

What is “Information”?

Information, *n.* Data communicated or received that resolves uncertainty about a particular fact or circumstance.

Example: you receive some data about a card drawn at random from a 52-card deck. Which of the following data conveys the most information? The least?

↙ # of possibilities remaining

- 13** A. The card is a heart
- 51** B. The card is not the Ace of spades
- 12** C. The card is a face card (J, Q, K)
- 1** D. The card is the “suicide king”



Quantifying Information

(Claude Shannon, 1948)

Given discrete random variable X

- N possible values: $x_1, x_2, \dots, x_N$
- Associated probabilities: $p_1, p_2, \dots, p_N$

Information received when learning that choice was x_i :

$$I(x_i) = \log_2 \left(\frac{1}{p_i} \right)$$

$1/p_i$ is proportional to the uncertainty of choice x_i .



Information is measured in bits (binary digits) = number of 0/1's required to encode choice(s)



Information Conveyed by Data

Even when data doesn't resolve all the uncertainty

$$I(\text{data}) = \log_2 \left(\frac{1}{p_{\text{data}}} \right) \quad \text{e.g., } I(\text{heart}) = \log_2 \left(\frac{1}{13/52} \right) = 2 \text{ bits}$$

Common case: Suppose you're faced with N equally probable choices, and you receive data that narrows it down to M choices. The probability that data would be sent is $M \cdot (1/N)$ so the amount of information you have received is

$$I(\text{data}) = \log_2 \left(\frac{1}{M \cdot (1/N)} \right) = \log_2 \left(\frac{N}{M} \right) \text{ bits}$$

Example: Information Content

Examples:

- information in one coin flip:

$$N = 2 \quad M = 1 \quad \text{Info content} = \log_2(2/1) = 1 \text{ bit}$$

- card drawn from fresh deck is a heart:

$$N = 52 \quad M = 13 \quad \text{Info content} = \log_2(52/13) = 2 \text{ bits}$$

- roll of 2 dice:

$$N = 36 \quad M = 1 \quad \text{Info content} = \log_2(36/1) = 5.17$$

.17 bits ???



Probability & Information Content



data	p_{data}	$\log_2(1/p_{\text{data}})$
a heart	13/52	2 bits
not the Ace of spades	51/52	0.028 bits
a face card (J, Q, K)	12/52	2.115 bits
the “suicide king”	1/52	5.7 bits



— *Shannon's definition for information content lines up nicely with my intuition: I get more information when the data resolves more uncertainty about the randomly selected card.*

Entropy

In information theory, the **entropy** $H(X)$ is the average amount of information contained in each piece of data received about the value of X :

