Huffman Coding

Optimal Prefix-Free Variable-Length Codes

Data Compression — Class 4

Spring 2026

Today's Roadmap

Part 1: Foundations

- Why variable-length codes?
- Prefix-free property
- Information & Entropy recap

Part 2: Huffman Algorithm

- Greedy construction
- Step-by-step examples (ABRACADABRA, HELLO)
- Optimality & Huffman vs Shannon–Fano

Part 3: Worked Examples

- BANANA, BOOKKEEPER, COCONUT
- Encoding & decoding walkthrough

Part 4: Real-World Applications

- JPEG, PNG, DEFLATE, MP3
- Adaptive & Extended Huffman

Part 5: Practice Problems

- Build trees, compute savings
- Huffman vs Shannon–Fano

Part 1

Foundations

Why not just use ASCII?

Fixed-Length vs Variable-Length Codes

Fixed-Length (e.g. ASCII)

- Every symbol gets the **same** number of bits
- ASCII: 8 bits per character, always
- Simple but wasteful

Example: "AAABBC" in 3-bit fixed code
A=000, B=001, C=010
Encoded: 000 000 000 001 001 010
Total: **18 bits**

Variable-Length

- Frequent symbols → **short** codes
- Rare symbols → **long** codes
- Matches the information content!

Same "AAABBC" with Huffman:
A=0, B=10, C=11
Encoded: 0 0 0 10 10 11
Total: **10 bits** (44% savings!)

Key Insight

Frequent symbols carry **less information** (less surprise). Give them shorter codes. This is the core idea behind Huffman coding.

The Prefix-Free Property: Why It Matters

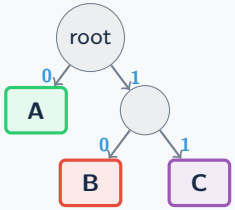
Problem with arbitrary variable-length codes:

Suppose A=0, B=01, C=1.
Bitstream 01 — is it "AB" or "B"?

Ambiguous! Can't decode uniquely.

Prefix-free rule:
No codeword is a **prefix** of another codeword.

Example: A=0, B=10, C=11 ✔
01011 → 0 — 10 — 11 → "ABC" (unique!)



Symbols only at **leaves** → prefix-free

🌲 Binary Tree Connection

Any prefix-free code corresponds to a binary tree. Symbols at leaves, left edge = 0, right edge = 1. Huffman finds the **optimal** such tree.

Quick Recap: Information & Entropy

Shannon Information:

Information of symbol s_i with probability p_i:

$I(s_i) = -\log_2(p_i)$ bits

Entropy (average info):

$H = -\sum_i p_i \log_2(p_i)$ bits/symbol

Entropy is the theoretical minimum average bits per symbol for lossless compression.

Fundamental Limit

No lossless code can achieve average length $< H$. Huffman gets **closest** among symbol-by-symbol codes.

Example: "AABBAC" (6 chars)

Sym	Count	Prob	Info (bits)
A	3	0.5	$-\log_2(0.5) = 1.0$
B	2	$\frac{1}{3}$	$-\log_2(\frac{1}{3}) = 1.58$
C	1	$\frac{1}{6}$	$-\log_2(\frac{1}{6}) = 2.58$

$H = 0.5(1) + 0.33(1.58) + 0.17(2.58)$

$H = \mathbf{1.46}$ bits/symbol

Huffman: A=0, B=10, C=11 $\rightarrow \bar{L} = 1.5$

Part 2

The Huffman Algorithm

A Greedy Approach to Optimal Codes

Huffman Algorithm: The Steps

Algorithm

1. Create a **leaf node** for each symbol
2. Insert all into a **min-heap** (priority queue)
3. **While** queue size > 1:
   a. Remove **two lowest** frequency nodes
   b. Create internal node (freq = sum)
   c. Attach as left/right children
   d. Insert new node back
4. Remaining node = **root**

Why Greedy Works

Least frequent symbols get the **longest** codes (deepest in tree). Merging them first guarantees this. Greedy $\rightarrow$ globally optimal!

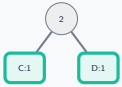
Complexity

$O(n \log n)$ — dominated by heap operations.

