CS 7800: Algorithms & Data

Lecture 23: Data Compression
- Huffman Codes

Jonathan Ullman
12-06-2022

Data Compression

- How do we store strings of text compactly?

- A binary code is a mapping from $\Sigma \to \{0,1\}^*$
  - Simplest code: assign numbers $1,2,\ldots,|\Sigma|$ to each symbol, map to binary numbers of $\lceil \log_2 |\Sigma| \rceil$ bits
  - Morse Code:

A •–	J •–––	S •••
B –•••	K –•–	T –
C –•–•	L •–••	U ••–
D –••	M ––	V •••–
E •	N –•	W •––
F ••–•	O –––	X –••–
G ––•	P •––•	Y –•––
H ••••	Q ––•–	Z ––••
I ••	R •–•	

$2+4+8+16 = 30$

Data Compression

- Letters have uneven frequencies!
  - Want to use short encodings for frequent letters, long encodings for infrequent leters

	a	b	c	d	avg. len.
Frequency	1/2	1/4	1/8	1/8	
Encoding 1	00	01	10	11	**2.0**
Encoding 2	0	10	110	111	**1.75**

Data Compression

- What properties would a good code have?
  - Easy to encode a string

    Encode(KTS) = – ● – – ● ● ●

  - The encoding is short on average

    ≤ 4 bits per letter (30 symbols max!)

  - Easy to decode a string?

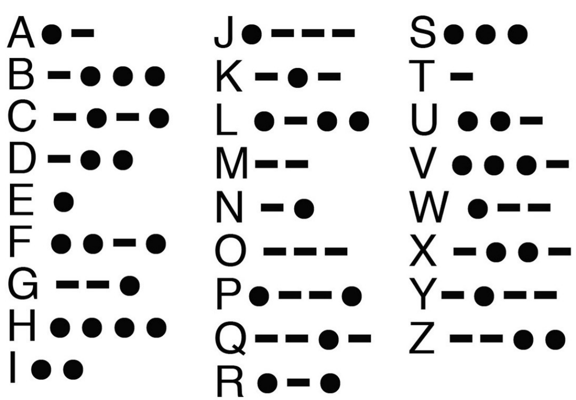
    Decode(– ● –|–|● ● ●) =

A ● –	J ● – – –	S ● ● ●
B – ● ● ●	K – ● –	T –
C – ● – ●	L ● – ● ●	U ● ● –
D – ● ●	M – –	V ● ● ● –
E ●	N – ●	W ● – –
F ● ● – ●	O – – –	X – ● ● –
G – – ●	P ● – – ●	Y – ● – –
H ● ● ● ●	Q – – ● –	Z – – ● ●
I ● ●	R ● – ●	

Prefix Free Codes

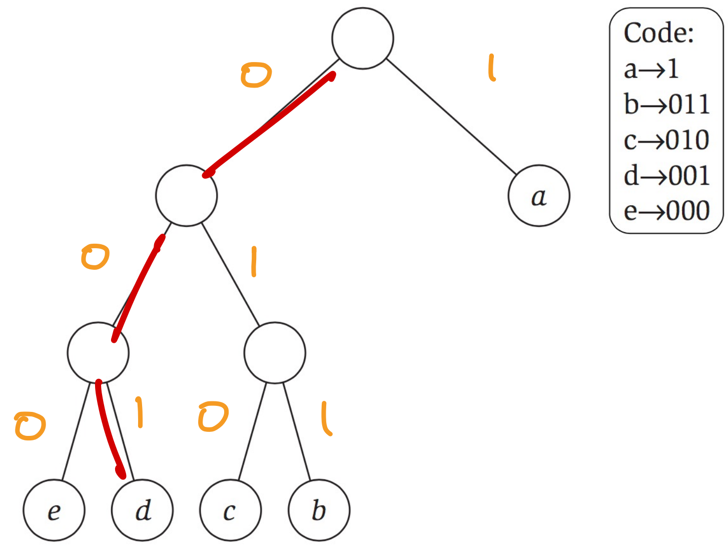
- Cannot decode if there are ambiguities
  - e.g. Encode(E) is a prefix of Encode(S)

