CS 3000: Algorithms & Data
Jonathan Ullman

Lecture 19:
- Data Compression
- Greedy Algorithms: Huffman Codes

~~Apr 5, 2018~~ Apr 8, 2020

Data Compression

- How do we store strings of text compactly?

Alphabet

- 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

A: 00000
B: 00001
C: 00010
D: 00011

- Morse Code:

(Variable length 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 •–•	

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**

$$\left(\frac{1}{2}\right)\times 1 + \left(\frac{1}{4}\right)\times 2 + \left(\frac{1}{4}\right)\times 3$$

$$= \frac{1}{2} + \frac{1}{2} + \frac{3}{4} = \frac{7}{4} = 1.75$$

Data Compression

- What properties would a good code have?

  - Easy to encode a string

    Encode(KTS) = – ● – | – | ● ● ● |

    K T S

  - The encoding is short on average

    average bits per letter given some frequencies

    ≤ 4 bits per letter (30 symbols max!)

  - Easy to decode a string?

    Decode(– ● – – ● ● ●) =

    K T S

    T E T T S

    T E T T E E E

    K N I ...

Many possibilities

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

E: •
S: • • •

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

- **Prefix-Free Code:**
  - A binary enc: $\Sigma \to \{0,1\}^*$ 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

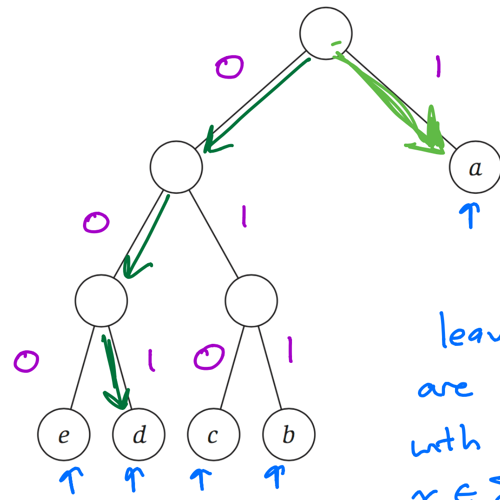
a: 00
b: 01
c: 10
d: 11

a: 0
b: 10
c: 110
d: 111
(a prefix-free variable-length code)

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

Prefix Free Codes

- Can represent a prefix-free code as a binary tree



Code:
a→1
b→011
c→010
d→001
e→000

leaves are labeled with a symbol $x \in \Sigma$

- Encode by going up the tree (or using a table)
  - d a b → ~~0 0 1 1 0 0 1 1~~ 001 | 1 | 011
- Decode by going down the tree
  - 0 1 1 | 0 0 0 | 1 | 0 0 1 | 0 1 0 | 1 | 0 1 1 |

b e a d c a b

Huffman Codes

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

average number of bits per letter

- **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 8th most frequent letter in English (6.094%) but the 20th most frquent in Italian (0.636%)

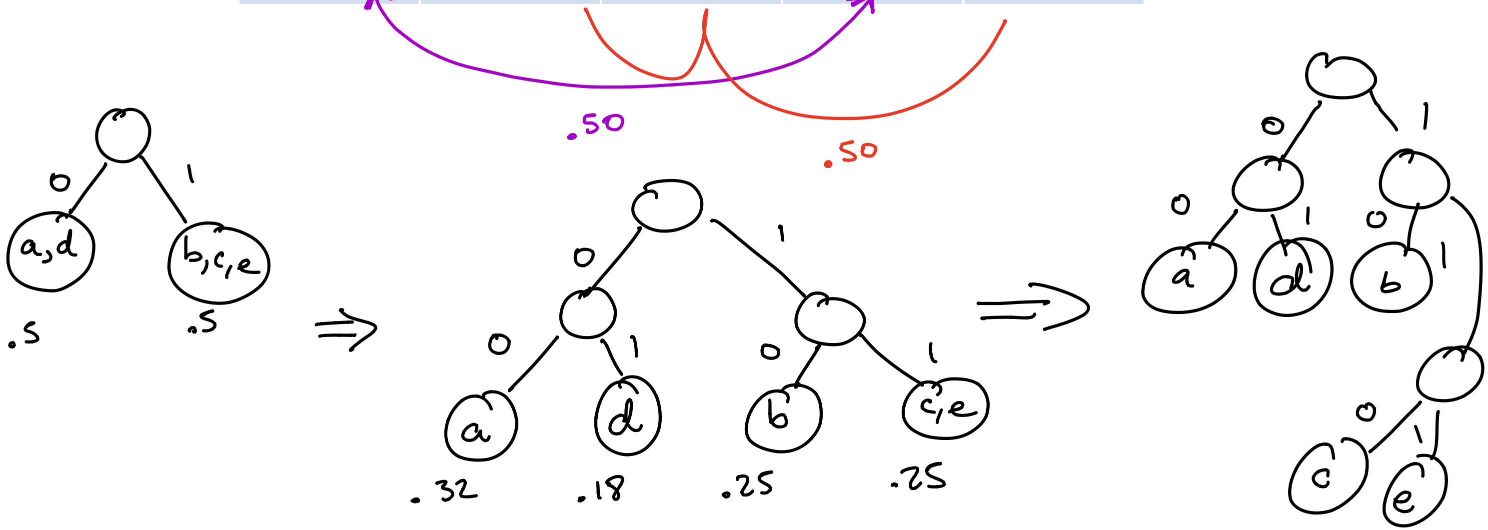
	a	b	c	d
	f_a	f_b	f_c	f_d
Frequency	1/2	1/4	1/8	1/8
Encoding	0	10	110	111

