Huffman Coding

Optimal Prefix-Free Data Compression

Mahesh C
FISAT

Federal Institute of Science and Technology (FISAT)
Multimedia Technology Class

February 3, 2026

Outline

Remember Shannon-Fano?

Key Ideas from Last Class:

- Frequent symbols → Short codes
- Rare symbols → Long codes
- Prefix-free for instant decoding
- Top-down divide approach

Shannon-Fano Limitation:

- Not always optimal
- Division may not be perfect
- Can we do better?

The Question

Is there a method that **always** produces the optimal prefix-free code?

Answer: YES!

Huffman Coding (1952)
Guaranteed optimal for symbol-by-symbol coding!

A Simple Puzzle to Start

You have 4 items with weights: 1, 2, 3, 4

How would you pair them to minimize total "cost"?

Option A: Pair heaviest first

- $(4+3) = 7$, then $(7+2) = 9$, then $(9+1) = 10$

Option B: Pair lightest first

- $(1+2) = 3$, then $(3+3) = 6$, then $(6+4) = 10$

A Simple Puzzle to Start

You have 4 items with weights: 1, 2, 3, 4

How would you pair them to minimize total "cost"?

Option A: Pair heaviest first

- $(4+3) = 7$, then $(7+2) = 9$, then $(9+1) = 10$

Option B: Pair lightest first

- $(1+2) = 3$, then $(3+3) = 6$, then $(6+4) = 10$

Huffman's Insight

Always combine the **two smallest** items first!
This greedy approach leads to optimal results.

Who is David Huffman?

The Story (1951):

- MIT graduate student
- Professor Robert Fano's class
- Term paper OR final exam choice
- Paper topic: Find optimal binary codes

The Breakthrough:

- Huffman almost gave up
- Threw away his notes in frustration
- Suddenly realized: **build bottom-up!**
- Proved it was optimal

Fun Fact

Huffman's algorithm beat his professor's own Shannon-Fano method!

Published in 1952, still used today in:

- JPEG images
- MP3 audio
- ZIP files
- DEFLATE

What is Huffman Coding?

Definition

Huffman coding is a **greedy algorithm** that constructs an **optimal prefix-free** binary code by building a tree from the **bottom up**.

Key Properties:

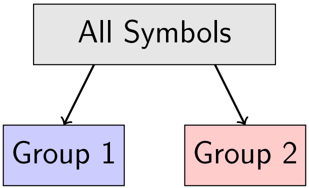
- Optimal — Minimum average code length among all prefix codes
- Prefix-free — No codeword is a prefix of another
- Bottom-up — Start with leaves, build toward root
- **Greedy** — Always merge two smallest probability nodes

Key Difference from Shannon-Fano

Shannon-Fano: **Top-down** (divide) Huffman: **Bottom-up** (merge)

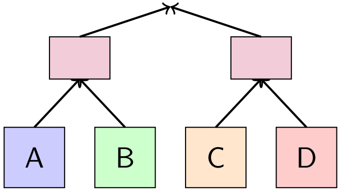
The Core Idea: Build a Tree

Shannon-Fano (Top-Down)



Divide → Assign bits

Huffman (Bottom-Up)



Merge smallest → Build up

The Huffman Algorithm

Algorithm Steps

1. Create a **leaf node** for each symbol with its probability
2. Put all nodes in a **priority queue** (min-heap by probability)
3. While more than one node remains:
    1. Remove the **two nodes** with lowest probability
    2. Create a **new internal node** with these as children
    3. New node's probability $=$ sum of children's probabilities
    4. Add new node back to the queue
4. The remaining node is the **root** of the Huffman tree
5. Assign **0** to left branches, **1** to right branches

Algorithm Pseudocode

Algorithm 1 Huffman Coding

Input: Symbols $S = \{s_1, s_2, \ldots, s_n\}$ with probabilities P
Output: Huffman tree (optimal prefix-free code)

Create leaf node for each symbol
$Q \leftarrow$ priority queue of all nodes (by probability)
while $|Q| > 1$ **do**
    $left \leftarrow \text{ExtractMin}(Q)$
    $right \leftarrow \text{ExtractMin}(Q)$
    Create new node z with children $left$, $right$
    $z.prob \leftarrow left.prob + right.prob$
    $\text{Insert}(Q, z)$
end while
return $\text{ExtractMin}(Q)$ {Root of Huffman tree}

Why Does This Work?