- Prefix-Free Code:
  - A binary $\text{enc}: \Sigma \rightarrow \{0,1\}^*$ such that
    for every $x \neq y \in \Sigma$, $\text{enc}(x)$ is not a prefix of $\text{enc}(y)$
  - Any fixed-length code is prefix-free



Prefix Free Codes

- Can represent a prefix-free code as a tree

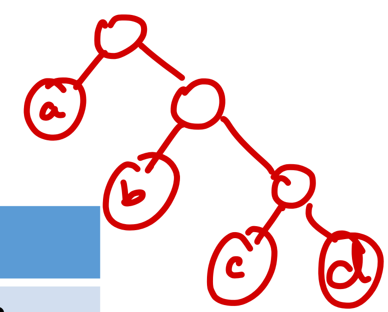


- Encode by going up the tree (or using a table)
  - d a b → 0 0 1 1 0 0 1 1
- Decode by going down the tree
  - 0 1 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1

Huffman Codes

- (An algorithm to find) an **optimal** prefix-free code

- **optimal** = $\min_{\text{prefix-free } T} \text{len}(T) = \sum_{i \in \Sigma} f_i \cdot \text{len}_T(i)$
  - Note, optimality depends on what you're compressing
  - H is the 8^{th} most frequent letter in English (6.094%) but the 20^{th} most frquent in Italian (0.636%)



	a	b	c	d
Frequency	1/2	1/4	1/8	1/8
Encoding	0	10	110	111

$\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = \frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{3}{8} = \frac{14}{8} = 1.75$

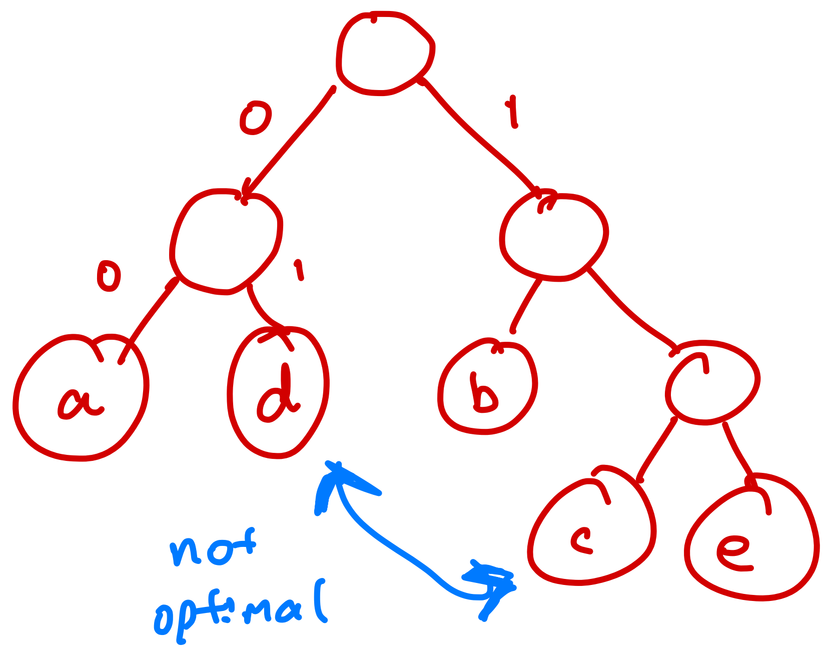
Huffman Codes

- First Try: split letters into two sets of roughly equal frequency and recurse
  - Balanced binary trees should have low depth