$$H(X) = E(I(X)) = \sum_{i=1}^N p_i \cdot \log_2 \left(\frac{1}{p_i} \right)$$

Example: $X = \{A, B, C, D\}$

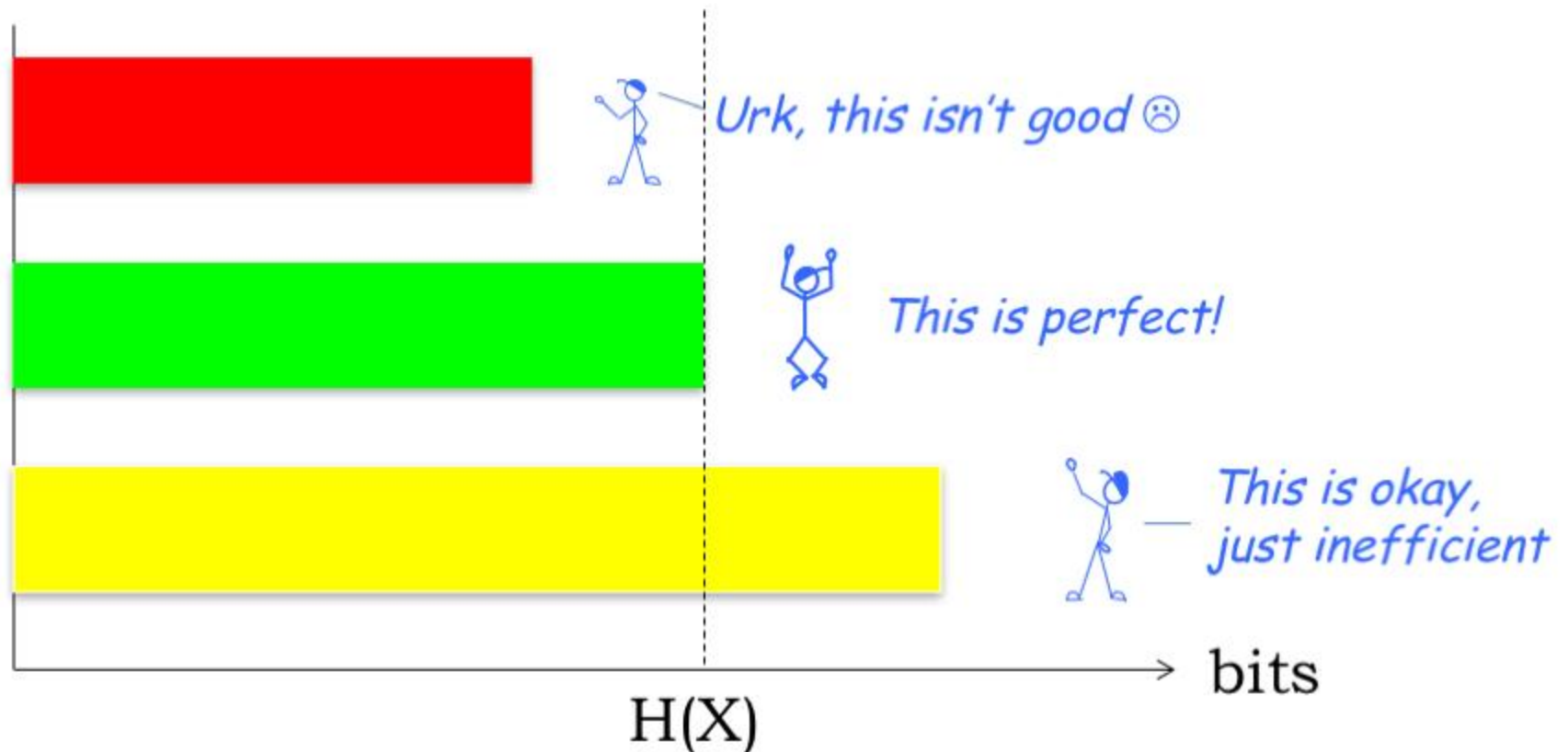
$choice_i$	p_i	$\log_2(1/p_i)$
"A"	1/3	1.58 bits
"B"	1/2	1 bit
"C"	1/12	3.58 bits
"D"	1/12	3.58 bits

$$\begin{aligned} H(X) &= (1/3)(1.58) + \\ &\quad (1/2)(1) + \\ &\quad 2(1/12)(3.58) \\ &= 1.626 \text{ bits} \end{aligned}$$

Meaning of Entropy

Suppose we have a data sequence describing the values of the random variable X .

Average number of bits used to transmit choice



Encodings

An **encoding** is an *unambiguous* mapping between bit strings and the set of possible data.

Encoding for each symbol				Encoding for "ABBA"
A	B	C	D	
00	01	10	11	00 01 01 00
01	1	000	001	01 1 1 01
0	1	10	11	0 1 1 0  ABBA? ABC? ADA?



Encodings as Binary Trees

It's helpful to represent an unambiguous encoding as a binary tree with the symbols to be encoded as the leaves. The labels on the path from the root to the leaf give the encoding for that leaf.

