Huffman Coding Algorithm

Optimal Prefix-Free Codes for Data Compression

Bhuvan Kumar and Biprojit Roy

Department of Computer Science and Engineering
IIIT Bhubaneswar

November 21, 2025

Overview

Problem Statement

- Many applications (compression, communication) require efficient encoding of data.
- **Fixed-length** codes (ASCII, 8 bits/symbol) waste space when symbol frequencies are uneven.
- Goal: assign **short codes** to frequent symbols and **long codes** to rare ones.
- Must remain **prefix-free** and **uniquely decodable**.

Motivation

- Real-world text has skewed frequency distribution (e.g., ‘e’ appears more often than ‘z’).
- Fixed-length coding does not exploit this.
- Optimal *lossless* compression requires minimizing average code length:

  $$L_{\text{avg}} = \sum_i f_i \cdot l_i$$

  where f_i is the frequency and l_i is the code length for symbol i.
- How do we construct a prefix-free code with minimal average length?

Challenges

- Maintain **prefix-free** property (no code is prefix of another).
- Ensure **unique decodability**.
- Achieve near-optimal average length, close to entropy:

$$H(X) = -\sum_i p_i \log_2 p_i$$

- Efficient construction for large alphabets.

Introduction to Huffman Coding

- Invented by David Huffman in 1952 while a graduate student at MIT.
- Greedy algorithm for optimal prefix-free binary codes.
- Builds a binary tree bottom-up based on symbol frequencies.
- Final codes obtained by traversing the tree (left $= 0$, right $= 1$).

Fixed-Length vs Variable-Length Coding

Fixed-Length Codes

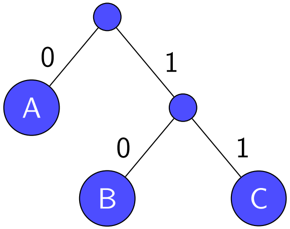
- Same number of bits per symbol.
- Example (2-bit for 4 symbols):
  A: 00, B: 01, C: 10, D: 11
- Simple but inefficient.

Variable-Length Codes

- Short codes for frequent symbols.
- Example:
  A: 0, B: 10, C: 110, D: 111
- Must be prefix-free to avoid ambiguity.
- More efficient compression.

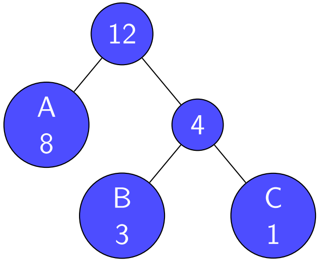
Prefix-Free Codes

- A prefix-free code ensures no codeword is the prefix of another.
- Allows instant and unambiguous decoding.
- Represented naturally using **binary trees** (leaves = symbols).
- Huffman guarantees optimal prefix-free code given frequencies.



Intuition Behind Huffman Coding

- Think of code length as depth of a leaf in a tree.
- Frequent symbols $\rightarrow$ place them closer to root (shorter codes).
- Rare symbols $\rightarrow$ deeper in the tree (longer codes).
- Huffman merges two least-frequent subtrees at each step.

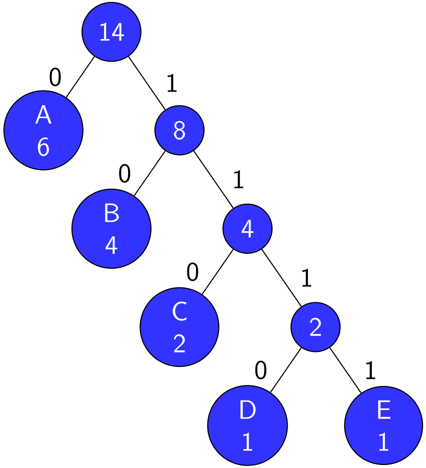


Running Example

Example string: `AAAAAABBBBCCDE`

- Frequencies: A:6, B:4, C:2, D:1, E:1
- Total symbols: 14
- Alphabet size: 5 symbols
- Let's construct the Huffman tree step-by-step.

Huffman Tree Construction (Final)



Resulting Codes: A = 0, B = 10, C = 110, D = 1110, E = 1111

Huffman Coding Algorithm (High-Level)

1. Count frequencies of each symbol.
2. Create a leaf node for each symbol.
3. Insert all nodes into a **min-priority queue** (min-heap).
4. Repeat until only one node remains:
   - Extract two nodes with smallest frequency.
   - Create a new internal node whose frequency is their sum.
   - Insert the new node back into the queue.
5. The remaining node becomes the root of the Huffman tree.

Why Use a Min-Heap?

