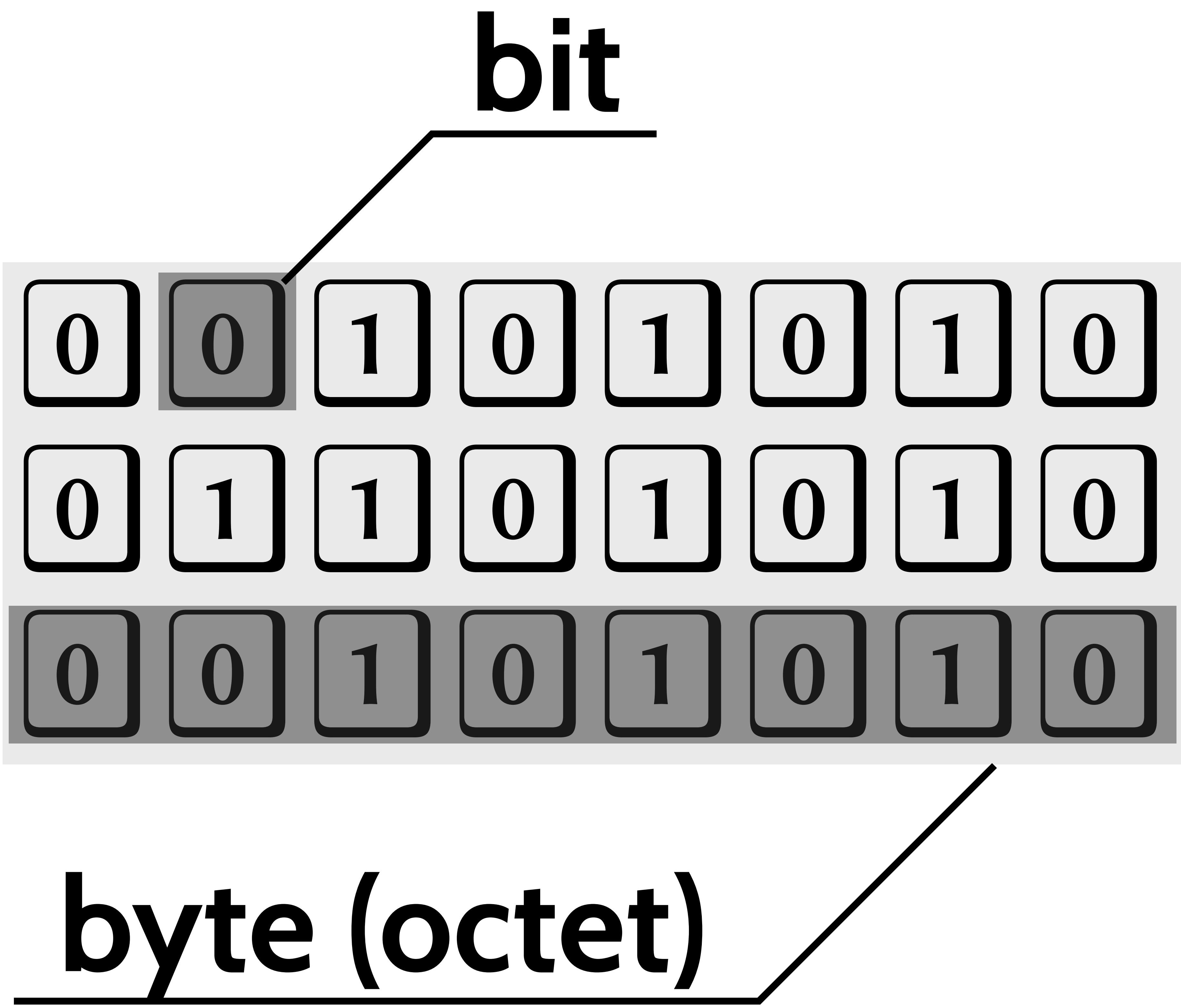
Coding

Digital Audio Coding

SEBASTIEN.BOISGERAULT@MINES-PARISTECH.FR

Binary Data



bit: - Binary digIT (J. Tukey, 1948)
- information unit (C. Shannon)

Binary Numbers

	DECIMAL	BINARY	HEXADECIMAL
BASE	10	2	16
DIGITS	$0, 1, \cdots, 9$	$0, 1$	$0, 1, \cdots, 9,$ $\mathrm{A}, \mathrm{B}, \cdots, \mathrm{F}$

TEN: $10 \longleftrightarrow 1010 \longleftrightarrow \mathrm{A}$

DEC. BIN. HEX.

$$10 = 1 \times 10^1 + 0 \times 10^0$$
$$10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$10 = 10 \times 16^0$$

Integer Literals

```
>>> 42
42
>>> 0b101010
42
>>> 0x2a
42
```

```
>>> print 42
42
>>> print bin(42)
0b101010
>>> print hex(42)
0x2a
```

Integer Arithmetic

```
>>> 2 + 2
4
>>> 2 - 1
1
>>> 3 * 2
6
```

```
>>> 5 // 2
2
>>> 5 % 2
1
>>> 5 ** 2
25
```

Binary Arithmetic I

```
>>> 42 << 3
336
>>> 42 >> 3
5
```

```
>>> 42 | 7
47
>>> 42 & 7
2
>>> 42 ^ 7
45
```

Binary Arithmetic II

```
>>> print bin(0b101010 << 3)
0b101010000
>>> print bin(0b101010 >> 3)
0b101
```

SHIFTS

<< : **left shift**

>> : **right shift**

LOGICAL

| : **or**

& : **and**

^ : **xor**

```
>>> print bin(0b101010 | 0b000111)
0b101111
>>> print bin(0b101010 & 0b000111)
0b10
>>> print bin(0b101010 ^ 0b000111)
0b101101
```

Integer Types

Python (2.x): UNBOUNDED INTEGERS

int		long
>>> 2 ** 12	>>> 2**24	>>> 2 ** 36
4096	16777216	68719476736L

NumPy: FIXED-SIZE + SIGNED/UNSIGNED

```
>>> int8(-127)
-127
>>> uint8(255)
255
```

```
>>> int8(255)
-1
>>> uint8(-127)
129
```

```
>>> int16(255)
255
>>> int16(-127)
-127
```

Unsigned 8-bit Integers

Range: 0–255, NumPy type: `uint8`.

n	bin(n)	BIT LAYOUT
0	0b0	0 0 0 0 0 0 0 0
42	0b101010	0 0 1 0 1 0 1 0
255	0b11111111	1 1 1 1 1 1 1 1
298	0b100101010	0 0 1 0 1 0 1 0

uint8 array to bitstream

```
def write_uint8(stream, integers):
    integers = array(integers)
    for integer in integers:
        mask = 0b10000000
        while mask != 0:
            stream.write((integer & mask) != 0)
            mask = mask >> 1
```

Byte Order

Given that `298` is `0b100101010`,

how do we describe `uint16(298)` ?

