

## Huffman Coding

### 1. What is Huffman Coding?

**Huffman Coding** is a **greedy algorithm** used for **lossless data compression**, where **variable-length binary codes** are assigned to characters based on their **frequency of occurrence**.

- Frequently occurring characters → **shorter codes**
- Less frequent characters → **longer codes**

### 2. Key Idea

“Use fewer bits for frequent symbols and more bits for rare symbols.”

This reduces the **average number of bits per character**.

### 3. Properties of Huffman Codes

- **Prefix-free code** (no code is a prefix of another)
- **Optimal** among all prefix codes
- **Lossless compression**
- Based on **binary trees**

### 4. Steps of Huffman Coding Algorithm

1. List all characters with their frequencies.
2. Create a **leaf node** for each character.
3. Insert all nodes into a **min-priority queue** (based on frequency).
4. Repeat until only one node remains:
  - Remove two nodes with the **smallest frequency**
  - Create a new internal node with frequency = sum of both
  - Insert the new node back into the queue
5. Assign:
  - **0** → **left edge**
  - **1** → **right edge**
6. Generate codes by traversing the tree.

## 5. Huffman Tree

- **Leaf nodes** → characters
- **Internal nodes** → combined frequencies
- Root represents the total frequency

### Given Data (Frequencies / Probabilities)

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

### Step 1: Arrange in Ascending Order

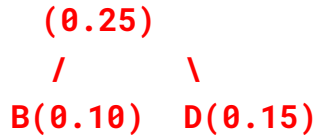
Character	Probability
B	0.10
D	0.15
E	0.15
C	0.20
A	0.40

## Step 2: Build the Huffman Tree (Greedy Steps)

### Step 2.1

Combine two smallest:

- $B(0.10) + D(0.15) = 0.25$



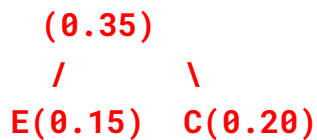
### Step 2.2

Remaining nodes:

- $E(0.15), C(0.20), A(0.40), BD(0.25)$

Combine:

- $E(0.15) + C(0.20) = 0.35$



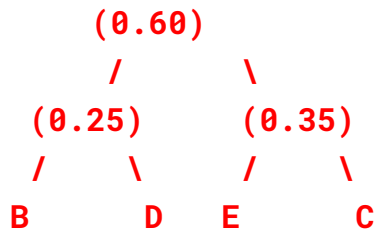
### Step 2.3

Remaining nodes:

- $BD(0.25), EC(0.35), A(0.40)$

Combine:

- $BD(0.25) + EC(0.35) = 0.60$



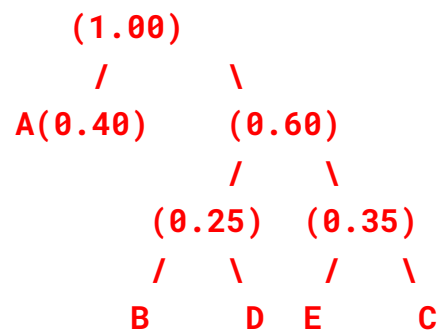
### Step 2.4 (Final Step)

Remaining nodes:

- A (0.40), BDEC (0.60)

Combine:

- A (0.40) + BDEC (0.60) = **1.00 (Root)**



### Step 3: Assign Binary Codes

(Left = 0, Right = 1)

Character	Huffman Code
A	0
B	100
D	101
E	110
C	111

**Step 4: Encode ABACABAD :**

**0100011101000101**

**Step 5: Decode 100010111001010**

Using the Huffman codes:

**100 → B**

**0 → A**

**101 → D**

**110 → E**

**0 → A**

**101 → D**

**0 → A**

**Decoded String**

**BADEADA**

**Step 6: Average Bits per Character (Compressed)**

Character	Probability	Code Length
A	0.40	1
B	0.10	3
C	0.20	3
D	0.15	3
E	0.15	3

$$\begin{aligned}\text{Profit/ Average bits} &= (0.40 \times 1) + (0.10 \times 3) + (0.20 \times 3) + (0.15 \times 3) + (0.15 \times 3) \\ &= 0.4 + 0.3 + 0.6 + 0.45 + 0.45 \\ &= 2.20 \text{ bits/character}\end{aligned}$$

## Compression Percentage Calculation

The compression percentage is calculated using the formula:

$$\text{Compression (\%)} = \frac{\text{Max bits} - \text{Average bits}}{\text{Max bits}} \times 100$$

Substituting the given values:

$$\begin{aligned} \text{Compression (\%)} &= \frac{3 - 2.2}{3} \times 100 \\ &= \frac{0.8}{3} \times 100 \\ &= 26.67\% \end{aligned}$$

Hence, the compression achieved using Huffman coding is **26.67%**.

### Question

The probabilities of occurrence of characters in a text are given below:

Character	A	B	C	D	E	F
Probability	0.05	0.09	0.12	0.13	0.16	0.45

- Construct the Huffman tree for the given characters, showing all intermediate steps.
- Assign binary Huffman codes to each character.
- Calculate the average number of bits per character.
- Assuming fixed-length coding, compute the compression ratio or compression percentage achieved using Huffman coding.

## Advantages of Huffman Coding

- **Lossless compression** (original data can be perfectly recovered)
- **Optimal prefix code** (minimum average code length).
- **No ambiguity** due to prefix-free property
- **Simple and efficient** to implement

## Range of Compression Ratio (Huffman Coding)

The compression achieved using Huffman coding generally ranges from **20% to 80%**. Higher compression is obtained when symbol frequencies are highly non-uniform, while lower compression occurs when symbol probabilities are nearly uniform.

## Disadvantages of Huffman Coding

- Requires **prior knowledge of symbol frequencies**
- **Huffman tree must be stored or transmitted** along with data
- Not efficient when **frequencies are nearly equal**
- Works on **symbols only**, not on higher-level patterns
- Compression performance is **data-dependent**