Greedy Choice Property:

Lemma 1

The two symbols with the **lowest probabilities** must have the **longest codes** in any optimal code.

Lemma 2

The two symbols with lowest probabilities can be made **siblings** (same parent) in an optimal tree without increasing average code length.

Intuition:

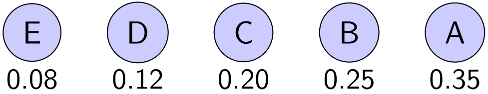
- Rare symbols should be deep in the tree (long codes)
- Merging them first puts them at the bottom
- Their combined probability competes with others
- Process continues optimally

Example: Building a Huffman Tree

Given: Source symbols with probabilities:

Symbol	A	B	C	D	E
Probability	0.35	0.25	0.20	0.12	0.08

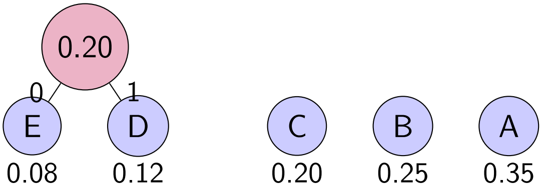
Step 1: Create leaf nodes and put in priority queue



Queue (sorted): E(0.08), D(0.12), C(0.20), B(0.25), A(0.35)

Example: Step 2 — First Merge

Merge two smallest: $E(0.08) + D(0.12) = 0.20$

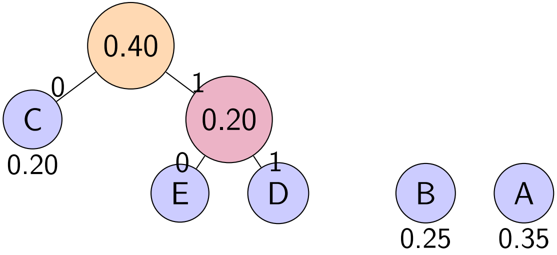


Queue (sorted): C(0.20), [ED](0.20), B(0.25), A(0.35)

Note: When probabilities are equal, either order works!

Example: Step 3 — Second Merge

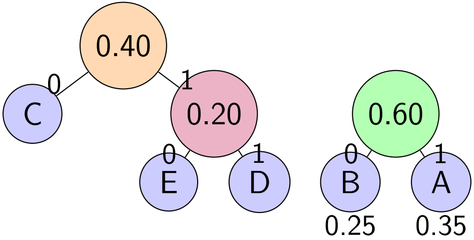
Merge two smallest: C(0.20) + [ED](0.20) = 0.40



Queue (sorted): B(0.25), A(0.35), [CED](0.40)

Example: Step 4 — Third Merge

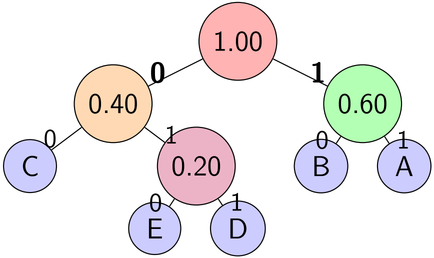
Merge two smallest: B(0.25) + A(0.35) = 0.60



Queue (sorted): [CED](0.40), [BA](0.60)

Example: Step 5 — Final Merge (Root)

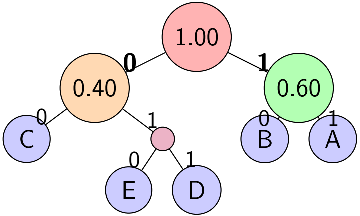
Merge last two: [CED](0.40) $+$ [BA](0.60) $=$ 1.00



Huffman Tree Complete!