$\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}\quad\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}$

big endian

most significant bits first

$\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}\quad\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}$

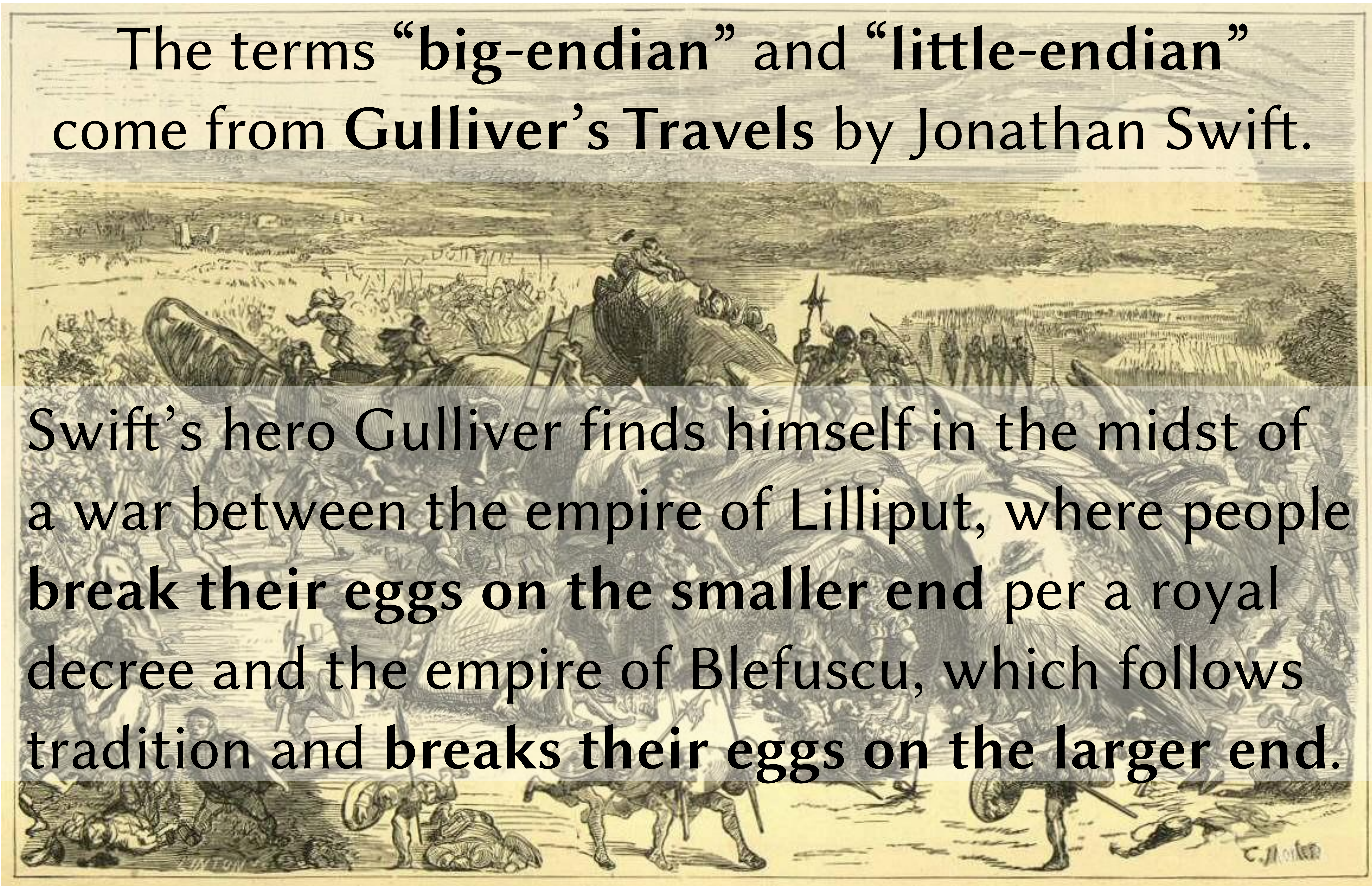
little endian

least significant bits first

Endianness

The terms “**big-endian**” and “**little-endian**” come from **Gulliver’s Travels** by Jonathan Swift.

Swift’s hero Gulliver finds himself in the midst of a war between the empire of Lilliput, where people **break their eggs on the smaller end** per a royal decree and the empire of Blefuscu, which follows tradition and **breaks their eggs on the larger end**.



“I lay all this while, as the reader may believe, in great uneasiness.”—*Page* 8.

Endianness

The **bitstream** default is big-endian:

```
>>> BitStream(298, uint16)
0000000100101010
```

but little-endian is possible too:

```
>>> uint16(298).newbyteorder()
10753
>>> BitStream(10753, uint16)
0010101000000001
```

Signed Integers

Bit layout of signed 8-bit integers

First design:

- first bit for the sign of `n`,
- the 7 following bits for `abs(n)`.

`int8(-42)`

$\boxed{1}\quad \boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}$

Issue: 0 has two distinct bit layouts.

The range would be -127 to + 127.

Signed Integers

Second design: when `n < 0`, use the 7 remaining bits to code `abs(n) - 1`.

`int8(-42):` $\boxed{1}\quad\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}$

The range is -128 to + 127.

Third design -- Two's complement:

If `n < 0`, also invert all bits (except sign)

`int8(-42):` $\boxed{1}\quad\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}$

Signed Integers

With two's complement model, arithmetic operations are easy:

Example: **-42 + 7 = -35**

$$\begin{array}{rc}
 & \boxed{1}\quad \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{0} \\
 & \boxed{0}\quad \boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{1} \\
+ & \\
\hline
 & \boxed{1}\quad \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{1}
\end{array}$$

Text Strings & ASCII

Binary data may be represented by **strings**, the class historically used to store text.

```
>>> ord('A')
65
```

```
>>> chr(65)
'A'
```

Indeed, **ASCII characters** have 255 possible ordinal values.

Text Strings & ASCII

Printable characters are in the range 20-7F. Outside this range, escape sequences **\x??** may be used

```
>>> name = 'S\xc3\xa9bastien'
```

where **??** denote the ordinal value of the character in hexadecimal.

Information Theory

relies on a probabilistic modeling

of information **sources** or **channels**.

universe **probability**

$$(\Omega, \mathfrak{E}, P)$$

events

it formalizes the relationships between:

- **event likelyhood** and **information content**,
- the mean information content of a source,

named **entropy** and the number of bits required for the source coding.