a	b	c	d	e
.32	.25	.20	.18	.05

$f_{\{a,d\}} = .5$

$f_{\{b,c,e\}} = .5$

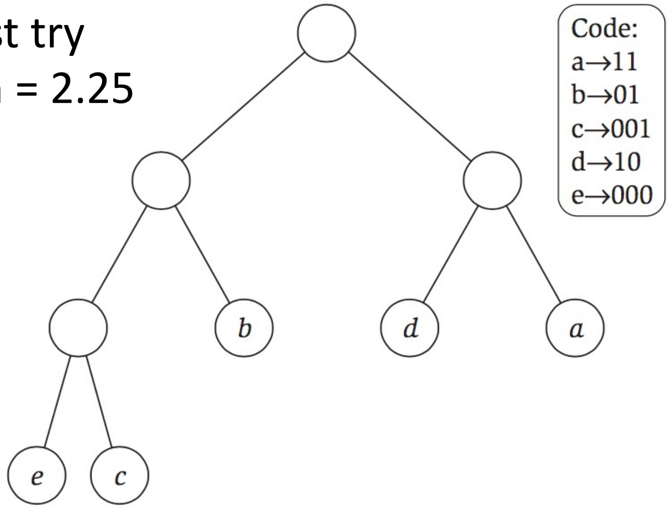


Huffman Codes

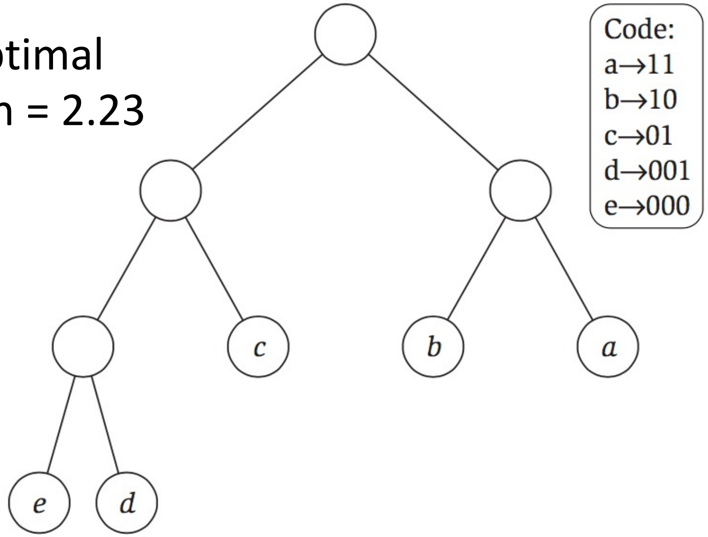
- First Try: split letters into two sets of roughly equal frequency and recurse

a	b	c	d	e
.32	.25	.20	.18	.05

first try
len = 2.25



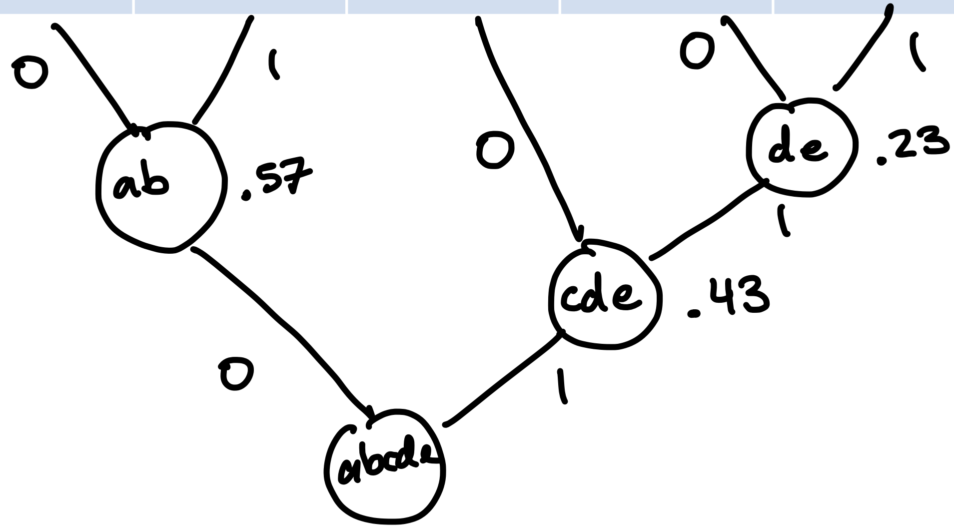
optimal
len = 2.23



Huffman Codes

- Huffman's Algorithm: pair up the two letters with the lowest frequency and recurse

