CS4800: Algorithms & Data
Jonathan Ullman

Lecture 22:
- Greedy Algorithms: Huffman Codes
- Data Compression and Entropy

Apr 10, 2018

Data Compression

- How do we store strings of text compactly?

alphabet

- A (binary) code is a mapping from $\boxed{\Sigma} \rightarrow \boxed{\{0,1\}^*}$

binary strings

e.g. ASCII

fixed length code

  - 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 ●▬●	

variable length code

Data Compression

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

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

Fixed length code: 2 bits per symbol

Variable length code: $\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{4} \times 3$

$= \frac{1}{2} + \frac{1}{2} + \frac{3}{4} = 1.75$ bits per symbol

Data Compression

To encode a string w/ n letters, do n array lookups

- 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!)

    Enc(EEE)
    = Enc(S)

  - Easy to decode a string

    Decode(– • – – • • •) =

    TETTS
    TETTEEE
    KDE

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. enc(E) is a prefix of enc(S)

- Prefix-Free Code:
  - A binary $\text{enc}: \Sigma \to \{0,1\}^*$ s.t. for every $x \neq y \in \Sigma$, $\text{enc}(x)$ is not a prefix of $\text{enc}(y)$

Morse Code is NOT prefix-free

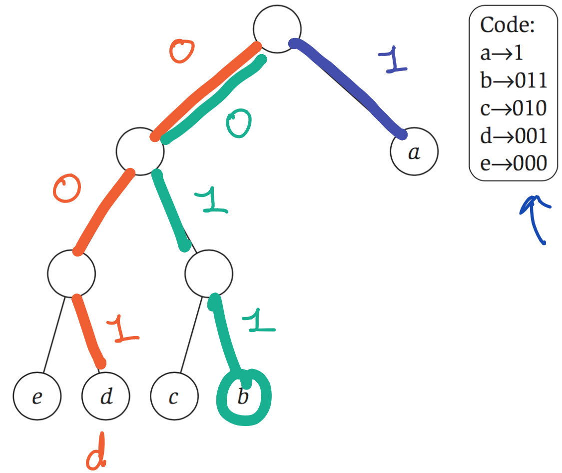
  - Any fixed-length code is prefix-free
  - Can make any code prefix-free by adding some string meaning STOP

This will increase the number of bits to store.

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

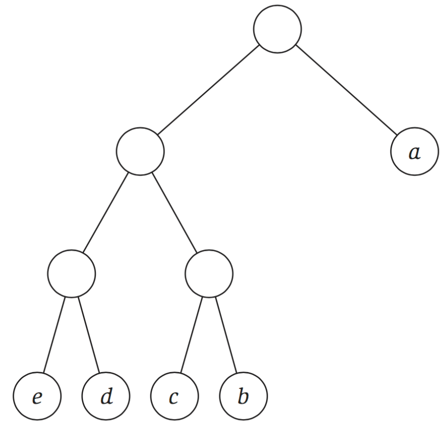
- Can represent a prefix-free code as a tree



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

Prefix Free Codes

- Can represent a prefix-free code as a tree



Code:
a→1
b→011
c→010
d→001
e→000

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

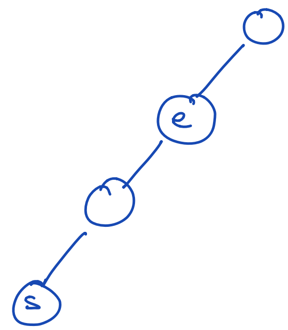
d→ 001 a→ 1 b→ 011

- 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

e = ○

s = ○ ○ ○



Huffman Codes

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

average bits per symbol

- optimal = minimizes $\mathrm{len}(T) = \sum_{i \in \Sigma} f_i \cdot \mathrm{len}_T(i)$
  - 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%)

$\mathrm{len}(T) = 1.75$

	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
	$\mathrm{len}_T(a) = 1$	$\mathrm{len}_T(b) = 2$	$\mathrm{len}_T(c) = 3$	$\mathrm{len}_T(d) = 3$