Info. Content Axioms

Positive

$$I : \mathfrak{E} \to [0, +\infty]$$

Additive

$$I(E_1 \wedge E_2) = I(E_1) + I(E_2) \text{ if } E_1, E_2 \text{ independent}$$

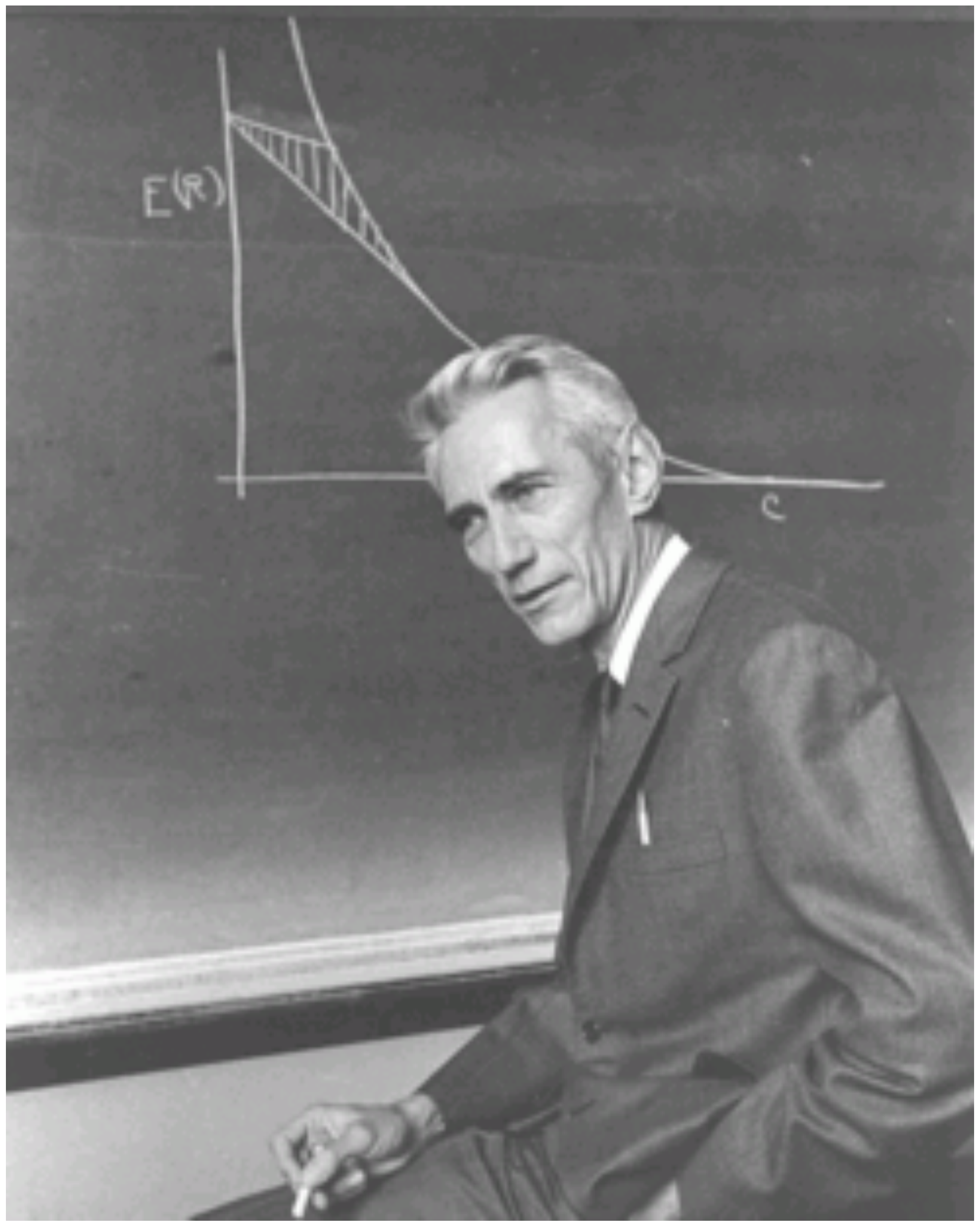
Neutral

$$I(E) = F \circ P(E)$$

Normalized

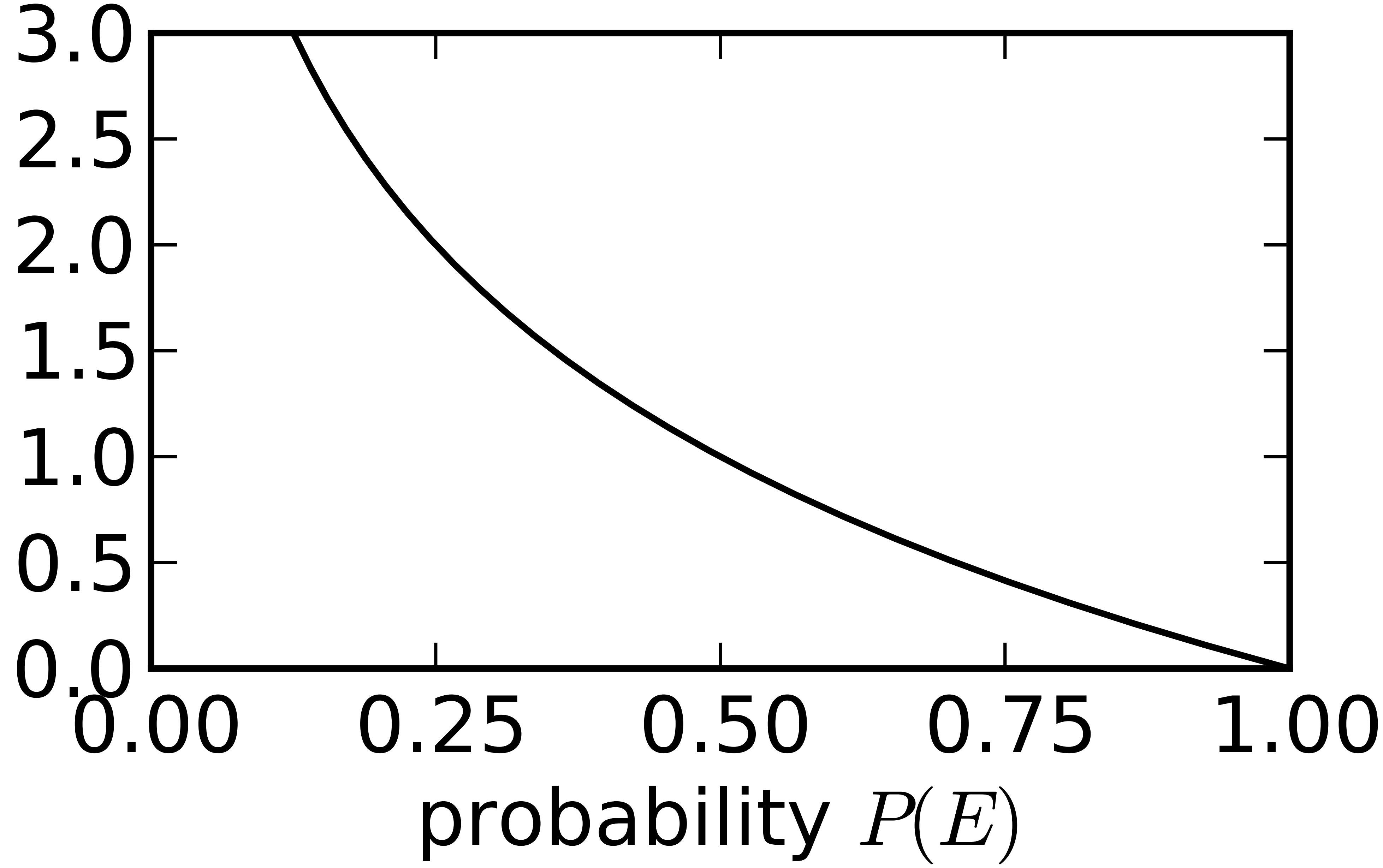
$$I(E) = 1 \text{ if } P(E) = 0.5$$

Information Content



C. Shannon

$$I(E) = -\log_2 P(E)$$



Entropy

Let X be a source (discrete random variable).
The **entropy** of X is the mean info. content of X.

$$H(X) = \mathbb{E}(I(X))$$

Explicitely, with $p(x) = P(X = x)$:

$$H(X) = -\sum_{x} p(x) \log_2 p(x)$$

Entropy Maximum

Among sources with N values, the entropy is maximal if $p(x_0) = p(x_1) = \cdots = p(x_{N-1})$. Then, we have:

$$H(X) = \log_2 N$$

The entropy of a 2^n-state system whose states are equally likely is n.