a	b	c	d	e
.32	.25	.20	.18	.05



a → 00
b → 01
c → 10
d → 110
e → 111

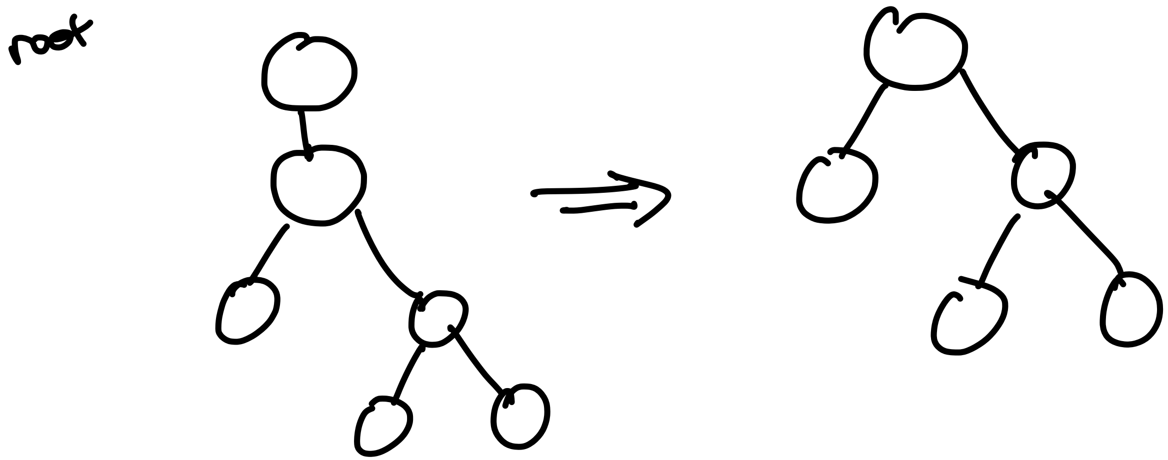
Huffman Codes

- Huffman's Algorithm: pair up the two letters with the lowest frequency and recurse

- Theorem: Huffman's Algorithm produces a prefix-free code of optimal length
  - We'll prove the theorem using an exchange argument

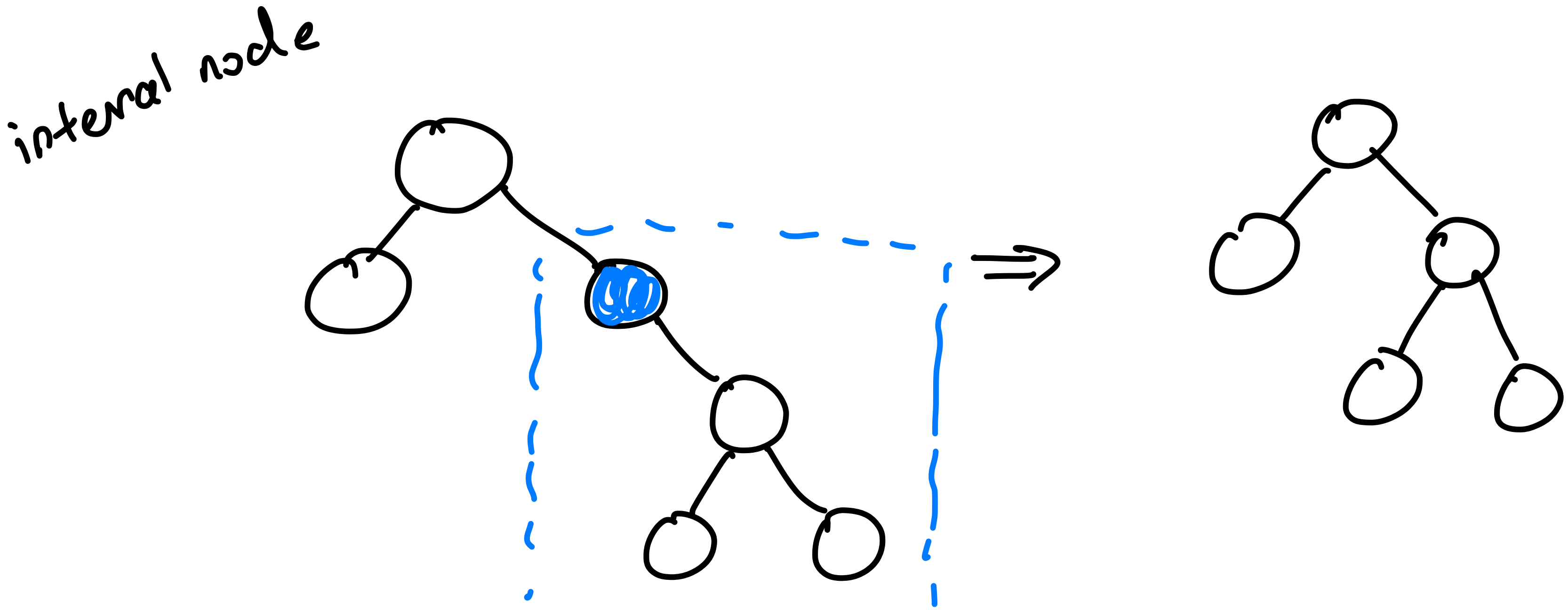
Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- (1) In an optimal prefix-free code (a tree), every internal node has exactly two children



