary

7.5 Applications

- **GIF** image format
- **TIFF** (optional compression)
- **PDF** (optional)
- **Unix compress** utility (.Z files)

Chapter 8

DEFLATE

8.1 Overview

DEFLATE combines LZ77 and Huffman coding to achieve good compression with reasonable speed. It's one of the most widely used compression algorithms.

- **Inventor**: Phil Katz (PKZIP, 1990)
- **Standard**: RFC 1951 (1996)
- **Type**: Hybrid (LZ77 + Huffman)

8.2 Algorithm Pipeline

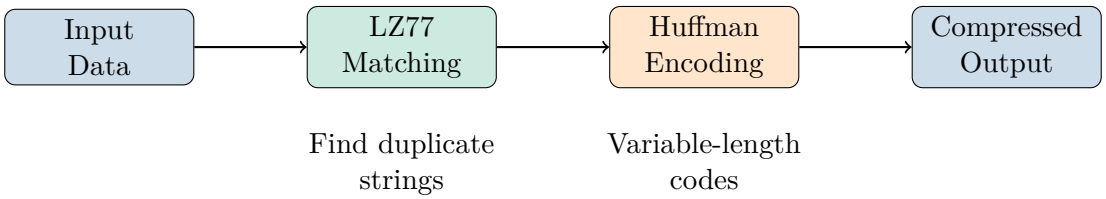


Figure 8.1: DEFLATE compression pipeline

8.3 Step 1: LZ77 Compression

- Window size: 32 KB
- Match length: 3-258 bytes
- Outputs: Literals and (length, distance) pairs

8.4 Step 2: Huffman Coding

Two Huffman trees are used:

1. **Literal/Length tree**: Encodes literals (0-255), end-of-block (256), and lengths (257-285)
2. **Distance tree**: Encodes distances (0-29 with extra bits)

8.5 Block Types

Code	Type	Description
00	Stored	No compression, raw data
01	Static	Pre-defined Huffman trees
10	Dynamic	Custom Huffman trees included
11	Reserved	Not used

Table 8.1: DEFLATE block types

8.6 Compression Levels

Level	Speed	Compression
0	Fastest	None (store only)
1-3	Fast	Low
4-6	Balanced	Medium
7-9	Slow	High

Table 8.2: DEFLATE compression levels

8.7 Applications

- **gzip** (.gz files)
- **ZIP** archives
- **PNG** images
- **HTTP compression**
- **PDF** (FlateDecode)

Chapter 9

Modern Compression Algorithms

9.1 LZ4

LZ4 is an extremely fast compression algorithm focused on speed over compression ratio.

- **Author**: Yann Collet
- **Focus**: Speed
- **Compression**: 500 MB/s
- **Decompression**: 3000 MB/s

Key features:

- No entropy coding (simpler, faster)
- Small dictionary
- Used in: Linux kernel, ZFS, MySQL

9.2 Zstandard (zstd)

Zstandard combines high compression ratios with excellent speed.

- **Author**: Yann Collet (Facebook, 2016)
- **Standard**: RFC 8478
- **Levels**: -7 to 22

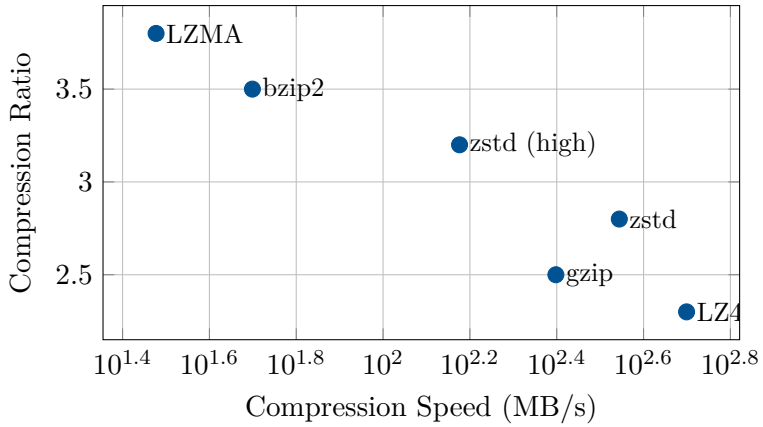


Figure 9.1: Compression algorithms: Speed vs Ratio comparison

9.3 Brotli

Brotli is optimized for web content, offering better compression than gzip.

- **Authors**: Google (Alakuijala, Szabadka, 2013)
- **Standard**: RFC 7932 (2016)
- **Features**: 120 KB static dictionary, context modeling

Web improvements over gzip:

- JavaScript: ∼15% smaller
- HTML: ∼20% smaller
- CSS: ∼16% smaller

9.4 bzip2

bzip2 uses the Burrows-Wheeler Transform for high compression.

Figure 9.2: bzip2 compression pipeline

9.5 LZMA/7z

LZMA achieves the highest compression ratios among common algorithms.

- **Author**: Igor Pavlov (7-Zip, 1998)
- **Features**: Large dictionary (up to 4 GB), range coding
- **Trade-off**: Slow compression, fast decompression

9.6 Snappy

Snappy prioritizes speed for database and big data applications.