Huffman Codes

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

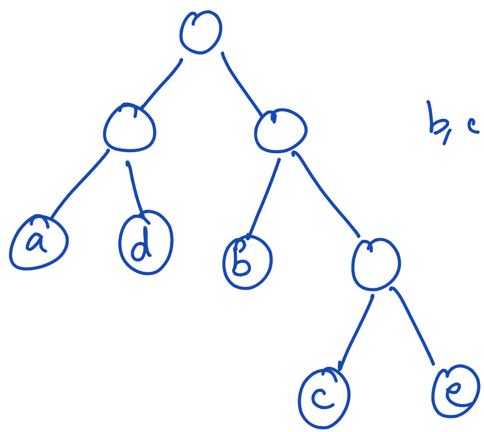
a	b	c	d	e
.32	.25	.20	.18	.05

$f_a + f_d = .5$

$f_c + f_e = .25$

a, d

b, c, e



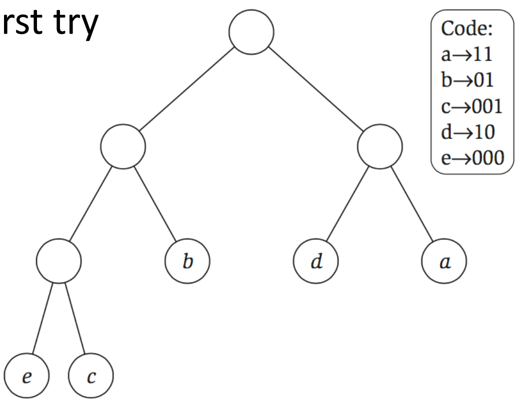
Huffman Codes

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

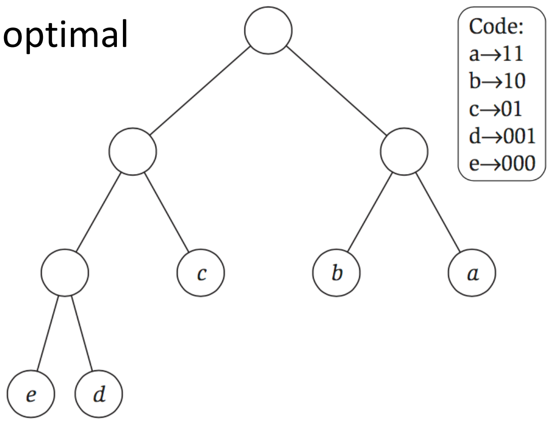
a	b	c	d	e
.32	.25	.20	.18	.05

len(opt) < len(first try)

first try



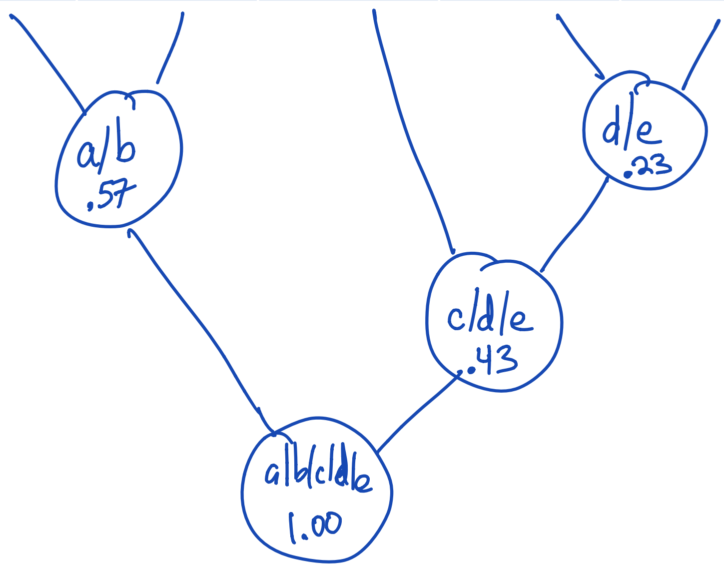
optimal



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

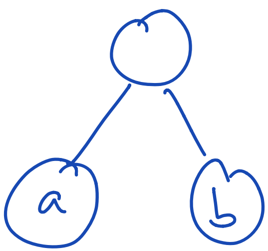
Huffman Codes

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

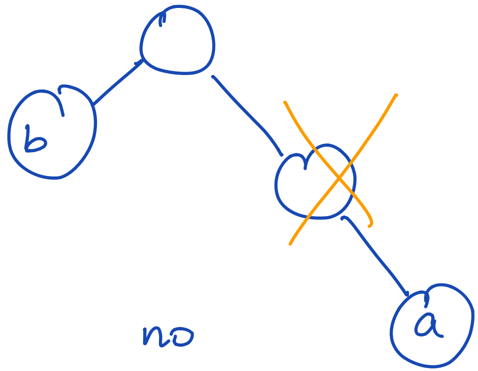
- Theorem: Huffman's Algorithm produces a prefix-free code of optimal length
  - We'll prove the theorem using an exchange argument