Huffman Codes

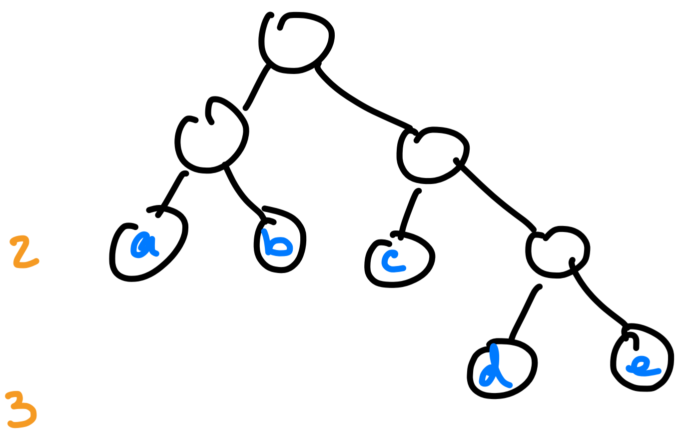
- Theorem: Huffman's Alg produces an optimal prefix-free code
- (1) In an optimal prefix-free code (a tree), every internal node has exactly two children



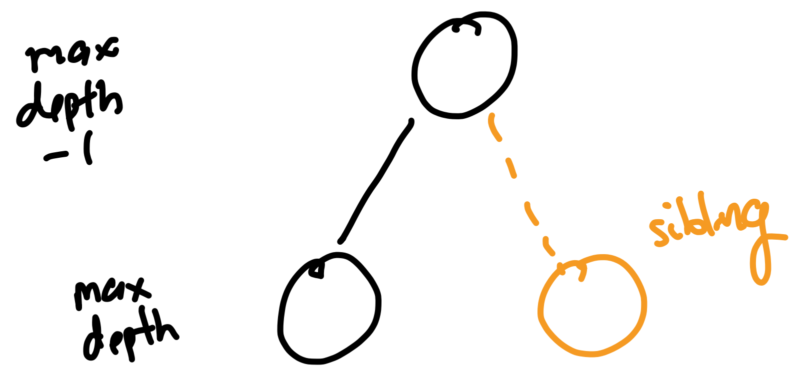
Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- (2) If x, y have the lowest frequency, then there is an optimal code where x, y are siblings and are at the bottom of the tree

If I know the shape of the tree T, then there is a greedy way to label the leaves