Encoding

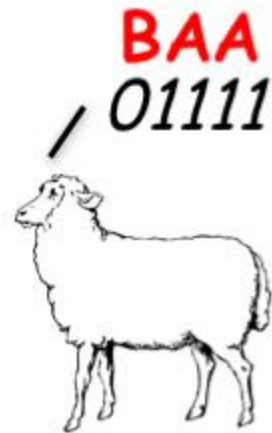
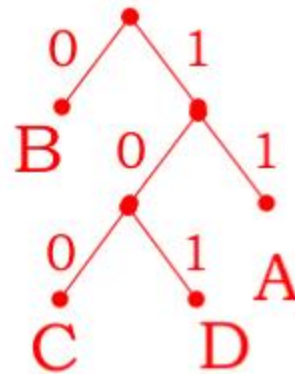
$B \leftrightarrow 0$

$A \leftrightarrow 11$

$C \leftrightarrow 100$

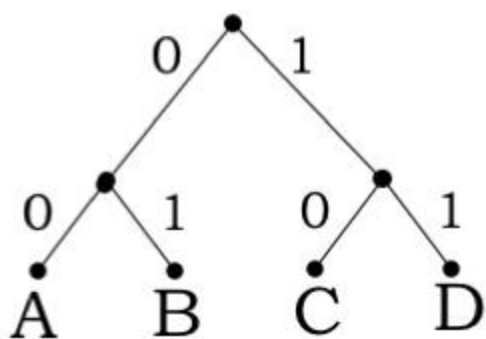
$D \leftrightarrow 101$

Binary tree



Fixed-length Encodings

If all choices are **equally likely** (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code will use at least enough bits to represent the information content.



All leaves have the same depth!

Note that the entropy for N equally-probable symbols is

$$\sum_{i=1}^N \left(\frac{1}{N}\right) \log_2 \left(\frac{1}{\frac{1}{N}}\right) = \log_2(N)$$

Examples:

Fixed-length are often a little inefficient...



- 4-bit binary-coded decimal (BCD) digits $\log_2(10)=3.322$
- 7-bit ASCII for printing characters $\log_2(94)=6.555$

Encoding Positive Integers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an N-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

$$\begin{aligned} V &= 0 \cdot 2^{11} + 1 \cdot 2^{10} + 1 \cdot 2^9 + \dots \\ &= 1024 + 512 + 256 + 128 + 64 + 16 \\ &= 2000 \end{aligned}$$

Smallest number: 0

Largest number: $2^N - 1$

Hexademical Notation

Long strings of binary digits are tedious and error-prone to transcribe, so we usually use a higher-radix notation, choosing the radix so that it's simple to recover the original bits string.

A popular choice is transcribe numbers in base-16, called hexadecimal, where each group of 4 adjacent bits are representated as a single hexadecimal digit.

Hexadecimal - base 16

0000	-	0	1000	-	8
0001	-	1	1001	-	9
0010	-	2	1010	-	A
0011	-	3	1011	-	B
0100	-	4	1100	-	C
0101	-	5	1101	-	D
0110	-	6	1110	-	E
0111	-	7	1111	-	F