Entropy is measured in **bits**.

Application of Entropy

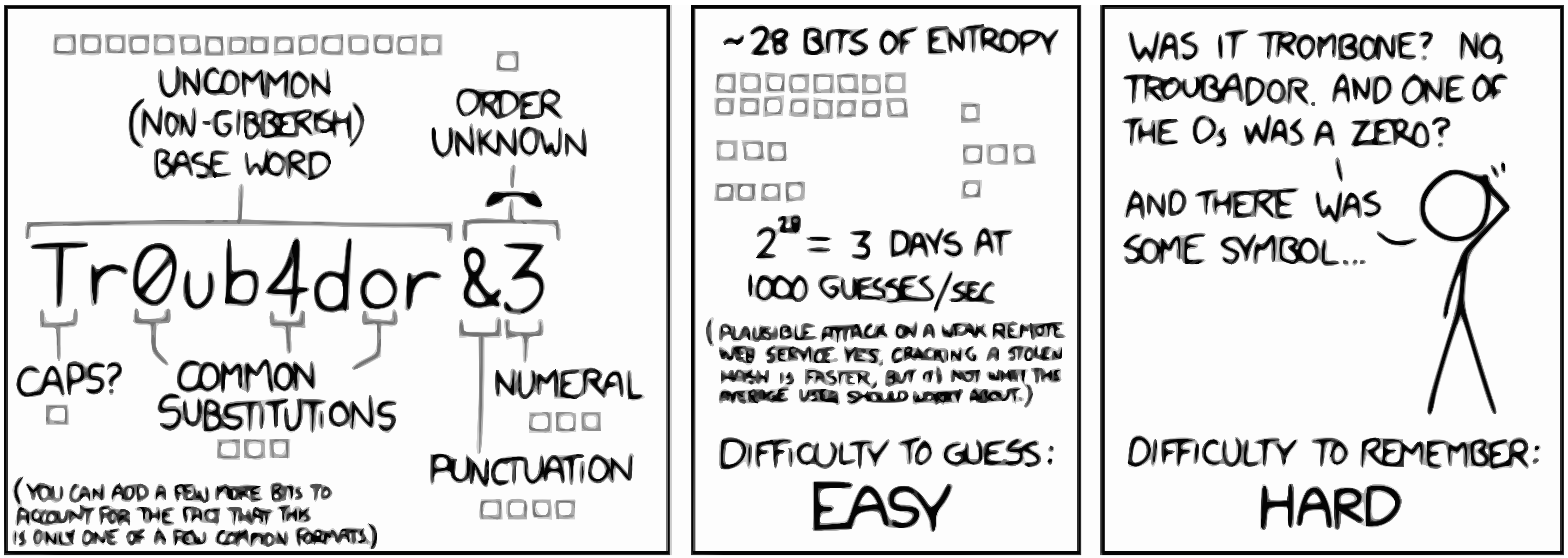
Password Strength

"Through 20 years of effort, we have successfully trained everyone to use passwords that are hard for humans to remember, but easy for computers to guess."

Randall Munroe, http://xkcd.com/936/

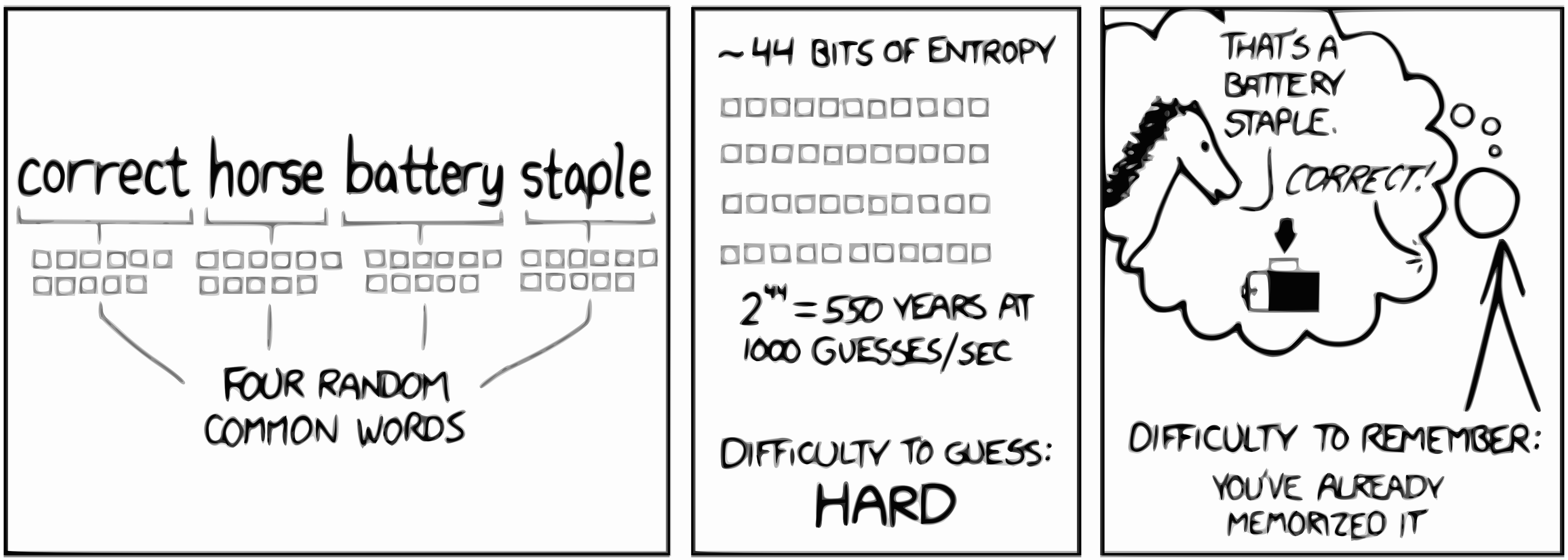
$$\frac{\text{password}}{\text{Tr0ub4dor\&3}} \quad \text{or} \quad \frac{\text{passphrase}}{\text{correct horse battery staple}}$$

Password



UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor&3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^{28} = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS:
EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER:
HARD

Passphrase


correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^{44} = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT

Alphabet

$\mathcal{A}$ countable set of **symbols**

Examples:

- $\mathbb{N}$: the non-negative integers,
- $\{0, 1\}$: the binary digits,
- letters, digits and punctuations marks,
- the english words.