Step-by-Step: Building a Huffman Tree

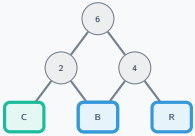
Message: "ABRACADABRA" Frequencies: A=5, B=2, R=2, C=1, D=1

Step 1:
Merge C(1)+D(1)=2

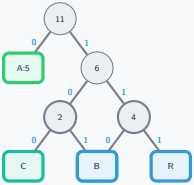


Step 2:
Merge B(2)+R(2)=4

Step 3:
Merge CD(2)+BR(4)=6



Step 4:
Merge A(5)+CDBR(6)=11



title

"ABRACADABRA" $\rightarrow$ 0 110 111 0 100 0 101 0 110 111 0 = **23 bits** vs 33 bits fixed (3-bit) = **30% savings**

Textbook Example: Coding "HELLO"

Message: "HELLO" Freq: L=2, H=1, E=1, O=1

Algorithm 7.1 trace:

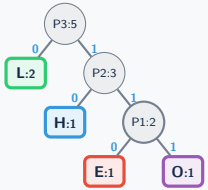
Step	List contents
Init	L(2), H(1), E(1), O(1)
1	Merge E(1)+O(1) → P1(2)
2	Merge H(1)+P1(2) → P2(3)
3	Merge L(2)+P2(3) → P3(5)

Codes: L=0, H=10, E=110, O=111

Avg bits/char: $(1+1+2+3+3)/5 = \mathbf{2.0}$

Fixed 3-bit: $5 \times 3 = 15$ bits

Huffman: $1+1+2+3+3 = \mathbf{10}$ bits (33% saved)



Observation

L appears most (2×) → shortest code (1 bit). E and O are rarest → longest (3 bits). Frequent symbols naturally get shorter codes.

Why Huffman Is Optimal

Optimality Theorem

Among all prefix-free codes, Huffman produces the **minimum average codeword length**.

Key properties (proven in [Huffman 1952]):

1. Two least-frequent symbols have **same-length** codes, differing only at the last bit
2. If $p_i \geq p_j$ then $l_i \leq l_j$ — more frequent $\rightarrow$ shorter code
3. Average code length satisfies: $H \leq \bar{L} < H + 1$

Proof idea: In any optimal tree, the two least-frequent symbols must be siblings at maximum depth. Merging them reduces to a smaller subproblem — induction completes the proof.

How Close to Entropy?

$H \leq \bar{L}_{\text{Huffman}} < H + 1$

Within **1 bit** of entropy. When probabilities are powers of 2, Huffman **achieves H exactly**.

Note

Huffman is optimal per-symbol. **Arithmetic coding** approaches H without the $+1$ gap.

Huffman vs Shannon–Fano

↓ Shannon–Fano (Top-Down)

Divide symbols into two groups of roughly equal total probability. Assign 0/1 to each group. Recurse.

Word: "ABRACADABRA" — A(5), B(2), R(2), C(1), D(1)

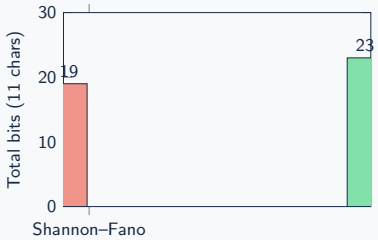
Split: {A}=5 vs {B,R,C,D}=6
Then {B,R}=4 vs {C,D}=2

Result: A=0, B=10, R=11, C=110, D=111
Total: $5(1) + 2(2) + 2(2) + 1(3) + 1(3) = \mathbf{19}$

↑ Huffman (Bottom-Up)

Merge two lowest-frequency nodes repeatedly.

Result: A=0, B=110, R=111, C=100, D=101
Total: $5(1) + 2(3) + 2(3) + 1(3) + 1(3) = \mathbf{23}$



Key Difference

Shannon–Fano is **not guaranteed optimal**. Huffman **always** produces minimum average code length. Ties in the queue can yield different valid trees.

Part 3

Worked Examples

Let's build some trees

Worked Example 1: Coding “BANANA”

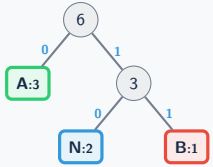
Message: “BANANA” (6 chars)
Frequencies: A=3, N=2, B=1

Sym	Count	Prob	Code	Bits
A	3	0.50	0	1
N	2	0.33	10	2
B	1	0.17	11	2

Merges: B(1)+N(2)=BN(3), then A(3)+BN(3)=root(6)

Average code length:
$\bar{L} = 0.5(1) + 0.33(2) + 0.17(2) = \mathbf{1.5}$ bits/sym

Encode “BANANA”:
11 0 10 0 10 0 = **9 bits**
Fixed 2-bit: $6 \times 2 = 12$ bits. **Saves 25%**



Entropy: $H = 1.459$ bits/sym
$\bar{L} = 1.5$ (within 1 bit of H)

Efficiency: $H/\bar{L} = 97.3\%$

Note

A appears most → gets 1-bit code. B is rarest → gets 2-bit code. Classic Huffman behavior.