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

7

D

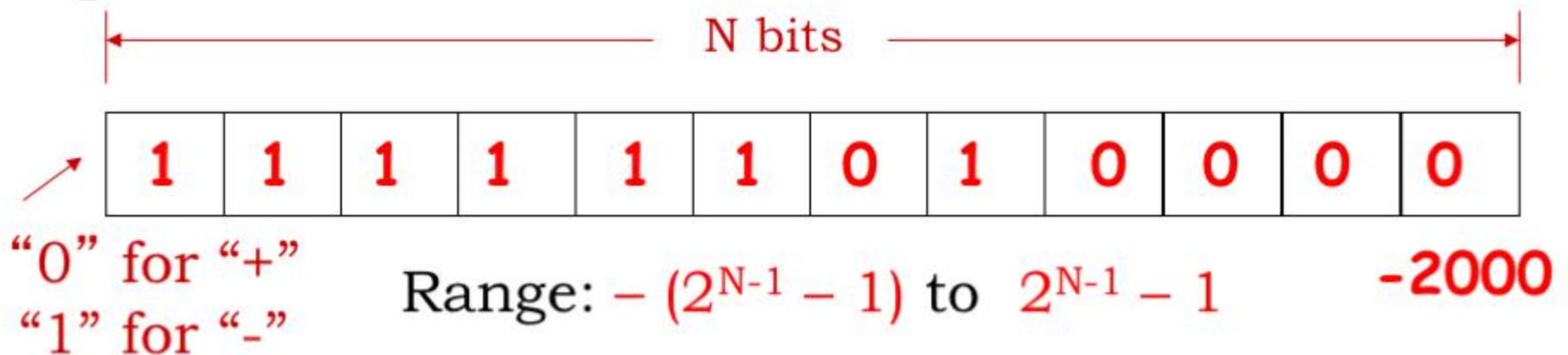
0

0b01111010000 = 0x7D0

Encoding Signed Integers

We use a signed magnitude representation for decimal numbers, encoding the sign of the number (using “+” and “-”) separately from its magnitude (using decimal digits).

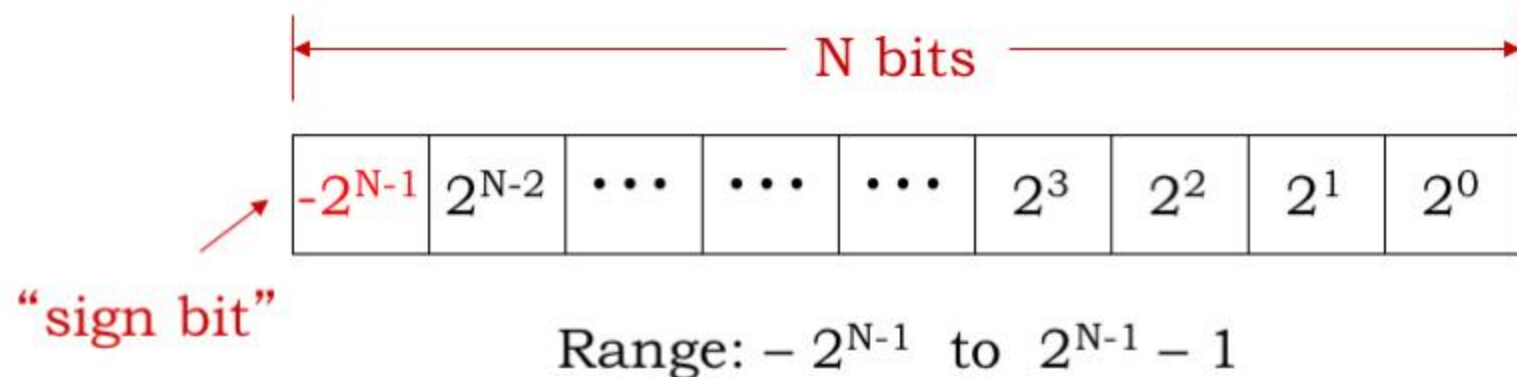
We could adopt that approach for binary representations:



But: two representations for 0 (+0, -0) and we’d need different circuitry for addition and subtraction

Two's Complement Encoding

In a two's complement encoding, the high-order bit of the N-bit representation has negative weight:



- Negative numbers have "1" in the high-order bit
- Most negative number: $10\dots0000$ -2^{N-1}
- Most positive number: $01\dots1111$ $+2^{N-1} - 1$
- If all bits are 1: $11\dots1111$ -1
- If all bits are 0: $00\dots0000$ 0

More Two's Complement

- Let's see what happens when we add the N-bit values for -1 and 1, keeping an N-bit answer:

$$\begin{array}{r} 11\dots1111 \\ +00\dots0001 \\ \hline 00000000 \end{array}$$



Just use ordinary binary addition, even when one or both of the operands are negative. 2's complement is perfect for N-bit arithmetic!

- To compute $B-A$, we'll just use addition and compute $B+(-A)$. But how do we figure out the representation for $-A$?

$$A+(-A) = 0 = 1 + -1$$

$$\begin{aligned} -A &= (-1 - A) + 1 \\ &= \sim A + 1 \end{aligned}$$

$$\begin{array}{r} 1 \\ -A_i \\ \hline \sim A_i \end{array}$$



To negate a two's complement value: bitwise complement and add 1.