Symbol Streams

finite sequences of symbols

$$\mathcal{A}^n = \overbrace{\mathcal{A} \times \cdots \times \mathcal{A}}^{n \text{ terms}}$$

$$a_0 a_1 \cdots a_{n-1} \in \mathcal{A}^n$$

$$\mathcal{A}^+ = \bigcup_{n=1}^{+\infty} \mathcal{A}^n$$

$$|a_0 a_1 \cdots a_{n-1}| = n$$

$$\mathcal{A}^* = \{\epsilon\} \cup \mathcal{A}^+$$

$$\epsilon : \text{empty sequence}$$

Codes

Variable-length, **binary**, **symbol code**:

$$c : \mathcal{A} \to \{0, 1\}^+$$

Usually implied: **non-ambiguous**:

$$c \text{ is injective.}$$

Extended as a **stream code**:

$$c : \mathcal{A}^+ \to \{0, 1\}^+$$

$$c(a_0 a_1 \cdots a_{n-1}) = c(a_0) c(a_1) \cdots c(a_{n-1})$$

Unicode

The Unicode Standard (6.3) consists of an alphabet of 110,187 characters among 1,114,112 possible **code points**.

Example:

∃ has the code point U+2203

mathematical operators

range: U+2200-U+22FF

http://unicode.org/charts/PDF/U2200.pdf

	220	221	222	223	224	225	226	227	228	229	22A	22B	22C	22D	22E	22F
0	∀ 2200	∐ 2210	∠ 2220	∰ 2230	≀ 2240	≐ 2250	≠ 2260	≰ 2270	⊀ 2280	⊐ 2290	⊠ 22A0	⊰ 22B0	⋀ 22C0	⋐ 22D0	⋠ 22E0	⋰ 22F0
1	∁ 2201	∑ 2211	∡ 2221	∱ 2231	≁ 2241	≑ 2251	≡ 2261	≱ 2271	⊁ 2281	⊑ 2291	⊡ 22A1	⊱ 22B1	⋁ 22C1	⋑ 22D1	⋡ 22E1	⋱ 22F1
2	∂ 2202	− 2212	∢ 2222	∲ 2232	≂ 2242	≒ 2252	≢ 2262	≲ 2272	⊂ 2282	⊒ 2292	⊢ 22A2	⊲ 22B2	⋂ 22C2	⋒ 22D2	⋢ 22E2	⋲ 22F2
3	∃ 2203	∓ 2213	∣ 2223	∳ 2233	≃ 2243	≓ 2253	≣ 2263	≳ 2273	⊃ 2283	⊓ 2293	⊣ 22A3	⊳ 22B3	⋃ 22C3	⋓ 22D3	⋣ 22E3	⋳ 22F3
4	∄ 2204	∔ 2214	∤ 2224	∴ 2234	≄ 2244	≔ 2254	≤ 2264	≴ 2274	⊄ 2284	⊔ 2294	⊤ 22A4	⊴ 22B4	⋄ 22C4	⋔ 22D4	⋤ 22E4	⋴ 22F4
5	∅ 2205	∕ 2215	∥ 2225	∵ 2235	≅ 2245	≕ 2255	≥ 2265	≵ 2275	⊅ 2285	⊕ 2295	⊥ 22A5	⊵ 22B5	⋅ 22C5	⋕ 22D5	⋥ 22E5	⋵ 22F5
6	∆ 2206	∖ 2216	∦ 2226	∶ 2236	≆ 2246	≖ 2256	≦ 2266	≶ 2276	⊆ 2286	⊖ 2296	⊦ 22A6	⊶ 22B6	⋆ 22C6	⋖ 22D6	⋦ 22E6	⋶ 22F6
7	∇ 2207	∗ 2217	∧ 2227	∷ 2237	≇ 2247	≗ 2257	≧ 2267	≷ 2277	⊇ 2287	⊗ 2297	⊧ 22A7	⊷ 22B7	⋇ 22C7	⋗ 22D7	⋧ 22E7	⋷ 22F7
8	∈ 2208	∘ 2218	∨ 2228	∸ 2238	≈ 2248	≘ 2258	≨ 2268	≸ 2278	⊈ 2288	⊘ 2298	⊨ 22A8	⊸ 22B8	⋈ 22C8	⋘ 22D8	⋨ 22E8	⋸ 22F8
9	∉ 2209	∙ 2219	∩ 2229	∹ 2239	≉ 2249	≙ 2259	≩ 2269	≹ 2279	⊉ 2289	⊙ 2299	⊩ 22A9	⊹ 22B9	⋉ 22C9	⋙ 22D9	⋩ 22E9	⋹ 22F9
A	∊ 220A	√ 221A	∪ 222A	∺ 223A	≊ 224A	≚ 225A	≪ 226A	≺ 227A	⊊ 228A	⊚ 229A	⊪ 22AA	⊺ 22BA	⋊ 22CA	⋚ 22DA	⋪ 22EA	⋺ 22FA
B	∋ 220B	∛ 221B	∫ 222B	∻ 223B	≋ 224B	≛ 225B	≫ 226B	≻ 227B	⊋ 228B	⊛ 229B	⊫ 22AB	⊻ 22BB	⋋ 22CB	⋛ 22DB	⋫ 22EB	⋻ 22FB
C	∌ 220C	∜ 221C	∬ 222C	∼ 223C	≌ 224C	≜ 225C	≬ 226C	≼ 227C	⊌ 228C	⊜ 229C	⊬ 22AC	⊼ 22BC	⋌ 22CC	⋜ 22DC	⋬ 22EC	⋼ 22FC
D	∍ 220D	∝ 221D	∭ 222D	∽ 223D	≍ 224D	≝ 225D	≭ 226D	≽ 227D	⊍ 228D	⊝ 229D	⊭ 22AD	⊽ 22BD	⋍ 22CD	⋝ 22DD	⋭ 22ED	⋽ 22FD
E	∎ 220E	∞ 221E	∮ 222E	∾ 223E	≎ 224E	≞ 225E	≮ 226E	≾ 227E	⊎ 228E	⊞ 229E	⊮ 22AE	⊾ 22BE	⋎ 22CE	⋞ 22DE	⋮ 22EE	⋾ 22FE
F	∏ 220F	∟ 221F	∯ 222F	∿ 223F	≏ 224F	≟ 225F	≯ 226F	≿ 227F	⊏ 228F	⊟ 229F	⊯ 22AF	⊿ 22BF	⋏ 22CF	⋟ 22DF	⋯ 22EF	⋿ 22FF

UTF-8

One of the most popular Unicode encoding.
Compatible with ASCII (U+0 - U+7F).

Range	Code Format					
U+0 – U+7f	0xxxxxxx					
U+80 – U+7ff	110xxxxx	10xxxxxx				
U+800 – U+ffff	1110xxxx	10xxxxxx	10xxxxxx			
U+100000 – U+1fffff	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
U+200000 – U+3ffffff	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
U+4000000 – U+7fffffff	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Example:

$\exists \longrightarrow$ U+2203 $\longrightarrow$ 00100010 00000011

$\longrightarrow$ 11100010 10001000 10000011

Stream Codes

Symbols codes shall be designed so that their stream code is non-ambiguous too. Such stream codes are **self-delimiting**.