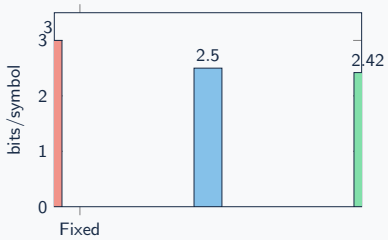
Worked Example 2: Coding "BOOKKEEPER"

Message: "BOOKKEEPER" (10 chars)
Freq: E=2, O=2, K=2, B=1, P=1, R=1

Sym	Count	Code	Bits
E	2	00	2
O	2	01	2
K	2	10	2
B	1	110	3
P	1	1110	4
R	1	1111	4

Merges: P(1)+R(1)=2, B(1)+PR(2)=3, E(2)+O(2)=4, K(2)+BPR(3)=5, EO(4)+KBPR(5)=9

Encode "BOOKKEEPER":
110 01 01 10 10 00 00 1110 00 1111 = **25 bits**
Fixed 3-bit: $10 \times 3 = 30$ bits. **Saves 17%**



Observation

With 6 distinct symbols in 10 characters, the distribution is fairly uniform. Huffman still saves 17%, but the gap to entropy is larger when frequencies are similar.

Worked Example 3: Encoding & Decoding “COCONUT”

Encoding

Message: “COCONUT” (7 chars)

Freq: C=2, O=2, N=1, U=1, T=1

Codes: C=00, O=01, N=10, U=110, T=111

Encode “COCONUT”:

C → 00 O → 01
C → 00 O → 01
N → 10 U → 110
T → 111

Bitstream: 00010001101101111 (**17 bits**)

Fixed 3-bit: $7 \times 3 = 21$ bits. Saves 19%

Decoding

Bitstream: 00010001101101111

Walk tree from root, output at leaf, restart.

0,0 → C 0,1 → O
0,0 → C 0,1 → O
1,0 → N 1,1,0 → U
1,1,1 → T

Result: **COCONUT** ✔

Key Point

Prefix-free → decoding is **unambiguous**. Just walk the tree bit by bit. No lookahead needed.

Part 4

Real-World Applications

Huffman is everywhere

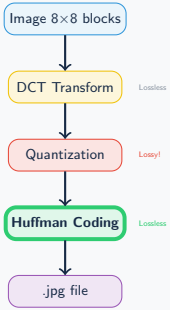
Huffman in JPEG Image Compression

JPEG pipeline (simplified):

1. Split image into 8×8 blocks
2. Apply **DCT** (Discrete Cosine Transform)
3. **Quantize** DCT coefficients (lossy step)
4. **Huffman encode** the quantized values

After quantization, most coefficients are **zero** or small values. Huffman exploits this skewed distribution perfectly.

JPEG uses predefined Huffman tables (Annex K) but also supports custom tables optimized per image.



Huffman Everywhere: PNG, DEFLATE, MP3 & More

PNG

Lossless image format. Uses DEFLATE = LZ77 + Huffman. Prediction filter → DEFLATE compress.

Every PNG you've ever seen uses Huffman.

DEFLATE / ZIP / GZIP

The backbone of web compression. LZ77 finds repeated patterns, then Huffman encodes the output.

HTTP gzip, .zip files, .gz archives.

MP3 / AAC

Audio codecs. Psychoacoustic model removes inaudible frequencies, then Huffman encodes the rest.

MP3 uses custom Huffman tables per frame.

The Big Picture

Huffman coding is a **building block** inside almost every compression format. It's rarely used alone — it's the final entropy coding stage after domain-specific transforms (DCT, LZ77, psychoacoustic models).

Adaptive Huffman & Modern Variants

Problem with Static Huffman

You need to know symbol frequencies **before** encoding. This means:

- Two-pass: scan data, build tree, then encode
- Must transmit the tree/table to the decoder
- Table overhead can negate savings for small files

Adaptive Huffman (FGK / Vitter)

Update the tree **on-the-fly** as symbols arrive. Both encoder and decoder maintain identical trees. Single-pass, no table overhead.

Comparison

	Static	Adaptive
Passes	2	1
Table overhead	Yes	No
Optimality	Optimal	Near-optimal
Complexity	$O(n \log n)$	$O(n \log n)$
Used in	JPEG, MP3	Telecom

→ Modern Alternative

Most modern formats now use **arithmetic coding** or **ANS** instead — closer to entropy. Huffman remains popular due to simplicity.

Extended Huffman Coding

The Integer-Length Problem

Standard Huffman assigns each symbol an **integer** number of bits. But the ideal length $\log_2(1/p_i)$ is rarely an integer.