- We repeatedly need the two smallest-frequency nodes.
- Min-heap provides efficient operations:

$$\text{Extract-Min} = O(\log n), \quad \text{Insert} = O(\log n)$$

- Overall complexity of building the tree:

$$O(n \log n)$$

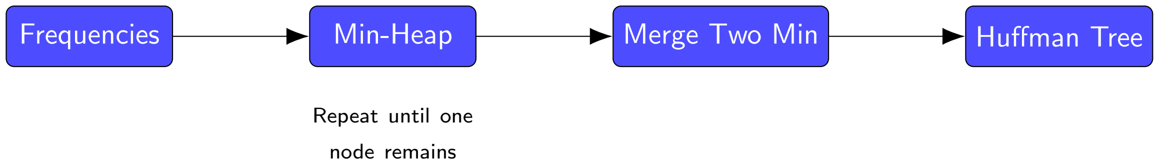
where n is the number of distinct symbols.

Pseudocode (Standard)

- Pseudocode adapted from CLRS:

```
Huffman(C):
1:    Q = MinPriorityQueue(C)
2:    while $|Q| > 1$ do
3:        $x$ = Extract-Min(Q)
4:        $y$ = Extract-Min(Q)
5:        create node $z$ with children $(x, y)$
6:        $z$.freq = $x$.freq + $y$.freq
7:        Insert(Q, z)
8:    return Extract-Min(Q)
```

Algorithm Flow (Diagram)



Why Does Huffman Work? (Proof Overview)

Question: We've seen the algorithm, but why is it guaranteed to be optimal?

Proof Strategy (from CLRS textbook):

1. **Lemma 16.2 (Greedy Choice):** Show that merging the two smallest frequencies first is always safe - it doesn't prevent us from reaching an optimal solution.
2. **Lemma 16.3 (Optimal Substructure):** Show that if we solve the smaller subproblem optimally, we get an optimal solution to the original problem.
3. **Theorem 16.4 (Optimality):** Combine the two lemmas using induction to prove Huffman always produces optimal codes.

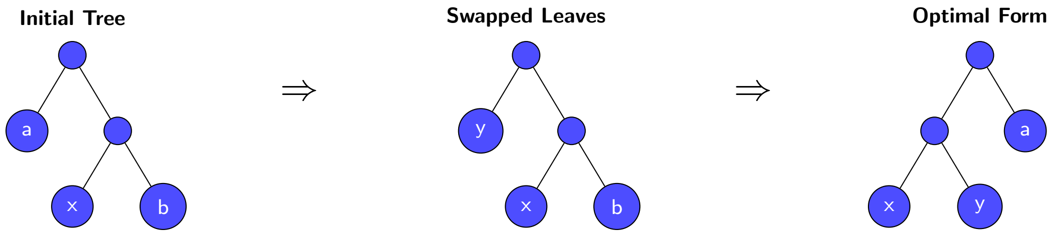
Lemma 16.2 (Greedy Choice)

Statement: Let x and y be the two symbols of minimum frequency. There exists an optimal prefix-free code in which x and y are siblings at maximum depth.

Key intuition:

- Leaves x and y appear at deepest level.
- Swapping positions of leaves never increases total cost.
- Therefore we may "lock" them as siblings without loss of optimality.

Lemma 16.2 — Illustration



Swapping lower-frequency nodes never increases cost — so we can assume x, y are siblings.

Lemma 16.3 (Optimal Substructure)

Statement: Let x and y be the two least-frequent symbols. Let z be a new combined symbol with:

$$f(z) = f(x) + f(y)$$

If T' is an optimal tree for the reduced alphabet (including z), then expanding z back into x and y gives an optimal tree for the original alphabet.

Cost relation:

$$B(T) = B(T') + f(x) + f(y)$$

This demonstrates optimal substructure property.

Theorem 16.4 (Optimality of Huffman Coding)

Theorem: Huffman's algorithm produces an optimal prefix-free code for a given set of symbol frequencies.

Proof Sketch:

- Lemma 16.2 justifies merging two least frequent symbols first (greedy choice).
- Lemma 16.3 gives optimal substructure.
- Combine via induction on alphabet size.

Entropy Bound:

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

Huffman coding is within 1 bit of theoretical entropy limit.