The simplest self-delimiting codes are **prefix(-free) codes**:

$$\forall\, c_1, c_2 \in \{0,1\}^+,\ c_1 \in \operatorname{range} c \implies c_1 c_2 \notin \operatorname{range} c$$

Self-delimiting Codes

Examples: $\mathcal{A} = \{0, 1, 2, 3\}$

$$c : 0 \to 0,\ 1 \to 1,\ 2 \to 10,\ 3 \to 11$$

ambiguous stream code: consider 10.

$$c : 0 \to 0,\ 1 \to 01,\ 2 \to 011,\ 3 \to 0111$$

self-delimiting, but not prefix: 0 is a prefix of 01.

$$c : 0 \to 0,\ 1 \to 10,\ 2 \to 110,\ 3 \to 1110$$

prefix code (unary coding).

Kraft's Inequality

Let $\mathcal{A}$ be an alphabet and (l_a), $a \in \mathcal{A}$ be a family of positive lengths. There exist a self-delimiting stream code c on $\mathcal{A}$ such that

$$\forall\, a \in \mathcal{A},\ |c(a)| = l_a$$

if and only if we have

$$K = \sum_{a \in \mathcal{A}} 2^{-l_a} \leq 1$$

Moreover, if the inequality holds, the code can be selected prefix-free.

Brainf*ck

A Turing-complete programming language with only 8 commands (Urban Müller, 1993):

```
>   <   +   -   .   ,   [   ]
```

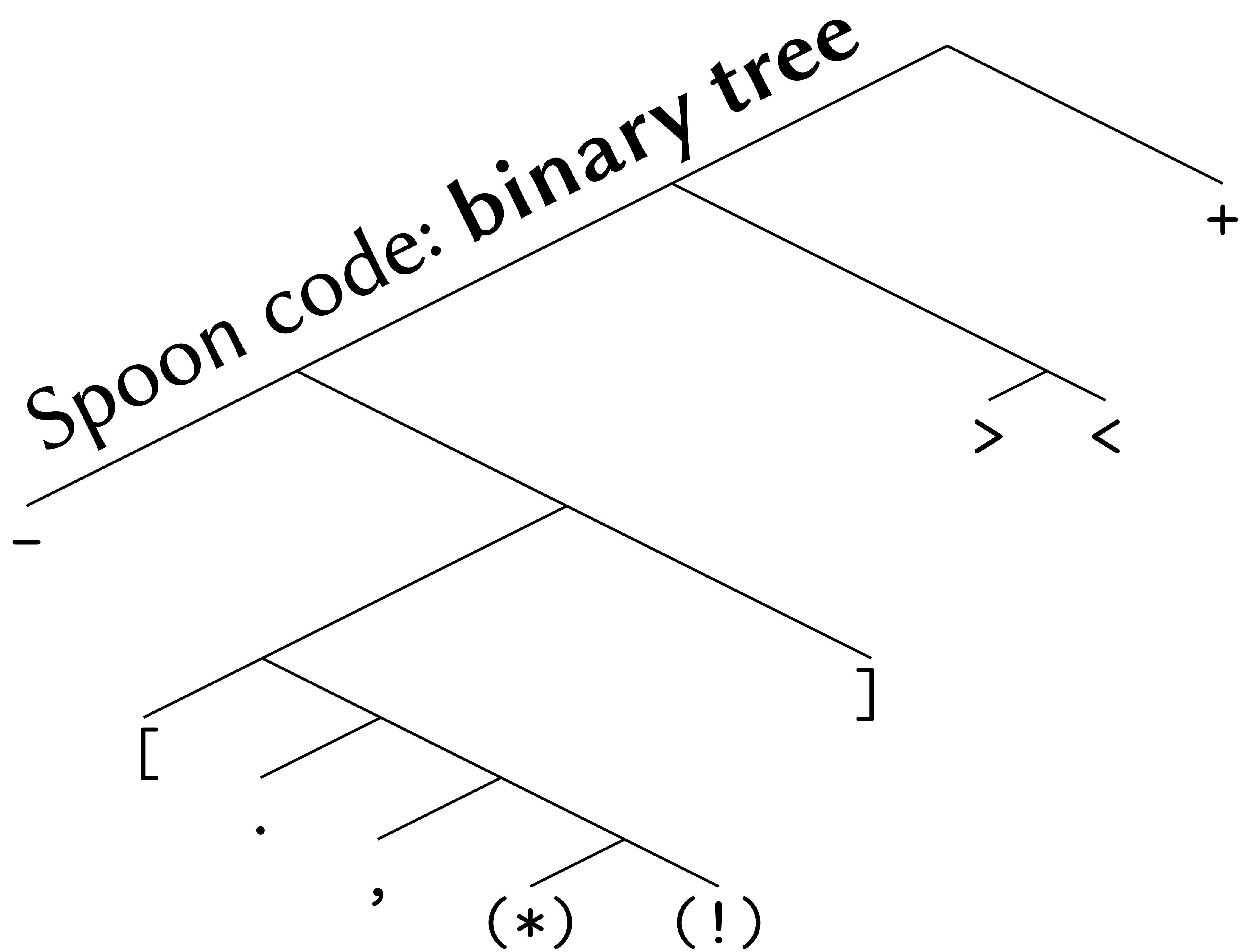
"Hello World!" program:

```
++++++++++[>+++++++>+++
+++++++>+++>+<<<<-]>++.
>+.+++++++..+++.>++.<<+
++++++++++++++.>.+++.--
----.--------.>+.>.
```

Codes as Trees

Spoon code: **table**

`>`	$\rightarrow$	010
`<`	$\rightarrow$	011
`+`	$\rightarrow$	1
`-`	$\rightarrow$	000
`.`	$\rightarrow$	001010
`,`	$\rightarrow$	0010110
`[`	$\rightarrow$	00100
`]`	$\rightarrow$	0011



Steven Goodwin, 1998.

Code Length

Spoon code:

>	$\rightarrow$	010
<	$\rightarrow$	011
+	$\rightarrow$	1
-	$\rightarrow$	000
.	$\rightarrow$	001010
,	$\rightarrow$	0010110
[	$\rightarrow$	00100
]	$\rightarrow$	0011

Fork code:

>	$\rightarrow$	000
<	$\rightarrow$	001
+	$\rightarrow$	010
-	$\rightarrow$	011
.	$\rightarrow$	100
,	$\rightarrow$	101
[	$\rightarrow$	110
]	$\rightarrow$	111

"Hello World!" binary code:

- Spoon: 245 bits,
- Fork: 333 bits (+36%).