Example: If $p = 0.9$, ideal $= 0.15$ bits. Huffman must use at least **1 bit** — huge waste!

Solution: Block Symbols

Group n symbols into blocks and build Huffman over all n-tuples. Per-symbol overhead shrinks:

$$H \leq \bar{L}_{\text{block}} < H + \frac{1}{n}$$

As $n \to \infty$, average length $\to$ entropy H.

Trade-off

Alphabet grows **exponentially**: k^n entries. For $k = 4$, $n = 4$: 256 super-symbols. Practical limit on block size.

In Practice

This motivated **arithmetic coding** — it effectively achieves infinite block length without the exponential blowup.

Part 5

Practice Problems

Try before you peek at the solution!

Practice Problem 1: Build a Huffman Tree

Problem

Build a Huffman code for the word "PINEAPPLE" (9 chars).

Symbol	P	E	I	N	A	L
Count	3	2	1	1	1	1

Tasks:

1. Build the Huffman tree step by step
2. Write the Huffman code for each symbol
3. Calculate the average code length $\bar{L}$
4. Calculate entropy H and compare with $\bar{L}$
5. Total bits for "PINEAPPLE": Huffman vs 3-bit fixed?

Solution 1: Huffman Tree for “PINEAPPLE”

Step-by-step merges:

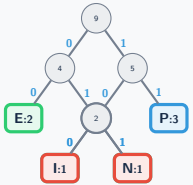
Step	Merge
1	A(1)+L(1) → AL(2)
2	I(1)+N(1) → IN(2)
3	E(2)+AL(2) → EAL(4)
4	IN(2)+P(3) → INP(5)
5	EAL(4)+INP(5) → root(9)

Codes: P=11, E=00, I=100, N=101, A=010, L=011

Avg length:
$\bar{L} = \frac{3(2)+2(2)+1(3)+1(3)+1(3)+1(3)}{9} = \frac{22}{9} = \mathbf{2.44}$

Entropy: $H = 2.28$ bits/sym

“PINEAPPLE”: Huffman = **22** bits
Fixed 3-bit: $9 \times 3 = 27$ bits. Saves 19%



Practice Problem 2: Decode a Bitstream

Problem

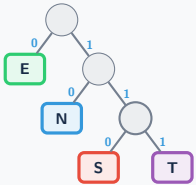
The word "TENNESSEE" gives frequencies: E=4, N=2, S=2, T=1.
A Huffman tree produces codes: E=0, N=10, S=110, T=111.

Tasks:

1. Draw the Huffman tree corresponding to these codes
2. Decode the bitstream: 111010100110011000
3. Calculate $\bar{L}$ and H. How close is Huffman to entropy?
4. Encode "TEEN" and count total bits

Solution 2: Decoding “TENNESSEE”

1. Tree:



2. Decode 111010100110011000:
111—0—10—10—0—110—110—0—0 → **TENNESSEE**

3. Average length & entropy:

$\bar{L} = \frac{4}{9}(1) + \frac{2}{9}(2) + \frac{2}{9}(3) + \frac{1}{9}(3)$
$= 0.44 + 0.44 + 0.67 + 0.33 = \mathbf{1.89}$ bits/sym

$H = 1.748$ bits/sym

✓ Observation

$\bar{L} = 1.89$ vs $H = 1.748$. Huffman is within 0.14 bits of entropy — **92.5% efficient**. Not powers of 2, so a small gap exists.

4. Encode “TEEN”:

111 — 0 — 0 — 10 = **6 bits**

Practice Problem 3: Compression Ratio

Problem

A text file contains 10,000 repetitions of "ABRACADABRA" (110,000 chars total). Character frequencies from the word: A=5, B=2, R=2, C=1, D=1.

Tasks:

1. Build the Huffman tree and assign codes
2. Total bits with Huffman vs fixed-length ($\lceil \log_2 5 \rceil = 3$ bits)
3. Compression ratio (fixed / Huffman)?
4. Entropy H and efficiency $\eta = H/\bar{L}$

Solution 3: Compression Ratio for "ABRACADABRA"

Freq per 11 chars: A=5, B=2, R=2, C=1, D=1

Merges: C(1)+D(1)=2, B(2)+R(2)=4, CD(2)+BR(4)=6, A(5)+CDBR(6)=11

Codes:

Sym	Prob	Code	Len
A	5/11	0	1
B	2/11	110	3
R	2/11	111	3
C	1/11	100	3
D	1/11	101	3

Avg length:

$$\bar{L} = \frac{5(1)+2(3)+2(3)+1(3)+1(3)}{11} = \frac{23}{11} = \mathbf{2.09}$$