$f_a \times 1 + f_b \times 2 + f_c \times 3 + f_d \times 3 = 1.75$

Huffman Codes

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

	00	10	110	01	111
	a	**b**	**c**	**d**	**e**
	.32	.25	.20	.18	.05

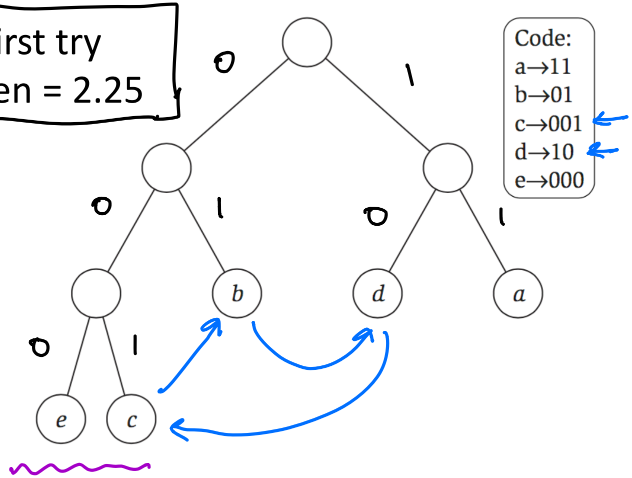


Huffman Codes

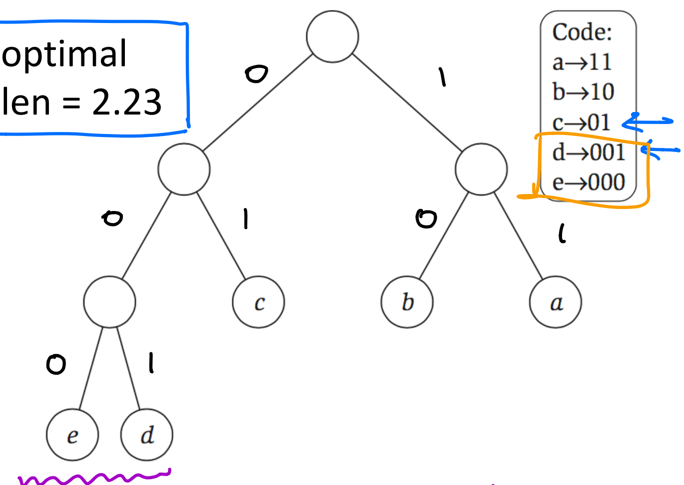
- First Try: split letters into two sets of roughly equal frequency and recurse