Variable-length Encodings

We'd like our encodings to use bits efficiently:

GOAL: When encoding data we'd like to match the length of the encoding to the information content of the data.

On a practical level this means:

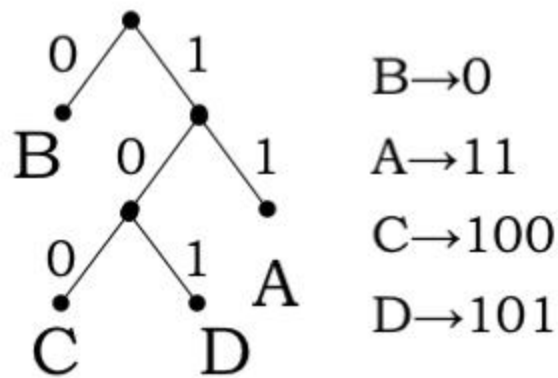
- Higher probability → shorter encodings
- Lower probability → longer encodings

Such encodings are termed **variable-length encodings**.

Example

$choice_i$	p_i	encoding
“A”	1/3	11
“B”	1/2	0
“C”	1/12	100
“D”	1/12	101

Entropy: $H(X) = 1.626$ bits



High probability,
Less information

Low probability,
More information

010011011101
 B C A B A D

Expected length of this encoding:

$$(2)(1/3) + (1)(1/2) + (3)(1/12)(2) = 1.667 \text{ bits}$$

Expected length for 1000 symbols:

- With fixed-length, 2 bits/symbol = 2000 bits
- With variable-length code = 1667 bits
- Lower bound (entropy) = 1626 bits

Huffman's Algorithm

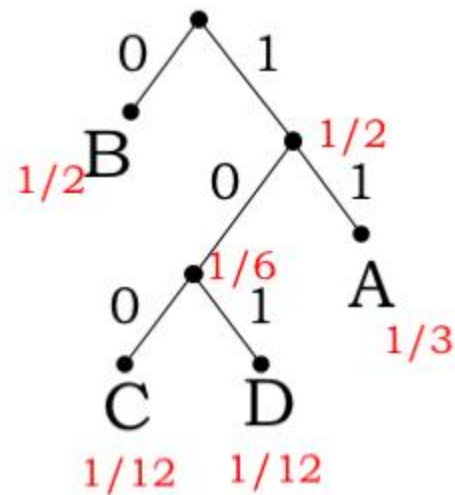
Given a set of symbols and their probabilities, constructs an optimal variable-length encoding.

Huffman's Algorithm:

- Build subtree using 2 symbols with lowest p_i
- At each step choose two symbols/subtrees with lowest p_i , combine to form new subtree
- Result: optimal tree built from the bottom-up

Example:

~~A=1/3~~, B=1/2, ~~C=1/12~~, ~~D=1/12~~



Can We Do Better?

Huffman's Algorithm constructed an optimal encoding... does that mean we can't do better?

To get a more efficient encoding (closer to information content) we need to encode **sequences of choices**, not just each choice individually. This is the approach taken by most file compression algorithms...

AA=1/9, AB=1/6, AC=1/36, AD=1/36
BA=1/6, BB=1/4, BC=1/24, BD=1/24
CA=1/36, CB=1/24, CC=1/144, CD=1/144
DA=1/36, DB=1/24, DC=1/144, DD=1/144