Example: Reading the Codes

Traverse from root to each leaf, collecting 0s and 1s:



Final Huffman Codes:

Symbol	Prob	Code	Len
A	0.35	11	2
B	0.25	10	2
C	0.20	00	2
D	0.12	011	3
E	0.08	010	3

Verify prefix-free:

No code is prefix of another ✓

Average Code Length Calculation

Average Code Length

$$L_{avg} = \sum_{i=1}^{n} p_i \cdot l_i \tag{1}$$

For our Huffman code:

Symbol	Probability	Code Length	$p_i \times l_i$
A	0.35	2	0.70
B	0.25	2	0.50
C	0.20	2	0.40
D	0.12	3	0.36
E	0.08	3	0.24
		Total:	**2.20**

Comparison with Entropy

Entropy of the source:

$$\begin{aligned}
H(X) &= -\sum_i p_i \log_2 p_i \\
&= -(0.35 \log_2 0.35 + 0.25 \log_2 0.25 + 0.20 \log_2 0.20 \\
&\quad + 0.12 \log_2 0.12 + 0.08 \log_2 0.08) \\
&= 0.530 + 0.500 + 0.464 + 0.367 + 0.292 \\
&= \boxed{2.153 \text{ bits/symbol}}
\end{aligned}$$

Efficiency

$$\eta = \frac{H(X)}{L_{avg}} \times 100\% = \frac{2.153}{2.20} \times 100\% = \boxed{97.86\%}$$

Redundancy: $R = L_{avg} - H(X) = 2.20 - 2.153 = \boxed{0.047 \text{ bits/symbol}}$

Huffman vs Shannon-Fano: Same Example

Using the same probabilities: A(0.35), B(0.25), C(0.20), D(0.12), E(0.08)

Shannon-Fano Codes:

Symbol	Code	Len
A	00	2
B	01	2
C	10	2
D	110	3
E	111	3

$L_{avg} = 2.20$ bits/symbol

Huffman Codes:

Symbol	Code	Len
A	11	2
B	10	2
C	00	2
D	011	3
E	010	3

$L_{avg} = 2.20$ bits/symbol

Same Result Here!

In this case, both methods achieve the same average code length.
But Huffman is **guaranteed** optimal; Shannon-Fano is not always.

Example Where Huffman Wins

Consider: Probabilities 0.35, 0.17, 0.17, 0.16, 0.15

Shannon-Fano:

Prob	Code	Len
0.35	00	2
0.17	01	2
0.17	10	2
0.16	110	3
0.15	111	3

$L_{avg}^{SF} = 2.31$ bits

Huffman:

Prob	Code	Len
0.35	0	1
0.17	100	3
0.17	101	3
0.16	110	3
0.15	111	3

$L_{avg}^{H} = 2.30$ bits

Huffman Advantage

Huffman gives the most frequent symbol (0.35) a 1-bit code!
Shannon-Fano's top-down division missed this optimization.

Why Huffman is Always Optimal

Optimality Proof Sketch:

1. **Sibling Property:** In any optimal code, two symbols with lowest probabilities can be siblings at maximum depth
2. **Induction:** After merging two lowest-probability symbols:
   - We have a smaller problem (n-1 symbols)
   - The merged node has combined probability
   - Optimal solution for smaller problem $\rightarrow$ optimal for original
3. **Greedy Choice:** Merging smallest first is always safe

Theorem

Huffman coding produces an optimal prefix-free code for any probability distribution.
No other prefix code can have a smaller average length!

Encoding with Huffman Codes

To encode a message:

1. Build Huffman tree from symbol frequencies
2. Replace each symbol with its Huffman code
3. Concatenate all codes

Example: Encode "ABCDE" using our codes

Symbol	Huffman Code
A	11
B	10
C	00
D	011
E	010

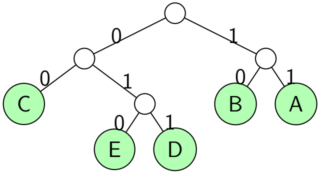
Encoded: A(11) + B(10) + C(00) + D(011) + E(010) = **1110000110010**

13 bits instead of 40 bits (8-bit ASCII) — 67.5% savings!

Decoding with Huffman Tree

To decode: Traverse tree from root, following 0=left, 1=right

Example: Decode "1000011010"



Decoding steps:

- **10** → B
- **00** → C
- **011** → D
- **010** → E

Result: BCDE

Prefix-free property ensures unambiguous decoding!