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

first try
len = 2.25



optimal
len = 2.23



two longest codewords are the least frequent letter

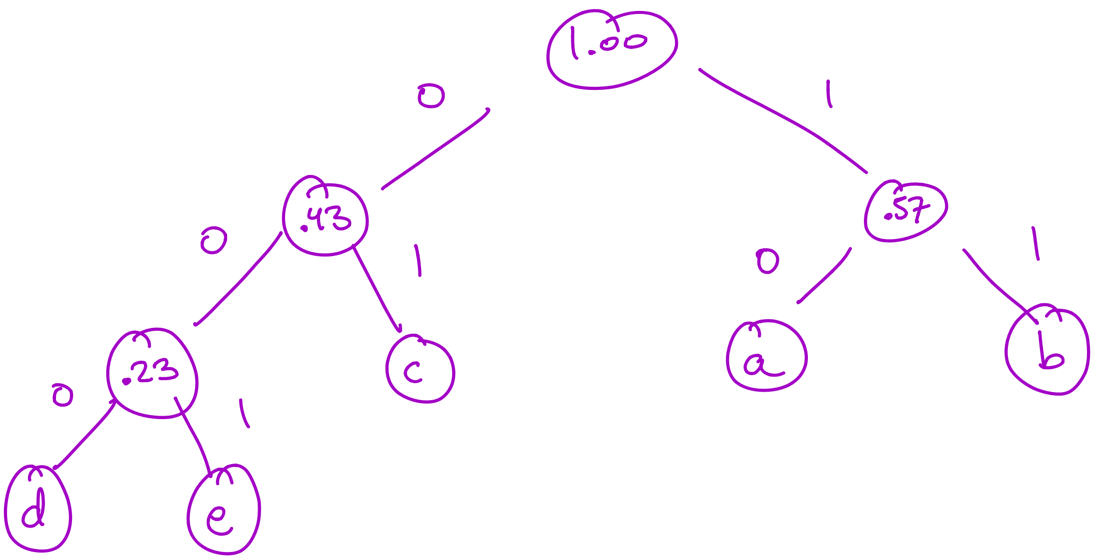
Huffman Codes

a b {c, d, e}
.32 .25 .43

.57 {a, b} .43 {c, d, e}

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

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

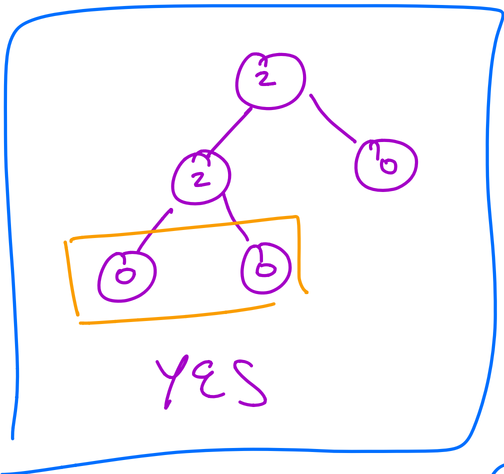


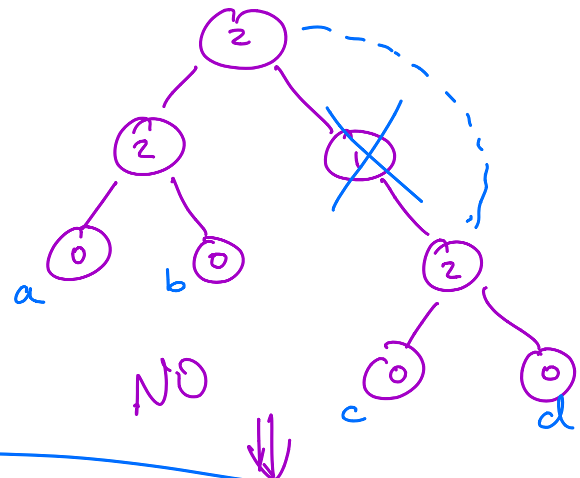
Huffman Codes

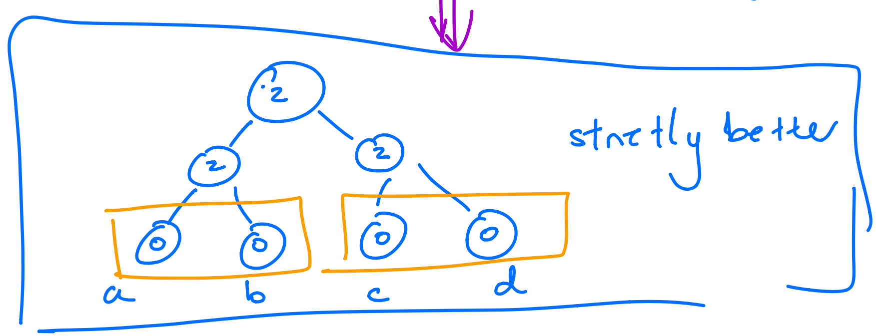
- Huffman's Algorithm: pair up the two letters with the lowest frequency and recurse
- Theorem: Huffman's Algorithm produces a prefix-free code of optimal length
  - We'll prove the theorem using an exchange argument