Using Huffman's Algorithm on pairs:
Average bits/symbol = 1.646 bits

*Lookup "LZW"
on Wikipedia*



Error Detection and Correction

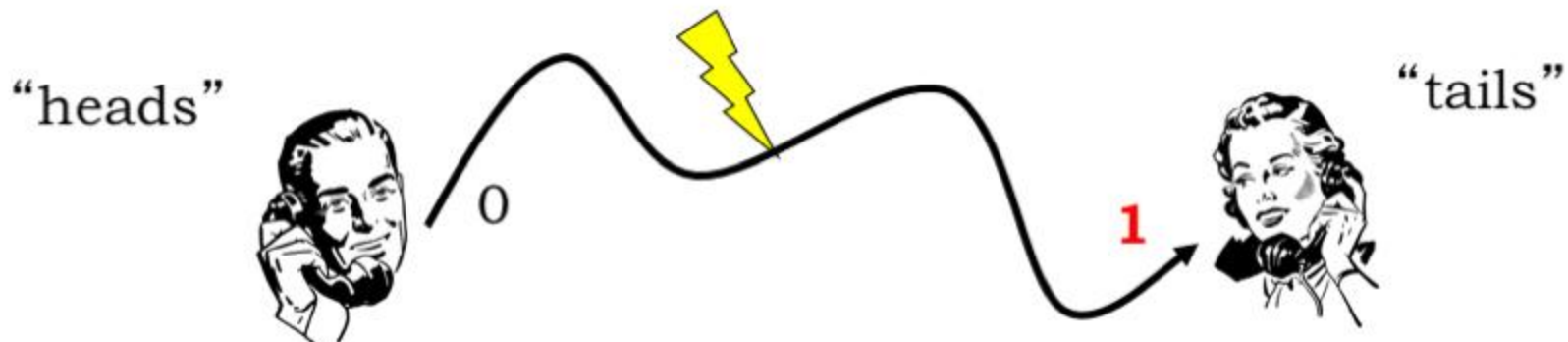
Suppose we wanted to reliably transmit the result of a single coin flip:

Heads: “0”

Tails: “1”



Further suppose that during processing a **single-bit error** occurs, i.e., a single “0” is turned into a “1” or a “1” is turned into a “0”.



Hamming Distance

HAMMING DISTANCE: The number of positions in which the corresponding digits differ in two encodings of the same length.

0	1	1	0	0	1	0
		↑		↑		
0	1	0	0	1	1	0

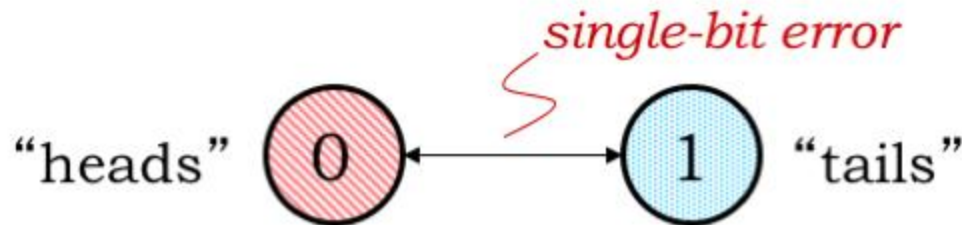


*Differs in 2 positions so
Hamming distance is 2...*

Hamming Distance & Bit Errors

The Hamming distance between a valid binary code word and the same code word with a single-bit error is 1.

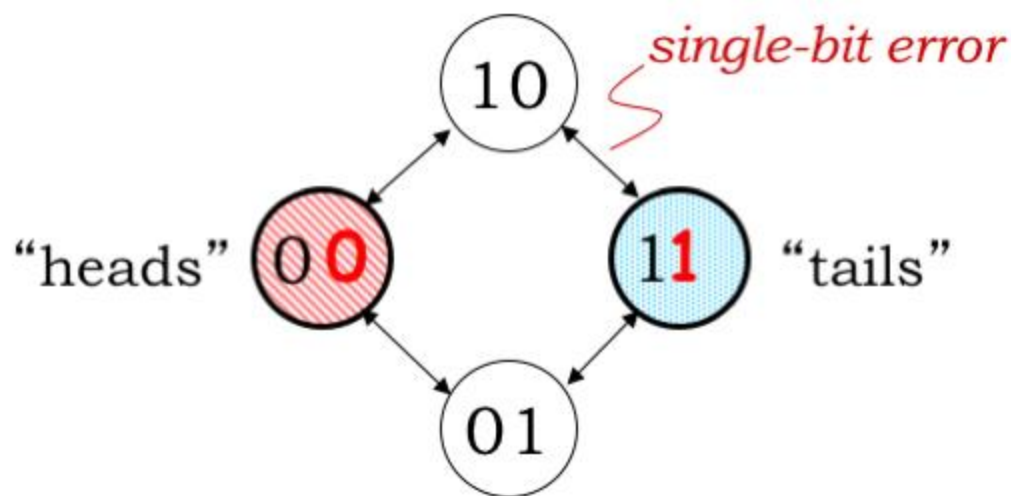
The problem with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word...



