



Algorithms: Design  
and Analysis, Part II

# Huffman Codes

---

Introduction and  
Motivation

# Binary Codes

**Binary code:** Maps each character of an alphabet  $\Sigma$  to a binary string.

**Example:**  $\Sigma = \text{a-z and various punctuation}$  (size 32 overall, say)

**Obvious encoding:** Use the 32 5-bit binary strings to encode this  $\Sigma$  (a fixed-length code)

**Can we do better?** Yes, if some characters of  $\Sigma$  are much more frequent than others, using a variable-length code.

# Ambiguity

**Example:** Suppose  $\Sigma = \{A,B,C,D\}$ . Fixed-length encoding would be  $\{00,01,10,11\}$ .

Suppose instead we use the encoding  $\{0,01,10,1\}$ . What is 001 an encoding of?

- A) AB  $\rightarrow$  Leads to 001
- B) CD
- C) AAD  $\rightarrow$  Also leads to 001
- D) Not enough info to answer question

# Prefix-Free Codes

**Problem:** With variable-length codes, not clear where one character ends + the next one begins.

**Solution:** Prefix-free codes - make sure that for every pair  $i, j \in \Sigma$ , neither of the encodings  $f(i), f(j)$  is a prefix of the other.

**Example:**  $\{0, 10, 110, 111\}$

**Why useful?** Can give shorter encodings with non-uniform character frequencies.

# Example

Example:

A	60%	00	0
B	25%	01	10
C	10%	10	110
D	5%	11	111

---

$\Sigma$	frequencies	fixed-length	variable-length (prefix free)
----------	-------------	--------------	----------------------------------

Fixed-length encoding: 2 bits/character

Variable-length encoding: How many bits needed on average?

A) 1.5    B) 1.55    C) 2    D) 2.5

$$0.6 \cdot 1 + 0.25 \cdot 2 + (0.1 + 0.05) \cdot 3 = 1.55$$



# Huffman Codes

---

## Problem Definition

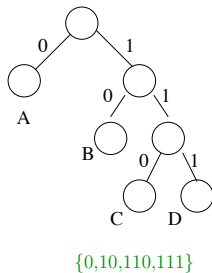
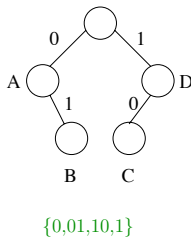
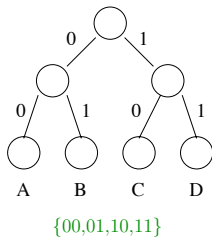
Algorithms: Design  
and Analysis, Part II

# Codes as Trees

**Goal:** Best binary prefix-free encoding for a given set of character frequencies.

**Useful fact:** Binary codes  $\leftrightarrow$  Binary trees

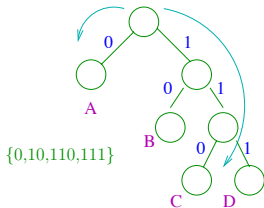
**Examples:** ( $\Sigma = \{A,B,C,D\}$ )



# Prefix-Free Codes as Trees

- In general:**
- Left child edges  $\leftrightarrow$  "0", right child edges  $\leftrightarrow$  "1"
  - For each  $i \in \Sigma$ , exactly one node labeled "i"
  - Encoding of  $i \in \Sigma \leftrightarrow$  Bits along path from node to the node "i"
  - Prefix-free  $\leftrightarrow$  Labelled nodes = the leaves
- [since prefixes  $\leftrightarrow$  one node an ancestor of another]

**To decode:** Repeatedly follow path from root until you hit a leaf.  
[ex. 0110111  $\mapsto$  ACD] (unambiguous since only leaves are labelled)



**Note:** Encoding length of  $i \in \Sigma =$  depth of  $i$  in tree.

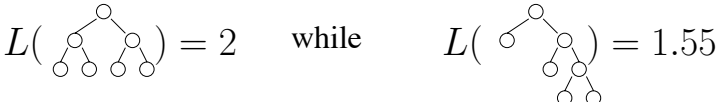


# Problem Definition

**Input:** Probability  $p_i$  for each character  $i \in \Sigma$ .

**Notation:** If  $T$  = tree with leaves  $\leftrightarrow$  symbols of  $\Sigma$ , then **average encoding length**  $L(T) = \sum_{i \in \Sigma} p_i \cdot [\text{depth of } i \text{ in } T]$

**Example:** If  $p_A = 60\%$ ,  $p_B = 25\%$ ,  $p_C = 10\%$ ,  $p_D = 5\%$ , then

$$L(\text{balanced tree}) = 2 \quad \text{while} \quad L(\text{unbalanced tree}) = 1.55$$


**Output:** A binary tree  $T$  minimizing the average encoding length  $L(\cdot)$ .



# Huffman Codes

---

## A Greedy Algorithm

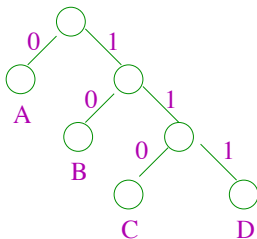
Algorithms: Design  
and Analysis, Part II