Huffman Codes

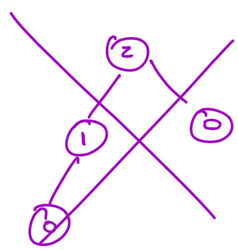
- Theorem: Huffman's Alg produces an optimal prefix-free code
- (1) In an optimal prefix-free code (a tree), every internal node has exactly two children







$\Longrightarrow$ In the optimal code. If the lowest depth is d, then there are at least two leaves at depth d, and they are siblings



CANT HAPPEN

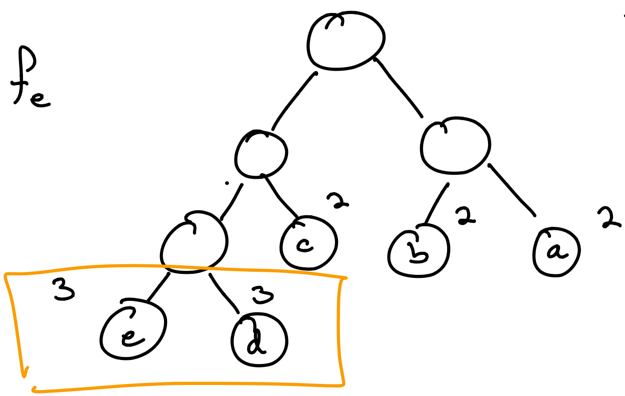
Huffman Codes

- 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

(i.e. have the lowest depth)

Suppose someone gave you the optimal tree, but without labels...

$f_a > f_b > f_c > f_d > f_e$



... then I should label the highest leaves with the most frequent symbols and go down

By (1) there are two siblings at the lowest depth.
My optimal code fills those siblings w/ the least frequent items

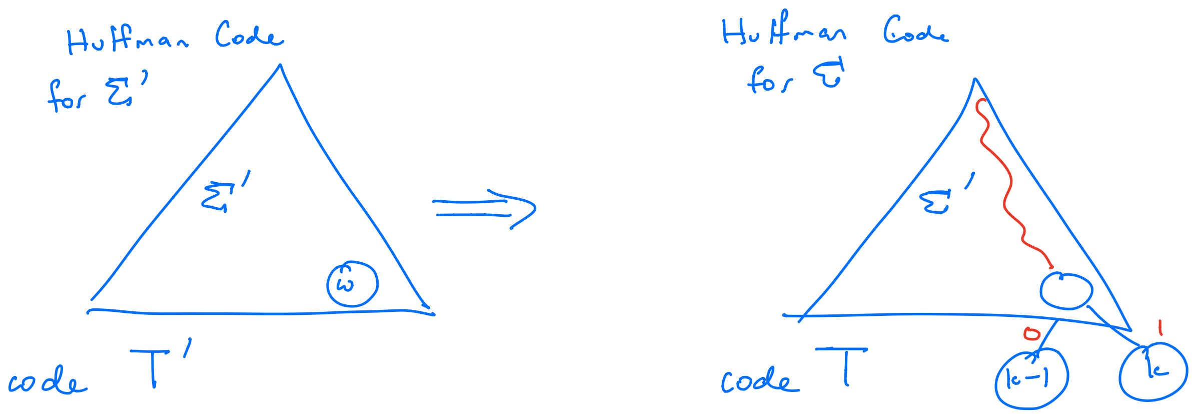
Huffman Codes

- 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 Step: If Huffmans alg is optimal for $|\Sigma| = k-1$ then its optimal for $|\Sigma| = k$

Suppose we have frequencies $f_1 \geq f_2 \geq \ldots \geq \boxed{f_{k-1} \geq f_k}$