- **Author**: Google (2011)
- **Compression**: ∼250 MB/s
- **Decompression**: ∼500 MB/s
- **Used in**: BigTable, MongoDB, Cassandra

Chapter 10

Algorithm Comparison

10.1 Performance Summary

Algorithm	Type	Compress	Decompress	Ratio
LZ4	Dictionary	Very Fast	Very Fast	Low
Snappy	Dictionary	Very Fast	Very Fast	Low
gzip/DEFLATE	Hybrid	Medium	Fast	Medium
Brotli	Hybrid	Medium	Fast	High
zstd	Hybrid	Fast	Fast	High
bzip2	Block-sort	Slow	Medium	High
LZMA/7z	Dictionary	Very Slow	Medium	Very High

Table 10.1: Compression algorithms performance comparison

10.2 Use Case Recommendations

Use Case	Recommended Algorithm
Real-time compression	LZ4, Snappy
Web content	Brotli, zstd
General archiving	zstd, gzip
Maximum compression	LZMA, bzip2
Text files	bzip2, LZMA
Database storage	LZ4, Snappy, zstd
Streaming data	LZ4, zstd

Table 10.2: Algorithm recommendations by use case

10.3 Historical Timeline

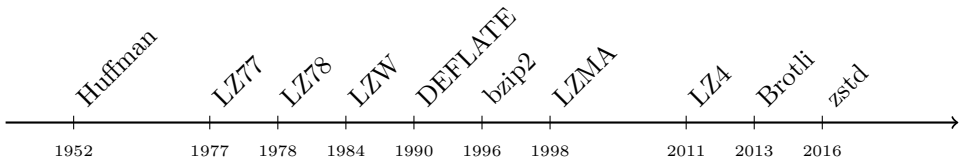


Figure 10.1: Evolution of compression algorithms

Chapter 11

Conclusion

11.1 Key Takeaways

1. **No single best algorithm**: The choice depends on speed, ratio, and use case requirements.
2. **Trade-offs**: Higher compression generally means slower speed.
3. **Modern algorithms**: Zstandard offers the best balance for most applications.
4. **Specialized use cases**: LZ4/Snappy for databases, Brotli for web, LZMA for archives.

11.2 Decision Flowchart

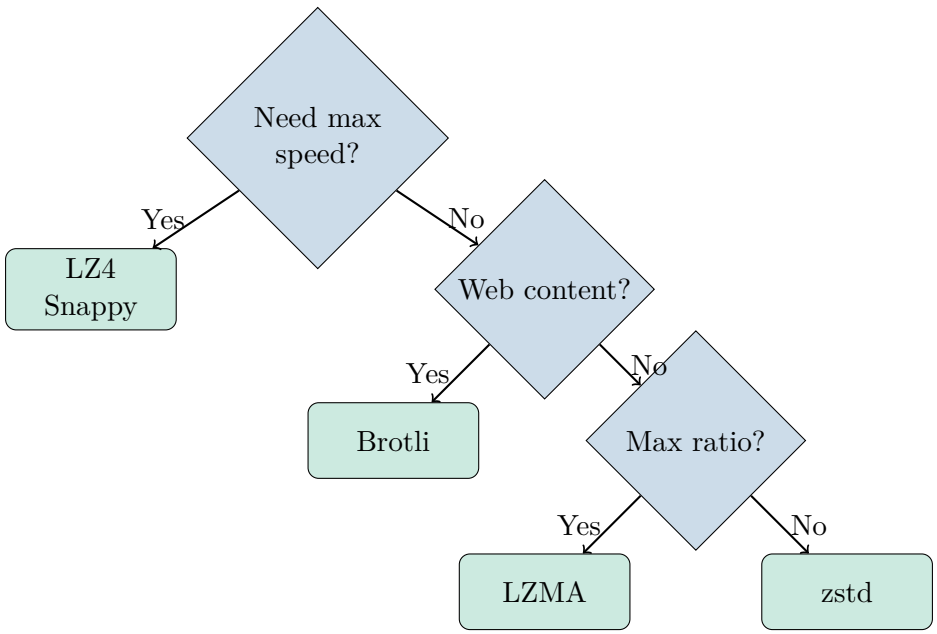


Figure 11.1: Algorithm selection decision tree

11.3 Future Trends

- **Hardware acceleration**: GPU and FPGA-based compression

- **Machine learning**: Neural network-based compression
- **Adaptive algorithms**: Context-aware compression
- **Quantum-resistant**: Compression with post-quantum security

References

1. Huffman, D. A. (1952). "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE*.
2. Ziv, J., Lempel, A. (1977). "A Universal Algorithm for Sequential Data Compression." *IEEE Trans. Information Theory*.
3. Ziv, J., Lempel, A. (1978). "Compression of Individual Sequences via Variable-Rate Coding." *IEEE Trans. Information Theory*.
4. Welch, T. (1984). "A Technique for High-Performance Data Compression." *Computer*.
5. RFC 1951 (1996). "DEFLATE Compressed Data Format Specification."
6. RFC 7932 (2016). "Brotli Compressed Data Format."
7. RFC 8478 (2018). "Zstandard Compression and the application/zstd Media Type."
8. Salomon, D. (2007). *Data Compression: The Complete Reference* (4th ed.). Springer.
9. Seward, J. "bzip2 and libbzip2 Documentation." sourceware.org.
10. Collet, Y. "LZ4 and Zstandard Documentation." github.com/facebook.