# Codes as Trees

**Input:** Probability  $p_i$  for each character  $i \in \Sigma$ .

**Output:** Binary tree (with leaves  $\leftrightarrow$  symbols of  $\Sigma$ ) minimizing the average encoding length:

$$L(T) = \sum_{i \in \Sigma} p_i [\text{depth of } i \text{ in } T]$$

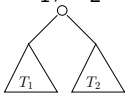


# Building a Tree

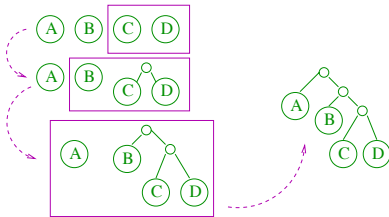
**Question:** What's a principled approach for building a tree with leaves  $\leftrightarrow$  symbols of  $\Sigma$ ?

**Natural but suboptimal idea:** Top-down/divide+conquer.

- Partition  $\Sigma$  into  $\Sigma_1, \Sigma_2$  each with  $\approx 50\%$  of total frequency.
- Recursively compute  $T_1$  for  $\Sigma_1$ ,  $T_2$  for  $\Sigma_2$ , return:



**Huffman's (optimal) idea:** Build tree bottom-up using successive mergers.

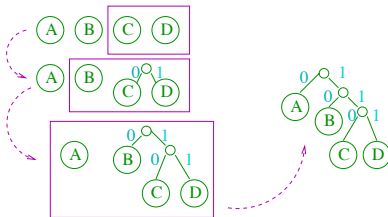


# A Greedy Approach

**Question:** Which pair of symbols is “safe” to merge?

**Observation:** Final encoding length of  $i \in \Sigma = \#$  of mergers its subtree endures.

[Each merger increases encoding length of participating symbols by 1]



**Greedy heuristic:** In first iteration, merge the two symbols with the smallest frequencies.

# How to Recurse?

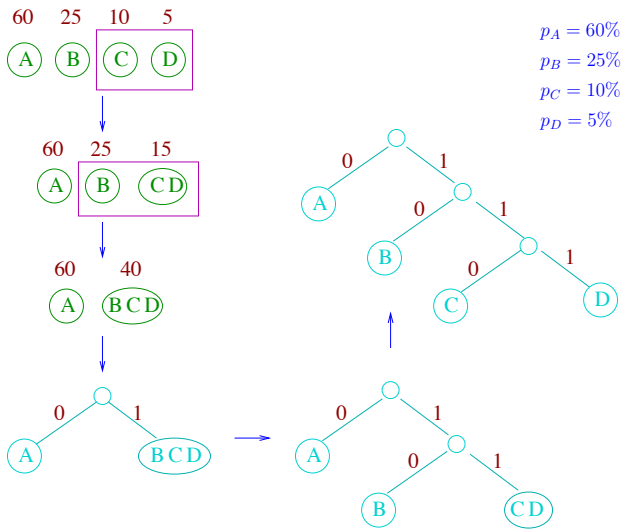
**Suppose:** 1st iteration of algorithm merges symbols  $a$  &  $b$ .

**Idea:** Replace symbols  $a, b$  by a new “meta-symbol”  $ab$ .

**Question:** What should be the frequency  $p_{ab}$  of this meta-symbol?

- A)  $\max\{p_a, p_b\}$
- B)  $\min\{p_a, p_b\}$
- C)  $p_a + p_b$  since  $ab$  is a proxy for “ $a$  or  $b$ ” (intuitively)
- D)  $p_a - p_b$

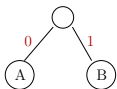
# Example



# Huffman's Algorithm

(Given frequencies  $p_i$  as input)

If  $|\Sigma| = 2$  return



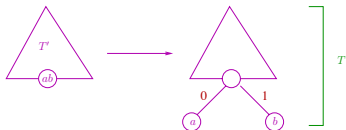
Let  $a, b \in \Sigma$  have the smallest frequencies.

Let  $\Sigma' = \Sigma$  with  $a, b$  replaced by new symbol  $ab$ .

Define  $p_{ab} = p_a + p_b$ .

Recursively compute  $T'$  (for the alphabet  $\Sigma'$ )

Extend  $T'$  (with leaves  $\leftrightarrow \Sigma'$ ) to a tree  $T$  with leaves  $\leftrightarrow \Sigma$  by splitting leaf  $ab$  into two leaves  $a$  &  $b$ .