$\Sigma' = \{1, 2, 3, \ldots, k-2, \omega\}$ $f_\omega = f_{k-1} + f_k$

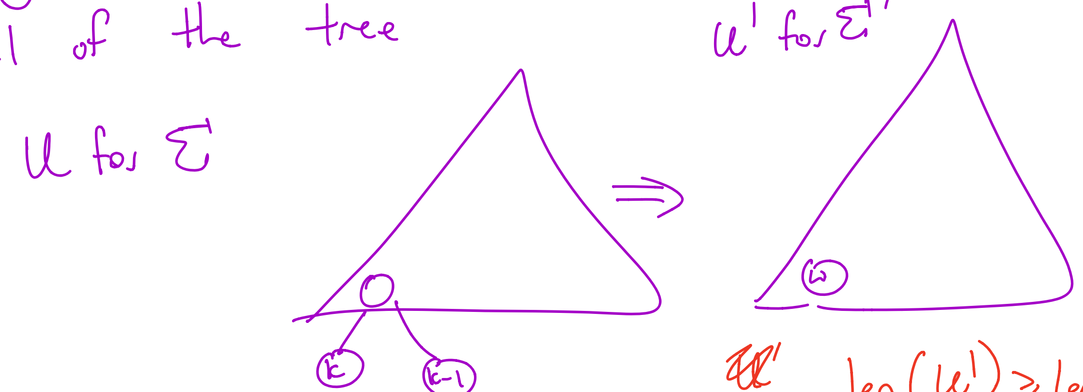
$|\Sigma'| = k-1$



$$\begin{aligned} len(T) &= len(T') + f_\omega \\ &= len(T') + f_{k-1} + f_k \end{aligned}$$

By the inductive hypothesis, T' is an optimal code for Σ' (minimizes $len(T')$)

- Suppose $\mathcal{U}$ is an optimal code for Σ
- By (2), $k-1$ and k are siblings at the lowest level of the tree



$$len(\mathcal{U}') = len(\mathcal{U}) - f_k - f_{k-1}$$

$$len(\mathcal{U}') \geq len(T')$$
$$len(\mathcal{U}) \geq len(T)$$

Huffman Codes

- Theorem: Huffman’s Alg produces an optimal prefix-free code
- Proof by Induction on the Number of Letters in Σ:
    - Inductive Hypothesis:

Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- Proof by Induction on the Number of Letters in Σ:
  - Inductive Hypothesis:
  - Without loss of generality, frequencies are $f_1, \ldots, f_k$, the two lowest are f_1, f_2
  - Merge 1,2 into a new letter $k+1$ with $f_{k+1} = f_1 + f_2$

Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- Proof by Induction on the Number of Letters in Σ:
  - Inductive Hypothesis:

  - Without loss of generality, frequencies are $f_1, \ldots, f_k$, the two lowest are f_1, f_2
  - Merge 1,2 into a new letter $k+1$ with $f_{k+1} = f_1 + f_2$

  - By induction, if T' is the Huffman code for $f_3, \ldots, f_{k+1}$, then T' is optimal
  - Need to prove that T is optimal for $f_1, \ldots, f_k$

Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- If T' is optimal for $f_3, \ldots, f_{k+1}$ then T is optimal for $f_1, \ldots, f_k$

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	3
'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	3
'U'	16726	01001	
'V'	5199	1111010	
'W'	14113	00101	
'X'	724	1111011011	10
'Y'	12177	111100	
'Z'	215	1111011000	10

  - File size is now 439,688 bytes

	Raw	Huffman
Size	799,940	439,688

≈ 55%

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? (Bonus material… will not test you on this)

Length of Huffman Codes

for integer ℓ_i

- What can we say about Huffman code length?
  - Suppose $f_i = 2^{-\ell_i}$ for every $i \in \Sigma$
  - Then, $\mathrm{len}_T(i) = \ell_i$ for the optimal Huffman code

- ~~Proof:~~

letter	a	b	c	d
freq	2^{-1}	2^{-2}	2^{-3}	2^{-3}
code	0	10	110	111
len	1	2	3	3

Length of Huffman Codes

- What can we say about Huffman code length?
  - Suppose $f_i = 2^{-\ell_i}$ for every $i \in \Sigma$
  - Then, $\text{len}_T(i) = \ell_i$ for the optimal Huffman code

- $\boxed{\text{len}(T) = \sum_{i\in\Sigma} f_i \cdot \log_2(1/f_i)}$

$$= \sum_{i\in\Sigma} 2^{-\ell_i} \cdot \ell_i$$

$f_i = 2^{-\ell_i}$

$\log_2(f_i) = -\ell_i$

$\log_2(1/f_i) = \ell_i$

Entropy

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

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

= lengths of the Huffman code

- Entropy is a "measure of randomness"