Huffman Codes

- Theorem: Huffman's Alg produces an optimal prefix-free code
- (1) In an optimal prefix-free code (an optimal tree), every internal node has exactly 2 children



yes



no

If we have a code like this,
we could reduce its length

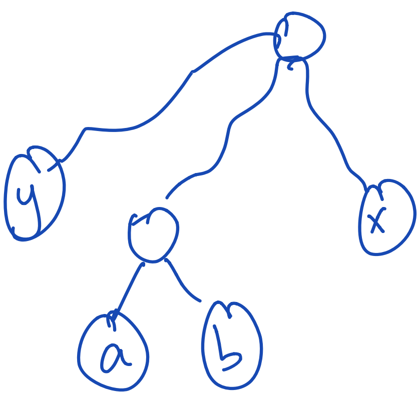
Huffman Codes

- 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

Suppose T is an optimal code w/ a, b as siblings at the lowest level

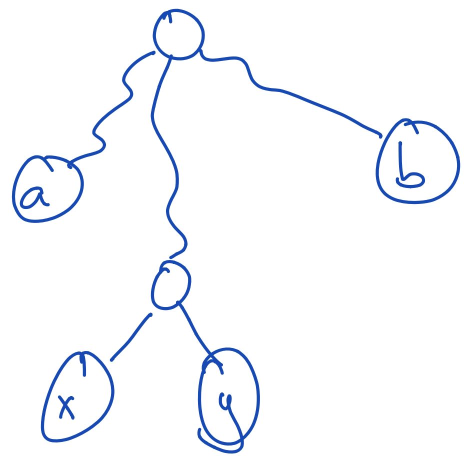
T' is better than T

T



$\Longrightarrow$

T'

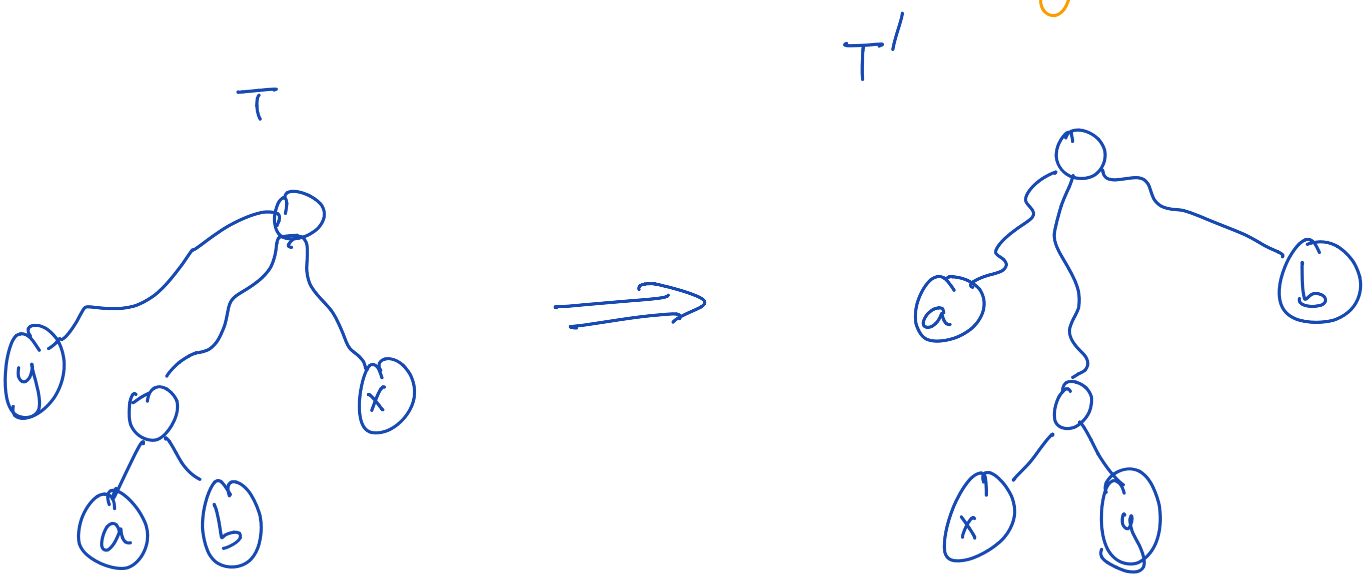


Huffman Codes

- 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

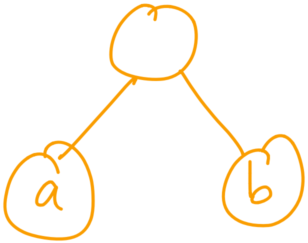
swapping a and x improves the length because $f_a > f_x$

swapping b and y improves the length because $f_b > f_y$



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



Huffman Codes

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

For any Σ' of size $k-1$ and any $f_1, \ldots, f_{k-1}$, Huffman's algorithm produces the optimal prefix-free code.