$f_a \geq f_b \geq \ldots \geq f_e$



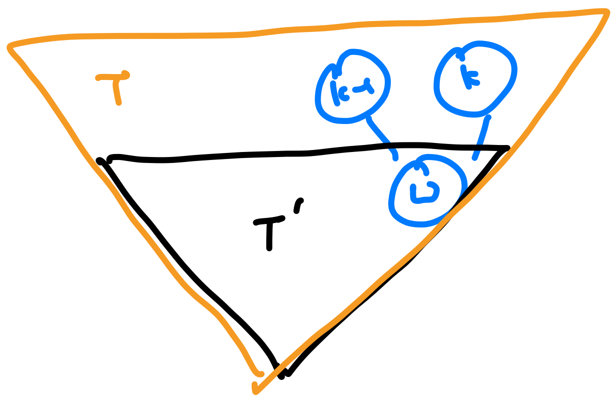
Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- Proof by Induction on the Number of Letters in Σ:
  - Base case ($|\Sigma| = 2$): rather obvious
  - Inductive Hypothesis ($|\Sigma| = k-1$):

$$f_1 \geq f_2 \geq \ldots \geq f_{k-1} \geq f_k$$

ω

$$f_\omega = f_{k-1} + f_k$$



$$len(T') = \sum_{i=1}^{k-2} f_i \cdot len_{T'}(i) + f_\omega \cdot len_{T'}(\omega)$$

T' is optimal for $\{1, 2, \ldots, k-2, \omega\}$

$$len(T) = \left(\sum_{i=1}^{k-2} f_i \cdot len_{T'}(i) + f_\omega \cdot len_{T'}(\omega)\right) + f_\omega = len(T') + f_{k-1} + f_k$$

An Experiment

- Take the Dickens novel *A Tale of Two Cities*
  - File size is 799,940 bytes
- Build a Huffman code and compress

char	frequency	code
'A'	48165	1110
'B'	8414	101000
'C'	13896	00100
'D'	28041	0011
'E'	74809	011
'F'	13559	111111
'G'	12530	111110
'H'	38961	1001

char	frequency	code
'I'	41005	1011
'J'	710	1111011010
'K'	4782	11110111
'L'	22030	10101
'M'	15298	01000
'N'	42380	1100
'O'	46499	1101
'P'	9957	101001
'Q'	667	1111011001

char	frequency	code
'R'	37187	0101
'S'	37575	1000
'T'	54024	000
'U'	16726	01001
'V'	5199	1111010
'W'	14113	00101
'X'	724	1111011011
'Y'	12177	111100
'Z'	215	1111011000

  - File size is now 439,688 bytes

	Raw	Huffman
Size	799,940	439,688

Huffman Codes

- Huffman's Algorithm: pair up the two letters with the lowest frequency and recurse
- Theorem: Huffman's Algorithm produces a prefix-free code of optimal length
- In what sense is this code really optimal?

Entropy and Compression

- Given a set of frequencies (probability distribution) the entropy is

$$H(f) = \sum_i f_i \cdot \log_2 \left({}^1/_{f_i} \right)$$

- Suppose that we generate string S by choosing n random letters independently with frequencies f
- Any compression scheme requires at least $H(f)$ bits-per-letter to store S (as $n \to \infty$)
  - Huffman codes are truly optimal!

But Wait!

- Take the Dickens novel *A Tale of Two Cities*
  - File size is 799,940 bytes
- Build a Huffman code and compress

char	frequency	code
'A'	48165	1110
'B'	8414	101000
'C'	13896	00100
'D'	28041	0011
'E'	74809	011
'F'	13559	111111
'G'	12530	111110
'H'	38961	1001

char	frequency	code
'I'	41005	1011
'J'	710	1111011010
'K'	4782	11110111
'L'	22030	10101
'M'	15298	01000
'N'	42380	1100
'O'	46499	1101
'P'	9957	101001
'Q'	667	1111011001

char	frequency	code
'R'	37187	0101
'S'	37575	1000
'T'	54024	000
'U'	16726	01001
'V'	5199	1111010
'W'	14113	00101
'X'	724	1111011011
'Y'	12177	111100
'Z'	215	1111011000

  - File size is now 439,688 bytes
- But we can do better!

	Raw	Huffman	gzip	bzip2
Size	799,940	439,688	301,295	220,156

What do the frequencies represent?

- Real data (e.g. natural language, music, images) have patterns between letters
  - U becomes a lot more common after a Q
- Possible approach: model pairs of letters
  - Build a Huffman code for pairs-of-letters
  - Improves compression ratio, but the tree gets bigger
  - Can only model certain types of patterns
- Zip is based on an algorithm called LZW that tries to identify patterns based on the data