Entropy

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

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

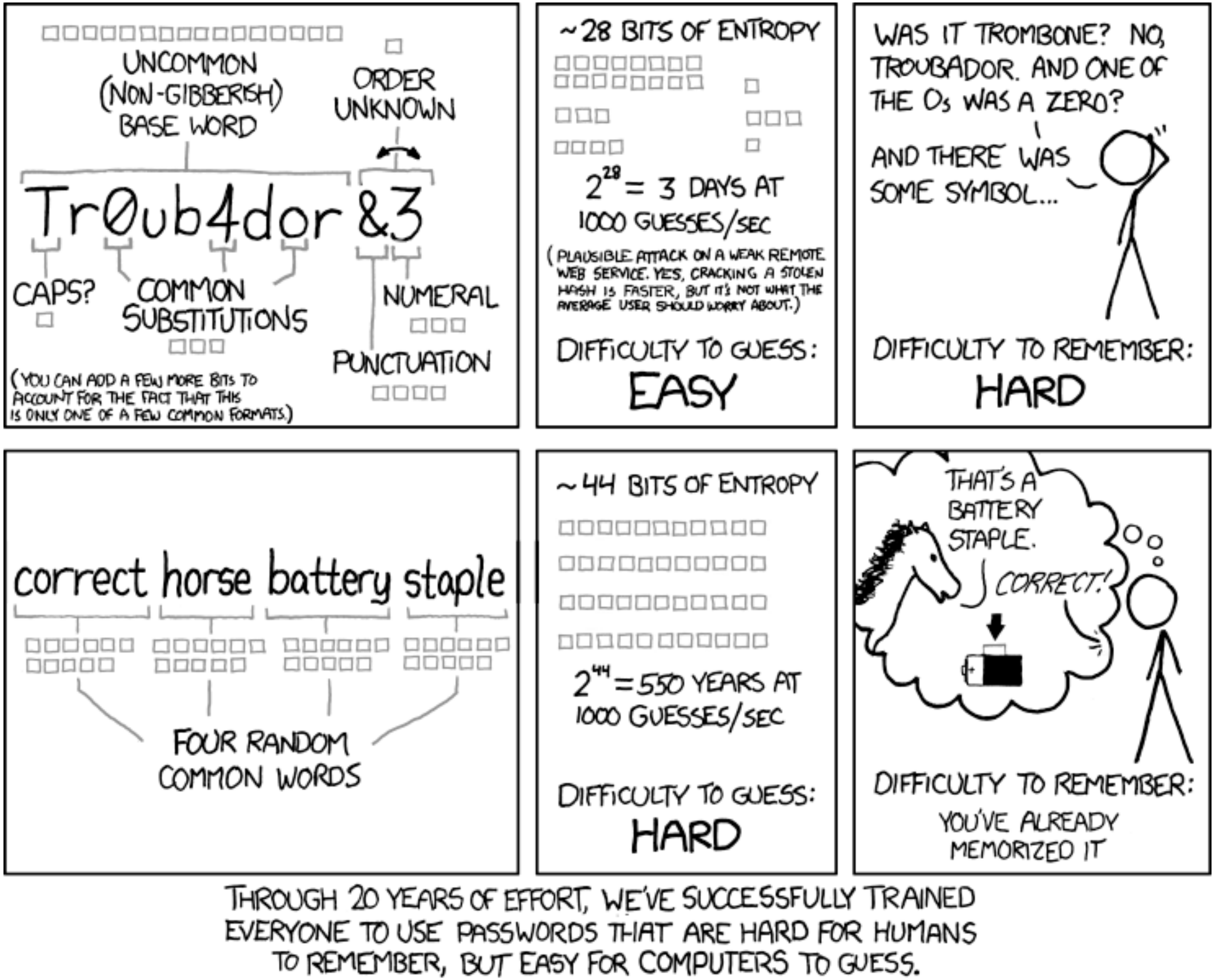
How "random" f is the text

- Entropy is a "measure of randomness"
- Entropy was introduced by Shannon in 1948 and is the foundational concept in:
  - Data compression
  - Error correction (communicating over noisy channels)
  - Security (passwords and cryptography)

Entropy of Passwords

- Your password is a specific string, so $f_{pwd} = 1.0$
- To talk about security of passwords, we have to model them as **random**
  - Random 16 letter string: $H = 16 \cdot \log_2 26 \approx 75.2$
  - Random IMDb movie: $H = \log_2 1764727 \approx 20.7$
  - Your favorite IMDb movie: $H \ll 20.7$
- Entropy measures how difficult passwords are to guess "on average"

Entropy of Passwords



Entropy and Compression

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

$$H(f) = \sum_i f_i \cdot \log_2 \left({}^{1}\!/\!_{f_i} \right) = \text{length of Huffman code}$$

- 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

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
  - Huffman codes are truly optimal if and only if there is no relationship between different letters!