Exercise 1: Build Huffman Code

Problem: Construct Huffman codes for the following source:

Symbol	Probability
A	0.40
B	0.20
C	0.15
D	0.15
E	0.10

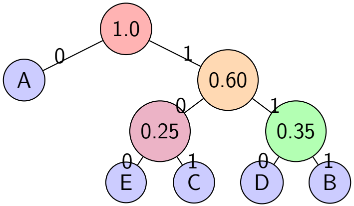
Tasks:

1. Build the Huffman tree step by step
2. Assign codes to each symbol
3. Calculate average code length
4. Calculate entropy and efficiency

Exercise 1: Solution — Building the Tree

Step-by-step merging:

1. Initial: E(0.10), C(0.15), D(0.15), B(0.20), A(0.40)
2. Merge E+C: [EC](0.25), D(0.15), B(0.20), A(0.40)
3. Merge D+B: [EC](0.25), [DB](0.35), A(0.40)
4. Merge [EC]+[DB]: [ECDB](0.60), A(0.40)
5. Merge A+[ECDB]: Root(1.00)



Exercise 1: Solution — Final Codes

Huffman Codes:

Symbol	Probability	Code	Length	$p_i \times l_i$
A	0.40	0	1	0.40
B	0.20	111	3	0.60
C	0.15	101	3	0.45
D	0.15	110	3	0.45
E	0.10	100	3	0.30
			Average Code Length:	**2.20**

Entropy: $H(X) = 2.122$ bits/symbol

Efficiency: $\eta = \frac{2.122}{2.20} \times 100\% = \boxed{96.45\%}$

Redundancy: $R = 2.20 - 2.122 = \boxed{0.078 \text{ bits/symbol}}$

Exercise 2: Text Compression

Problem: Analyze and compress the following text using Huffman coding:

Text (30 characters)

`ABRACADABRA_ABRACADABRA_ABRA`

Tasks:

1. Count frequency of each character
2. Calculate probabilities
3. Build Huffman tree
4. Calculate compression ratio vs 8-bit ASCII

Hint: Count carefully — A appears very frequently!

Exercise 2: Solution

Character Analysis:

Char	Count	Probability
A	12	0.40
B	6	0.20
R	6	0.20
_	2	0.067
C	2	0.067
D	2	0.067
Total	30	1.00

Huffman Codes: A=0, B=10, R=110, _=1110, C=11110, D=11111
Average length: $L_{avg} = 2.27$ bits/char
Compression: $\frac{2.27}{8} = 28.4\%$ of original size!

Exercise 3: Image Pixel Encoding

Problem: A grayscale image has the following pixel value distribution:

Pixel Value	Probability
0 (Black)	0.05
85 (Dark Gray)	0.15
170 (Light Gray)	0.35
255 (White)	0.45

Tasks:

1. Build Huffman codes for pixel values
2. Calculate average bits per pixel
3. Compare with fixed 8-bit encoding
4. Calculate compression ratio

Exercise 3: Solution

Huffman Tree Construction:

1. Merge 0(0.05) + 85(0.15) = [0,85](0.20)
2. Merge [0,85](0.20) + 170(0.35) = [0,85,170](0.55)
3. Merge 255(0.45) + [0,85,170](0.55) = Root(1.00)

Pixel	Prob	Code	Length
255 (White)	0.45	0	1
170 (Light Gray)	0.35	10	2
85 (Dark Gray)	0.15	110	3
0 (Black)	0.05	111	3

$L_{avg} = 0.45(1) + 0.35(2) + 0.15(3) + 0.05(3) = \boxed{1.75 \text{ bits/pixel}}$

Compression: $\frac{1.75}{8} = 21.9\%$ — **78% space saved!**

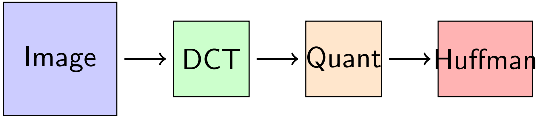
Huffman Coding in JPEG

How JPEG Uses Huffman Coding:
JPEG Pipeline:

1. Color space conversion (RGB $\rightarrow$ YCbCr)
2. Block splitting (8×8 pixels)
3. DCT (Discrete Cosine Transform)
4. Quantization
5. **Huffman Coding**

Why Huffman here?

- After DCT, many coefficients are 0
- Small values are common
- Perfect for variable-length coding!