Huffman Codes

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

    For any Σ' of size $k-1$ and any $f_1, \ldots, f_{k-1}$,
    Huffman's algorithm produces the optimal prefix-free code.
  - 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 $f_{k+1} = f_1 + f_2$

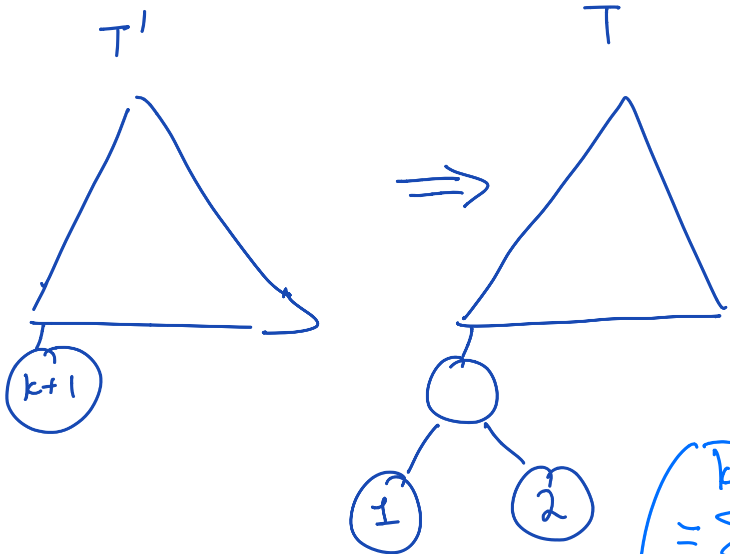
New alphabet $\Sigma' = \{3, 4, \ldots, k, k+1\}$ $\quad f_3, f_4, \ldots, f_k, f_{k+1}$

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 $f_1, \ldots, f_k$



$$\mathrm{len}(T) = \sum_{i=1}^{k} f_i \cdot \mathrm{len}_T(i)$$

$$= \sum_{i=3}^{k} f_i \cdot \mathrm{len}_T(i) + f_1 \,\mathrm{len}_T(1) + f_2 \cdot \mathrm{len}_T(2)$$

$$= \sum_{i=3}^{k} f_i \,\mathrm{len}_T(i) + f_1 \left(\mathrm{len}_{T'}(k+1) + 1\right) + f_2 \left(\mathrm{len}_{T'}(k+1) + 1\right)$$

$$= \sum_{i=3}^{k} f_i \cdot \mathrm{len}_{T'}(i) + f_{k+1} \cdot \mathrm{len}_{T'}(k+1) + f_1 + f_2$$

$$= \mathrm{len}(T') + f_1 + f_2$$

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?

Length of Huffman Codes

- 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:

Length of Huffman Codes

- 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

  - $\mathrm{len}(T) = \sum_{i \in \Sigma} f_i \cdot \log_2({}^1/_{f_i})$

Entropy

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

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

- Entropy is a "measure of randomness"

choose a random k bit string

$\forall i \quad f_i = 2^{-k}$

$H(f) = 2^k \cdot 2^{-k} \cdot k = k$

Entropy

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

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

- Entropy is a "measure of randomness"
- Entropy was introduced by Shannon in 1948 and is the foundational concept in:
  - Data compression
  - Communicating over a noisy connection
  - Cryptography and security