Optimal Code Length

Let A be a random symbol in $\mathcal{A}$, c a code for $\mathcal{A}$.
The **average code (bit-)length** of c is:

$$\mathbb{E}|c(A)| = \sum_{a \in \mathcal{A}} p(a)|c(a)|$$

Every prefix code c satisfies:

$$H(A) \leq \mathbb{E}|c(A)|$$

Moreover, there is a prefix code c such that:

$$\mathbb{E}|c(A)| < H(A) + 1$$

Huffman: Data Structures

Weighted Alphabets:

{'a': 0.5, 'b': 0.3, 'c': 0.2}

Weighted Binary Trees:

- terminal nodes:

  ('a', 0.5)

- non-terminal nodes:

  ([node1, node2], 0.5)

Huffman: Node Helpers

```
class Node(object):
    "Manage nodes as (symbol, weight) pairs"
    @staticmethod
    def symbol(node):
        return node[0]
    @staticmethod
    def weight(node):
        return node[1]
    @staticmethod
    def is_terminal(node):
        return not isinstance(Node.symbol(node), list)
```

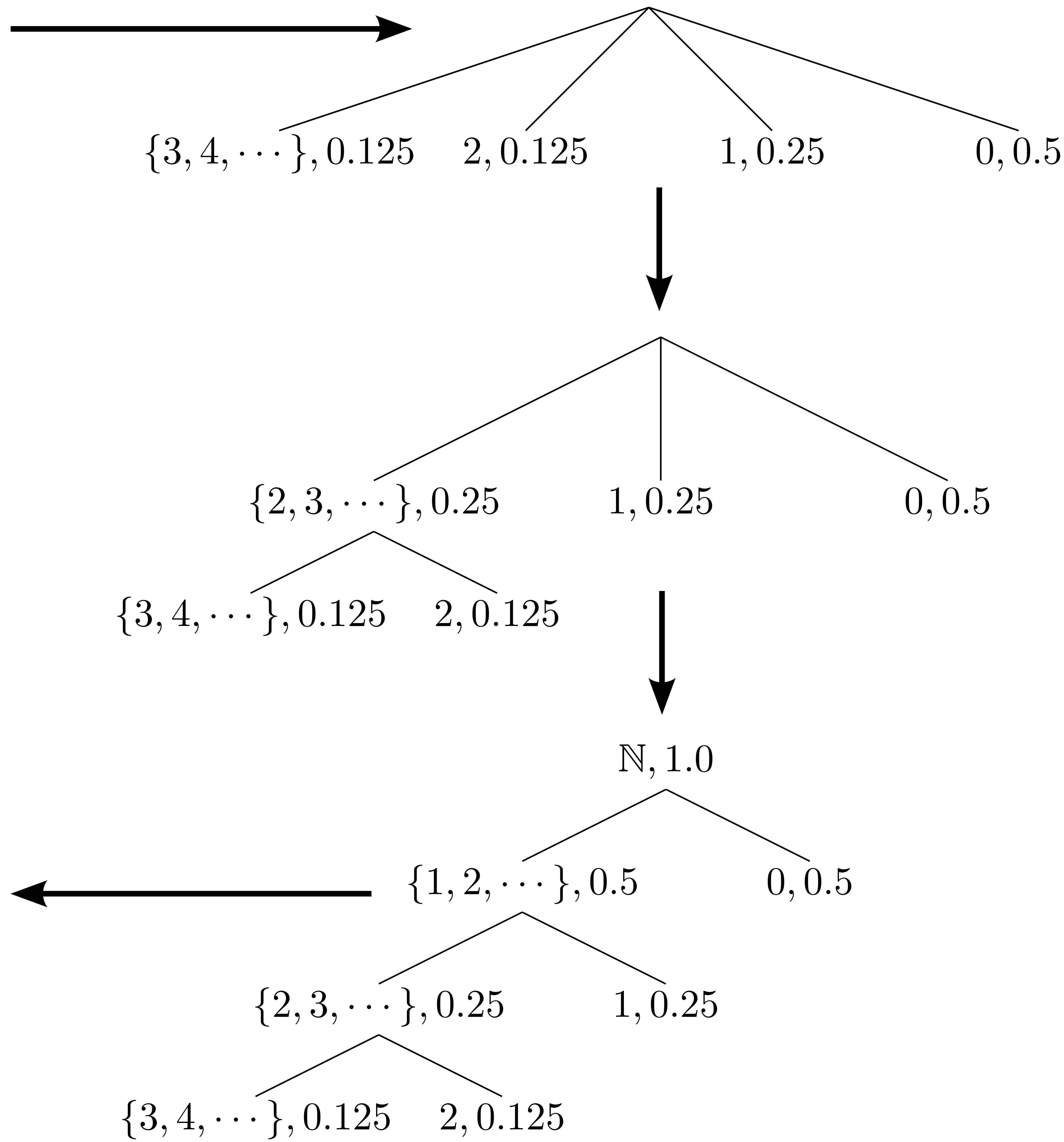
Huffman's Algorithm

```
class Huffman(object):
  @staticmethod
  def make_binary_tree(alphabet):
    nodes = alphabet.items()
    while len(nodes) > 1:
      nodes.sort(key=Node.weight)
      node1, node2 = nodes.pop(0), nodes.pop(0)
      node = ([node1, node2],
              Node.weight(node1) + Node.weight(node2))
      nodes.insert(0, node)
    return nodes[0]
```

Huffman's Algorithm

$$\mathcal{A} = \{0, 1, 2, \{3, 4, \cdots\}\}$$

$$p(0) = 0.5$$
$$p(1) = 0.25$$
$$p(2) = 0.125$$
$$p(\{3, 4, \cdots\}) = 0.125$$



unary coding

(separator 1)

$$\begin{array}{rcr} 0 & \to & 1 \\ 1 & \to & 01 \\ 2 & \to & 001 \\ \{3, 4, \cdots\} & \to & 000 \end{array}$$

Rice Coding

Consider a set of non-negative integers n, that almost fit into a b-bit fixed-size coding.

The Rice coding with parameter b of n

$n \bmod 2^b$	$\left\lfloor \frac{n}{2^b} \right\rfloor$
FIXED-SIZE	UNARY

almost achieves a coding in $b + 1$ bits.
If $b = 0$, we end up with unary coding.

Rice Coding - Example

$n = 0, 1, 2, 3$ most of the time,
$b = 2$ is a sensible choice.