Full Example: Step-by-step (Encoding)

Example string: `AAAAAABBBBCCDE`

Frequencies: A:6, B:4, C:2, D:1, E:1

Stepwise merges (building from bottom-up):

1. Merge D (1) and E (1) $\rightarrow$ node DE (2)
2. Merge C (2) and DE (2) $\rightarrow$ node CDE (4)
3. Merge B (4) and CDE (4) $\rightarrow$ node BCDE (8)
4. Merge A (6) and BCDE (8) $\rightarrow$ root (14)

Final codes (from tree): A $=$ 0, B $=$ 10, C $=$ 110, D $=$ 1110, E $=$ 1111

Encoding: Bit Calculation

Comparison 1: Optimal Fixed-Length for 5 symbols

- Fixed-length needs $\lceil \log_2 5 \rceil = 3$ bits/symbol: $14 \times 3 =$ **42 bits**
- Huffman: $(6 \times 1) + (4 \times 2) + (2 \times 3) + (1 \times 4) + (1 \times 4) =$ **28 bits**
- **Savings: 14 bits (33.3% compression)**

Comparison 2: Real-world ASCII (8 bits/char)

- Without compression: $14 \times 8 =$ **112 bits**
- With Huffman: 28 bits (data) + $\sim$40 bits (tree overhead) =
- **Savings: $\sim$44 bits (39% compression)**

Note: Tree overhead becomes negligible for larger files. For a 1KB file, the $\sim$40 bit overhead is only 0.5% of total size!

Decoding (Decompression)

Decoding procedure:

1. Receiver must have the Huffman tree (or code table) transmitted in header.
2. Start at root, read bits left-to-right from encoded stream.
3. Bit 0 $\rightarrow$ go left; Bit 1 $\rightarrow$ go right.
4. On reaching a leaf, output the symbol and return to root.
5. Repeat until all bits are consumed.

Why it is unambiguous: The prefix-free property ensures no codeword is a prefix of another, guaranteeing unique decodability.

Time Complexity

- Let n be the number of distinct symbols (alphabet size).
- Initial heap build: $O(n)$ using heapify method.
- Each of the $(n-1)$ merges performs:
  - Two Extract-Min operations: $O(\log n)$ each
  - One Insert operation: $O(\log n)$

$$O((n-1) \cdot 3 \log n) = O(n \log n)$$

- **Total time complexity:** $O(n \log n)$.
- If frequencies come from input text of length m, counting frequencies is $O(m)$, so overall $O(m + n \log n)$.

Space Complexity

- We create one leaf node per symbol plus $(n-1)$ internal nodes: $2n-1=O(n)$ nodes total.
- Min-heap holds at most n nodes at any time: $O(n)$ space.
- Storing code table: $O(n \cdot \bar{l})$ where $\bar{l}$ is average code length.
- Since $\bar{l} \leq \lceil \log_2 n \rceil$ typically, code table is $O(n \log n)$ in worst case.
- **Overall space complexity:** $O(n)$ to $O(n \log n)$.

Applications

- **File compression:** ZIP and GZIP files use Huffman as part of their compression algorithm.
- **Image formats:** PNG images use Huffman coding to reduce file size without losing quality.
- **Audio/Video:** JPEG images and MP3 audio use Huffman in their compression pipeline.
- **Web protocols:** HTTP/2 uses Huffman to compress web page headers, making websites load faster.
- **Any scenario:** Where you need lossless compression and know the symbol frequencies in advance.

Bottom line: Huffman coding is everywhere - in the files you download, images you view, and websites you visit!

Advantages

- Produces **provably optimal** prefix-free code for known symbol frequencies.
- Simple greedy algorithm; easy to understand and implement.
- Codes are near-entropy optimal: $H(X) \leq L_{\text{avg}} < H(X) + 1$.
- Widely used and proven effective in practice for decades.
- No loss of information (lossless compression).

Limitations

- Requires two passes for static Huffman:
  1. First pass to count frequencies
  2. Second pass to encode using generated codes
- Not adaptive to changing distributions during encoding.
- **Overhead:** The tree or code table must be transmitted/stored with compressed data.
- For very small files, the overhead can reduce overall compression benefit.
- Limited to integer code lengths (cannot achieve fractional bits per symbol).

Conclusion

- Huffman coding is a fundamental greedy algorithm for lossless data compression.
- It constructs optimal prefix-free codes when symbol frequencies are known a priori.
- Clear theoretical guarantees: optimality and near-entropy bound ($L < H + 1$).
- Practical utility demonstrated in widely-used formats: PNG, ZIP, JPEG, MP3.
- Remains relevant 70+ years after its invention as a cornerstone of information theory and compression technology.

References

- D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the IRE*, 1952.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (CLRS), 4th Edition, Chapter 16.
- T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd Edition.
- RFC 1951: DEFLATE Compressed Data Format Specification.
- PNG Specification: `https://www.w3.org/TR/PNG/`

Questions & Discussion

Thank you for your attention!

Questions?

Contact:
Bhuvan Kumar & Biprojit Roy
Department of CSE, IIIT Bhubaneswar