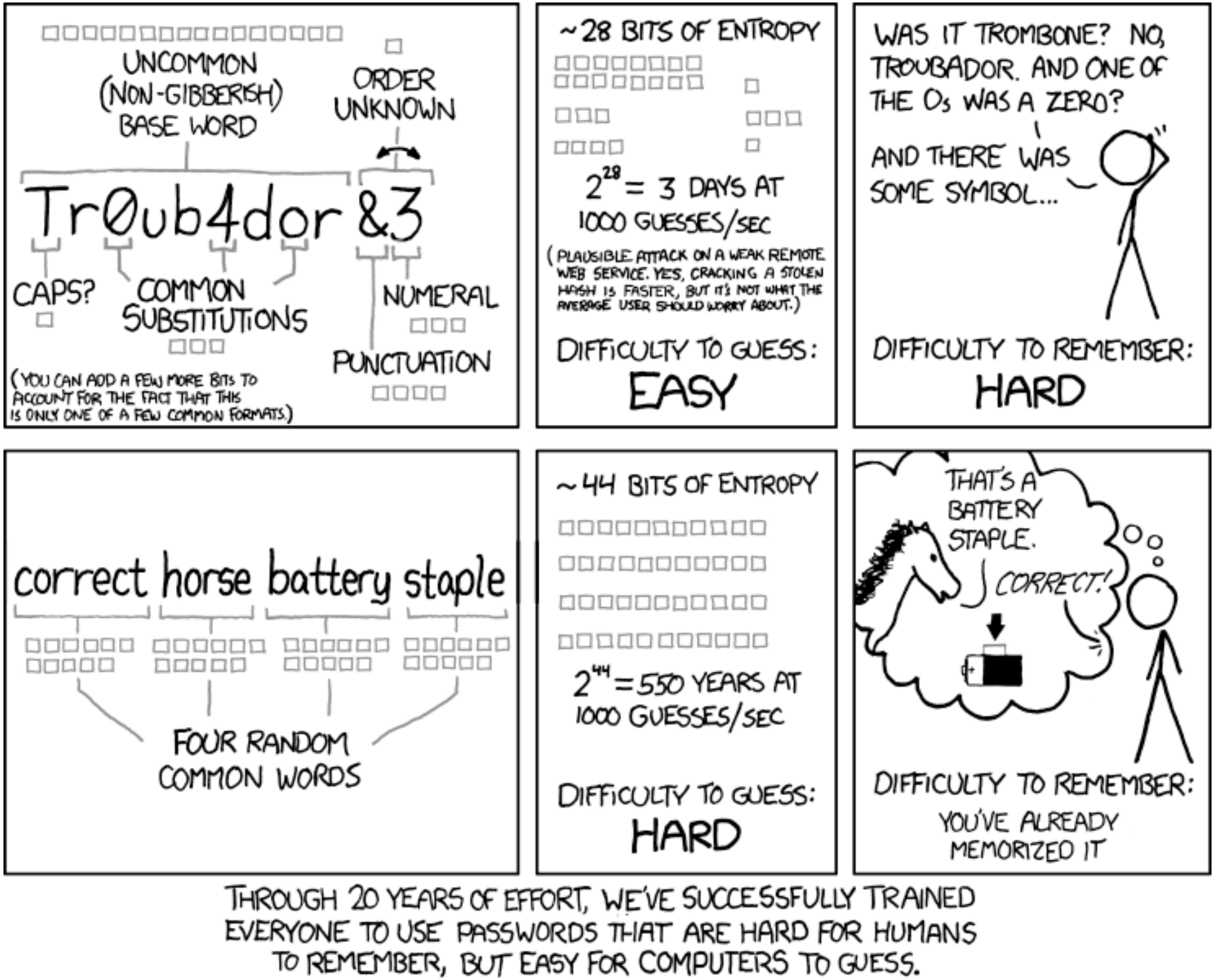
Entropy of Passwords

- 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

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

- Suppose that we generate string S by choosing n random letters independently with frequencies f
- Any compression scheme requires at least $n \cdot H(f)$ bits 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
  - The frequency of U is only 2.76% in English, but what if the previous letter is Q?
- Possible approach: model pairs of letters
  - Record the frequency of pairs of letters and build a huffman code for the pairs
  - Pros:
    - can improve compression ratio
  - Cons:
    - now the code tree is now much bigger
    - cannot identify patterns across more than pairs

Lempel-Ziv-Welch Compression

- Try to learn patterns as you find them!
- Compress: $ABBABABBACABBA$

Lempel-Ziv-Welch Compression

- Try to learn patterns as you find them!

1. Start with an initial empty “dictionary” D
2. Until input is empty:
    1. Find the longest prefix pre that matches D
    2. Output D(p) and remove p from the input
    3. Add (p + nextletter) to D

- *zip* uses (some version of) LZW compression

Entropy and Compression

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

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

- Suppose that we generate string S by choosing n random letters independently with frequencies f
- Any compression scheme requires at least $n \cdot H(f)$ bits to store S (as $n \to \infty$)
  - Huffman codes are truly optimal if (and only if) there is no relationship between different letters!