Return  $T$





Algorithms: Design  
and Analysis, Part II

# Huffman Codes

---

Huffman's Algorithm: A  
More Complex Example

# Input and Steps 1 and 2

**Input:**

Characters	A	B	C	D	E	F
Weights	3	2	6	8	2	6

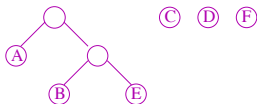
**Step 1:** Merge B and E:

A	BE	C	D	F
3	4	6	8	6



**Step 2:** Merge A and BE:

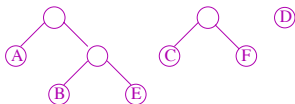
ABE	C	D	F
7	6	8	6



## Steps 3 and 4

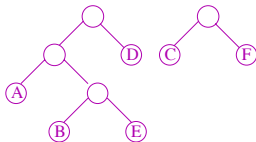
Step 3: Merge C and F:

ABE	CF	D
7	12	8



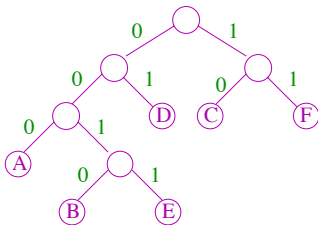
Step 2: Merge ABE and D:

ABDE	CF
15	12



# Final Output

Final tree:



Corresponding code:

A	000	D	01
B	0010	E	0011
C	10	F	11



# Huffman Codes

---

## Correctness Proof

Algorithms: Design  
and Analysis, Part II

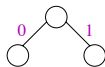
# Correctness of Huffman's Algorithm

**Theorem:** [Huffman 52] Huffman's algorithm computes a binary tree (with leaves  $\leftrightarrow$  symbols of  $\Sigma$ ) that minimizes the average encoding length

$$L(T) = \sum_{i \in \Sigma} p_i [\text{depth of leaf } i \text{ in } T].$$

**Proof:** By induction on  $n = |\Sigma|$ . (Can assume  $n \geq 2$ .)

**Base case:** When  $n = 2$ , algorithm outputs the optimal tree.  
(Needs 1 bit per symbol)



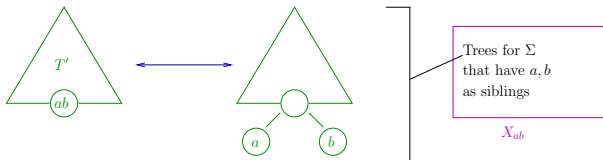
**Inductive step:** Fix input with  $n = |\Sigma| > 2$ .

**By inductive hypothesis:** Algorithm solves smaller subproblems (for  $\Sigma'$ ) optimally.

# Inductive Step

Let  $\Sigma' = \Sigma$  with  $a, b$  (symbols with smallest frequencies) replaced by meta-symbol  $ab$ . Define  $p_{ab} = p_a + p_b$ .

**Recall:** Exact correspondence between:



**Important:** For every such pair  $T'$  and  $T$ ,  $L(T) - L(T')$  is (after cancellation)

$$p_a [a's \text{ depth in } T] + p_b [b's \text{ depth in } T] - p_{ab} [ab's \text{ depth in } T'] =$$

Each is one more than

$$= p_a(d+1) + p_b(d+1) - (p_a + p_b)d = p_a + p_b, \text{ Independent of } T, T'!$$

# Proof of Theorem

**Inductive hypothesis:** Huffman's algorithm computes a tree  $\hat{T}'$  that minimizes  $L(T')$  for  $\Sigma'$ .

**Upshot of last slide:** Corresponding tree  $\hat{T}$  minimizes  $L(T)$  for  $\Sigma$  over all trees in  $X_{ab}$  (i.e., where  $a$  &  $b$  are siblings)

**Key lemma:** [Completes proof of theorem] There is an optimal tree (for  $\Sigma$ ) in  $X_{ab}$ . [i.e.,  $a$  &  $b$  were “safe” to merge]

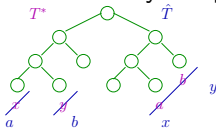
**Intuition:** Can make an optimal tree better by pushing  $a$  &  $b$  as deep as possible (since  $a, b$  have smallest frequencies).



# Proof of Key Lemma

By exchange argument. Let  $T^*$  be any tree that minimizes  $L(T)$  for  $\Sigma$ . Let  $x, y$  be siblings at the deepest level of  $T^*$ .

**The exchange:** Obtain  $\hat{T}$  from  $T^*$  by swapping  $a \leftrightarrow x$ ,  $b \leftrightarrow y$



**Note:**  $\hat{T} \in X_{ab}$  (by choice of  $x, y$ ).

**To finish:** Will show that  $L(\hat{T}) \leq L(T^*)$

$\Rightarrow \hat{T}$  also optimal, completes proof]

**Reason:**

$$\begin{aligned}
 L(T^*) - L(\hat{T}) &= (p_x - p_a) \quad [x\text{'s depth in } T^* - a\text{'s depth in } T^*] \\
 &+ (p_y - p_b) \quad [y\text{'s depth in } T^* - b\text{'s depth in } T^*] \\
 &\geq 0 \quad \text{QED!}
 \end{aligned}$$

$\geq 0$  since  $a, b$  have smallest frequencies

$\geq 0$  by choice of  $x, y$

# Notes on Running Time

**Naive implementation:**  $O(n^2)$  time, where  $n = |\Sigma|$ .

**Speed ups:** - Use a heap! [to perform repeated minimum computations]

- Use keys = frequencies
  - After extracting the two smallest-frequency symbols, re-Insert the new meta-symbol [new key = sum of the 2 old ones]
- ⇒ Iterative,  $O(n \log n)$  implementation.

**Even faster:** (Non-trivial exercise) Sorting +  $O(n)$  additional work.

- Manage (meta-)symbols using two queues.