Single-bit Error Detection



What we need is an encoding where a single-bit error does *not* produce another valid code word.



A parity bit can be added to any length message and is chosen to make the total number of "1" bits even (aka "even parity"). If $\min \text{HD}(\text{code words}) = 1$, then $\min \text{HD}(\text{code words} + \text{parity}) = 2$.

Parity check = Detect Single-bit errors

- To check for a single-bit error (actually any odd number of errors), count the number of 1s in the received message and if it's odd, there's been an error.

0 1 1 0 0 1 0 1 0 0 1 1 → original word with parity

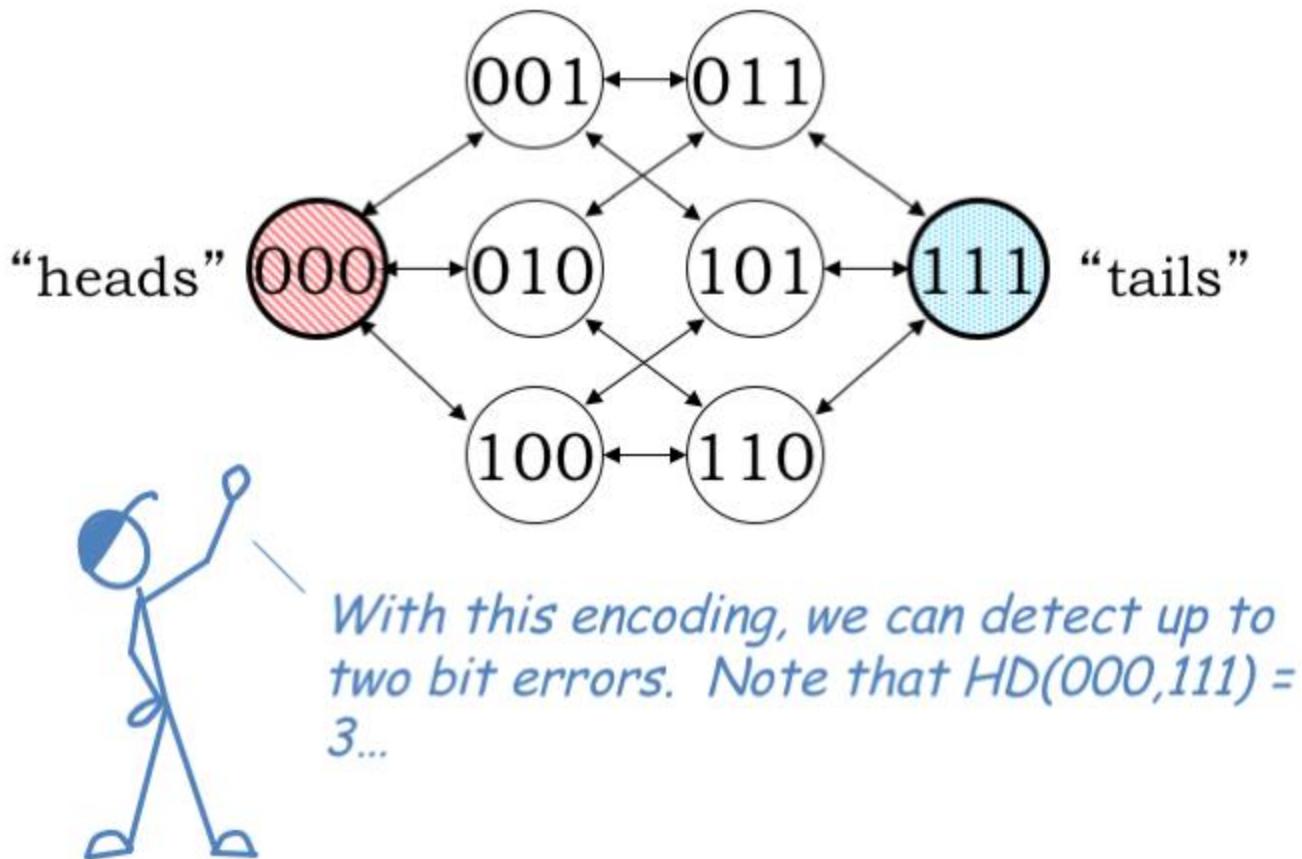
0 1 1 0 0 0 0 1 0 0 1 1 → single-bit error (detected)