$$n = 0 = 0 \times 2^2 + 0 \quad \rightarrow \quad 00|0$$

$$n = 3 = 0 \times 2^2 + 3 \quad \rightarrow \quad 11|0$$

$$n = 9 = 2 \times 2^2 + 1 \quad \rightarrow \quad 01|110$$

$$n = 15 = 3 \times 2^2 + 3 \rightarrow \quad 11|1110$$

Rice Coding Parameter

Heuristic

Define $\theta = \dfrac{m}{1+m}$ with $m \approx \mathbb{E}\, n$,

then select

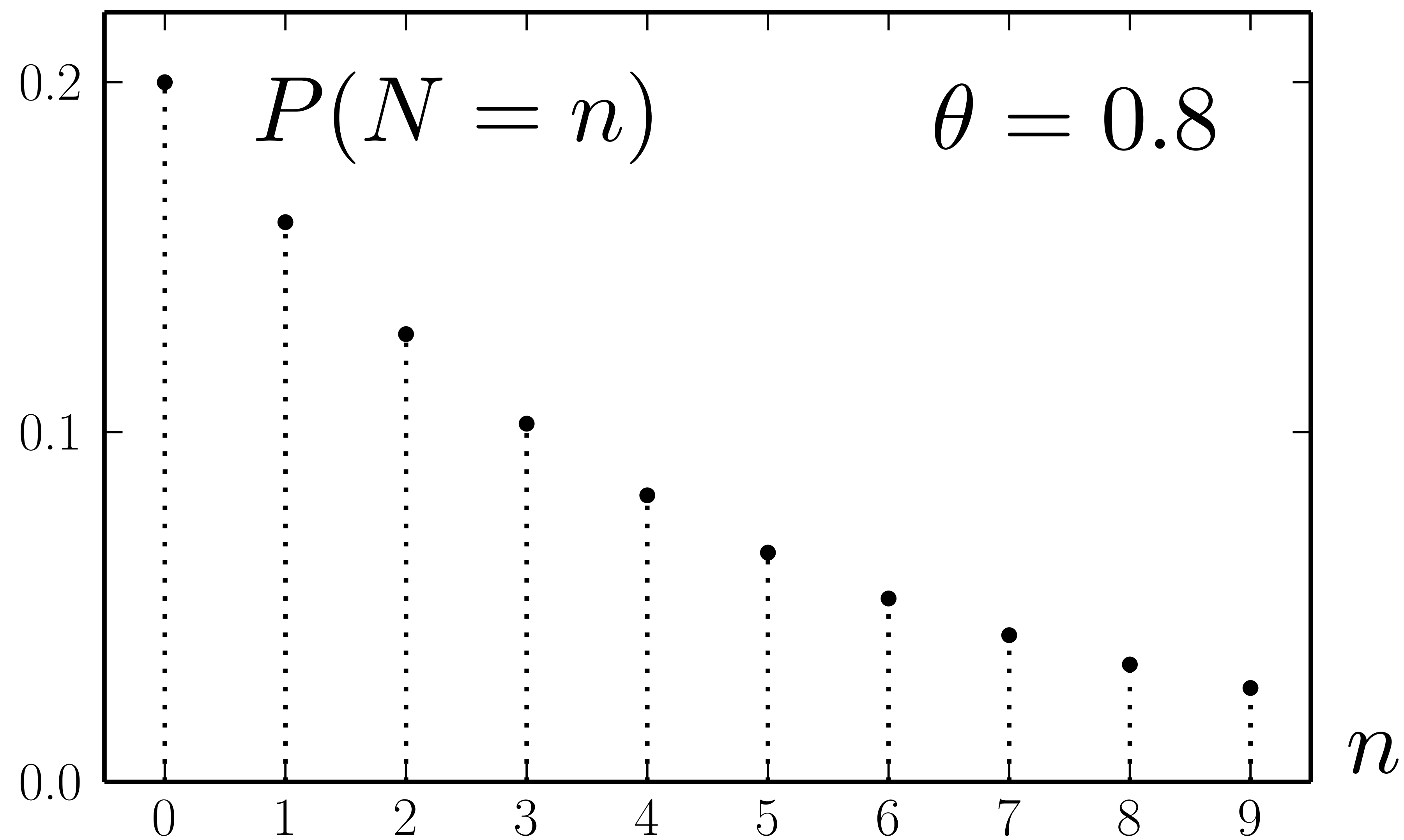
$$b = \max\left[0, 1 + \left\lfloor \log_2 \left(\frac{\log(\phi - 1)}{\log \theta} \right) \right\rfloor \right]$$

with $\phi = \dfrac{1+\sqrt{5}}{2}$.

Geometric Distributions

N: random symbol in $\mathcal{A} = \mathbb{N}$ such that:

$$P(N = n) \propto \theta^n,\ \theta \in (0, 1)$$



Unary coding is optimal if $\theta = 0.5$.

Geometric Distributions

Optimal Code

Compute $l = \min\{i \in \mathbb{N}^\star \mid \theta^i + \theta^{i+1} \leq 1\}$.

If $l = 1$ the unary code is optimal.

Otherwise, divide n by l

$$n = \lfloor n/l \rfloor \times l + n \bmod l$$

and concatenate the two codes:

$$n \to n \bmod l \to \text{Huffman code}$$
$$n \to \lfloor n/l \rfloor \to \text{unary code}$$

N.B. $P(N \bmod l = n) \propto \theta^n,\ n = 0, \cdots, l-1$

Golomb Power of 2

The Huffman code of $n \bmod l$ is almost fixed-length.
In particular, if $l = 2^b$, it has the fixed length b.
Hence, we approximate l as a power of two to replace the Huffman code by a fixed-length code.

This is precisely Rice coding (aka GPO2) !

Unary Coder

```
def unary_symbol_encoder(stream, symbol):
    bools = symbol * [True] + [False]
    return stream.write(bools)
```

```
def unary_symbol_decoder(stream):
    count = 0
    while stream.read(bool) is True:
        count += 1
    return count
```

```
unary_encoder = stream_encoder(unary_symbol_encoder)
unary_decoder = stream_decoder(unary_symbol_decoder)
class unary(object):
    pass
bitstream.register(unary, reader=unary_encoder, writer=unary_decoder)
```

Unary Coder: usage

```
>>> stream = BitStream()
>>> stream.write([0,1,2,3], unary)
>>> print stream
0101101110
>>> stream.read(unary, 4)
[0, 1, 2, 3]
```

Rice Codec Parameters

```
class rice(object):
  "Rice codec tag type"

  def __init__(self, b, signed):
    self.b, self.signed = b, signed

  def from_frame(frame, signed):
    "Return a rice tag from a sample frame."
    ...
```

number of bits used for the fixed-width encoding.

shall we encode the data sign ?

Rice Codec

```
def rice_symbol_encoder(rice_tag):
  def encoder(stream, symbol):
    if rice_tag.signed:
      stream.write(symbol < 0)
    symbol = abs(symbol)
    l = 2 ** rice_tag.b
    remain, fixed = divmod(symbol, l)
    fixed_bits = []
    for _ in range(rice_tag.b):
      fixed_bits.insert(0, bool(fixed % 2))
      fixed = fixed >> 1
    stream.write(fixed_bits)
    stream.write(remain, unary)
  return encoder
```

```
def rice_symbol_decoder(rice_tag):
  def decoder(stream):
    if rice_tag.signed and stream.read(bool):
      sign = -1
    else:
      sign = 1
    fixed_number = 0
    for _ in range(rice_tag.b):
      bit = int(stream.read(bool))
      fixed_number = (fixed_number << 1) + bit
    l = 2 ** rice_tag.b
    remain_number = l * stream.read(unary)
    return sign * (fixed_number + remain_number)
  return decoder
```

Rice Codec: usage

```
>>> data = [0, 8, 0, 8, 16, 0, 32, 0, 16, 8, 0, 8]
>>> rice_tag = rice.from_frame(data, signed=False)
>>> rice_tag.b
3
>>> stream = BitStream()
>>> stream.write(data, rice_tag)
>>> stream
000000010000000010000110000000011
110000000011000010000000010
>>> stream.read(rice_tag, 12)
[0, 8, 0, 8, 16, 0, 32, 0, 16, 8, 0, 8]
```

Rice Coder: test

```
>>> for b in range(7):
...     stream = BitStream(data, rice(b=b, signed=False))
...     print "rice b={0}: {1} bits".format(b, len(stream))
...
rice b=0: 108 bits
rice b=1: 72 bits
rice b=2: 60 bits
rice b=3: 60 bits
rice b=4: 64 bits
rice b=5: 73 bits
rice b=6: 84 bits
```