Total bits (110,000 chars):

Huffman: $110{,}000 \times 2.09 = \mathbf{229{,}900}$ bits

Fixed: $110{,}000 \times 3 = \mathbf{330{,}000}$ bits

Compression ratio: $330{,}000/229{,}900 = \mathbf{1.44{:}1}$

Saves 100,100 bits (30.3%)

Entropy: $H = 2.04$ bits/sym

Efficiency: $\eta = 2.04/2.09 = \mathbf{97.6\%}$

✔ Takeaway

97.6% efficient — Huffman is very close to the entropy bound for this skewed distribution.

Practice Problem 4: Longer Word Analysis

Problem

Build a Huffman code for "MISSISSIPPI" (11 chars).

Symbol	I	S	P	M
Count	4	4	2	1

Tasks:

1. Build the Huffman tree and assign codes
2. Encode "MISSISSIPPI" and count total bits
3. Fixed-length needs $\lceil \log_2 4 \rceil = 2$ bits/char. Compare savings.
4. If this word repeats 1 million times, how many bytes does Huffman save vs fixed?

Solution 4: "MISSISSIPPI" Analysis

Merges: M(1)+P(2)=MP(3), then I(4)+S(4)=IS(8), then MP(3)+IS(8)=root(11)

Codes:

Sym	Count	Code	Len
I	4	00	2
S	4	01	2
P	2	10	2
M	1	11	2

Encode "MISSISSIPPI":

11 00 01 01 00 01 01 00 10 10 00 = **22 bits**

Fixed 2-bit: $11 \times 2 = 22$ bits

Same! No savings here.

Why no savings?

With 4 symbols and fairly balanced frequencies, all codes end up 2 bits — same as fixed-length.

Entropy: $H = 1.748$ bits/sym

$\bar{L} = 2.0$ bits/sym

Efficiency: 87.4%

✔ Key Lesson

Huffman helps most when frequencies are **highly skewed**. When symbols are roughly equally likely, fixed-length is already near-optimal. The gap $\bar{L} - H = 0.25$ bits is the "integer penalty."

Practice Problem 5: Huffman vs Shannon–Fano

Problem

Consider the word "ENGINEERING" (11 chars).

Symbol	E	N	G	I	R
Count	3	3	2	2	1

Tasks:

1. Build the Huffman tree and assign codes
2. Compute total bits to encode "ENGINEERING" with Huffman
3. Apply Shannon–Fano (top-down splitting) and assign codes
4. Compare total bits: which method wins?
5. Calculate entropy H and compare both methods

Solution 5: "ENGINEERING" — Huffman vs Shannon–Fano

Huffman (Bottom-Up):

Merge: R(1)+G(2)=RG(3), I(2)+RG(3)=IRG(5), E(3)+N(3)=EN(6), IRG(5)+EN(6)=root(11)

Codes: E=10, N=11, G=010, I=00, R=011

Total: $3(2) + 3(2) + 2(3) + 2(2) + 1(3) = \mathbf{25}$ bits

Shannon–Fano (Top-Down):

Split: {E,N}=6 vs {G,I,R}=5

Then E=00, N=01, G=10, I=110, R=111

Total: $3(2) + 3(2) + 2(2) + 2(3) + 1(3) = \mathbf{25}$ bits

Tied! Both give 25 bits.

Entropy:

$H = 2.149$ bits/sym

Avg lengths:

Huffman: $25/11 = \mathbf{2.27}$ bits/sym

Shannon–Fano: $25/11 = \mathbf{2.27}$ bits/sym

Takeaway

When frequencies are fairly balanced, both methods can produce the same result. The difference shows up with **highly skewed** distributions. Huffman is **guaranteed** optimal; Shannon–Fano is not.

Key Takeaways

What You Learned

- Variable-length beats fixed for unequal frequencies
- Prefix-free $\rightarrow$ unique decodability
- Huffman's greedy approach is optimal
- $H \leq \bar{L} < H + 1$
- Inside JPEG, PNG, ZIP, MP3

Common Pitfalls

- Forgetting prefix-free constraint
- Confusing frequency with probability
- Ties in queue $\rightarrow$ multiple valid trees
- Per-symbol only; arithmetic coding is better overall

Next Class

Arithmetic coding — approaching entropy without the $+1$ gap.

"The most frequent symbols
deserve the shortest codes."

David A. Huffman

1952 — "A Method for the Construction
of Minimum-Redundancy Codes"

Questions?