Result:

- 10 MB photo $\rightarrow$ 500 KB
- 95% compression!

Huffman in ZIP and DEFLATE

DEFLATE Algorithm (used in ZIP, gzip, PNG):

Two-Stage Compression

1. **LZ77:** Find repeated patterns, replace with references
2. **Huffman:** Encode the LZ77 output efficiently

Example:
Original: "ABCABCABC"
After LZ77: "ABC[back 3, len 6]"
After Huffman: Even shorter!

Used in:

- ZIP files
- gzip compression
- PNG images
- HTTP compression
- PDF files

Huffman in MP3 Audio

MP3 Compression Pipeline:

1. **Psychoacoustic Model:** Remove sounds humans can't hear
2. **MDCT:** Transform to frequency domain
3. **Quantization:** Reduce precision
4. **Huffman Coding:** Compress the quantized values

Why Huffman Works Well for Audio

- After quantization, small values dominate
- Zero is the most common value
- Huffman gives short codes to common values
- Result: 10:1 compression with good quality!

CD Quality: 1411 kbps $\rightarrow$ **MP3:** 128-320 kbps

Adaptive Huffman Coding

Problem with Static Huffman:

- Need to know all probabilities beforehand
- Must send code table with compressed data
- Two passes over data required

Adaptive (Dynamic) Huffman

- Build tree as you encode/decode
- Update tree after each symbol
- No need to transmit code table!
- Single pass over data

Algorithms:

- FGK Algorithm (Faller, Gallager, Knuth)
- Vitter's Algorithm (more efficient)

Canonical Huffman Codes

Problem: Different Huffman trees can have same code lengths

Canonical Huffman

Standardized way to assign codes given code lengths:

1. Sort symbols by code length, then alphabetically
2. First code of length L is 0...0 (L zeros)
3. Next code $=$ previous code $+ 1$
4. When length increases, shift left and add 0

Advantages:

- Only need to store code lengths (not full tree)
- Faster decoding with lookup tables
- Used in DEFLATE, JPEG, and many formats

Beyond Huffman: Modern Alternatives

Arithmetic Coding:

- Encodes entire message as one number
- Can achieve fractional bits per symbol
- Closer to entropy than Huffman
- Used in JPEG 2000, H.264

ANS (Asymmetric Numeral Systems):

- Modern alternative (2009)
- Speed of Huffman
- Compression of arithmetic coding
- Used in Zstandard, LZFSE

Comparison:

Method	Speed	Compression
Huffman	Fast	Good
Arithmetic	Slow	Best
ANS	Fast	Best

Huffman Still Relevant!

Simple, fast, patent-free, and "good enough" for many applications.

Key Takeaways

1. **Huffman coding** builds optimal prefix-free codes **bottom-up**
2. **Algorithm:** Repeatedly merge two lowest-probability nodes
3. **Optimality:** Guaranteed minimum average code length

$$H(X) \leq L_{avg} < H(X) + 1$$

4. **vs Shannon-Fano:** Same or better, never worse
5. **Applications:** JPEG, MP3, ZIP, PNG, and more
6. **Variants:** Adaptive, Canonical, Extended Huffman

Remember

Huffman = Greedy + Bottom-up = Optimal!

Important Formulas Summary

Entropy

$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$

Average Code Length

$L_{avg} = \sum_{i=1}^{n} p_i \cdot l_i$

Efficiency

$\eta = \frac{H(X)}{L_{avg}} \times 100\%$

Redundancy

$R = L_{avg} - H(X)$

Compression Ratio

Practice Problems

Problem 1: Build Huffman codes for: P(A)=0.30, P(B)=0.25, P(C)=0.20, P(D)=0.15, P(E)=0.10

Problem 2: A file has 1000 characters with frequencies:

Char	a	b	c	d	e
Count	450	250	150	100	50

Calculate compression ratio vs 8-bit ASCII.

Problem 3: Decode "0110100111" using codes: A=0, B=10, C=110, D=111

Problem 4: Compare Huffman and Shannon-Fano for: 0.4, 0.2, 0.2, 0.1, 0.1

Thank You!

Questions?

"I was able to do this because I didn't know it was supposed to be hard."

— David Huffman