0 1 1 0 0 0 1 1 0 0 1 1 → 2-bit error (not detected)

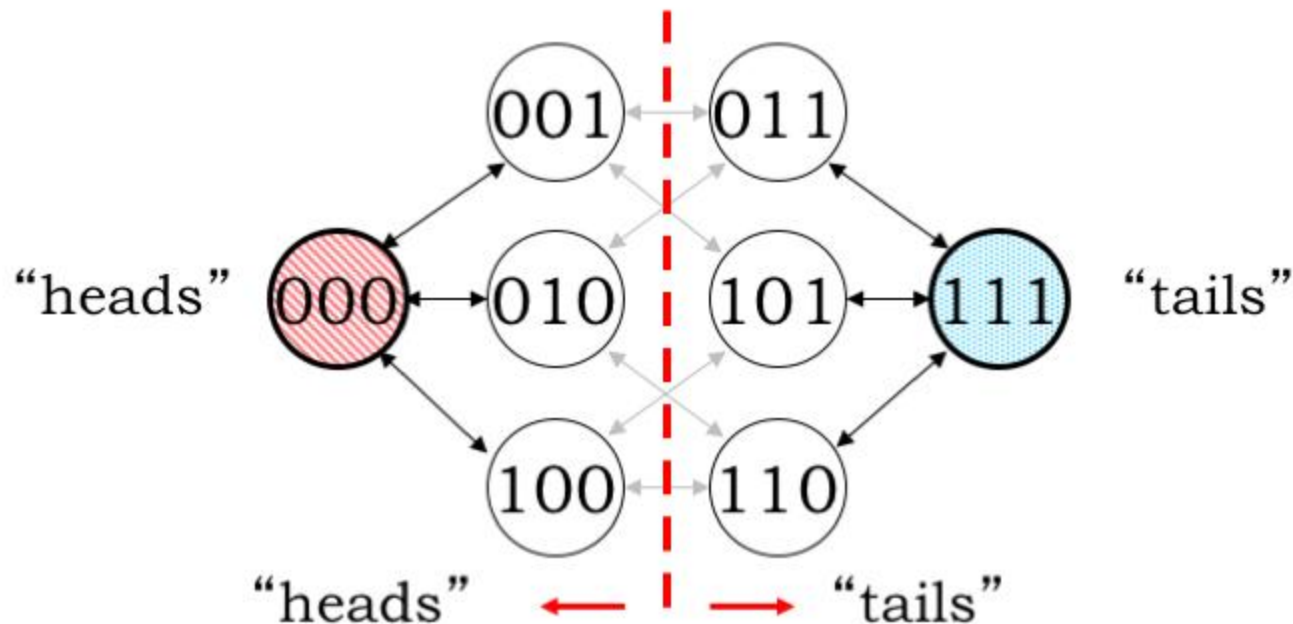
- One can “count” by summing the bits in the word modulo 2 (which is equivalent to XOR’ing the bits together).

Detecting Multi-bit Errors

To **detect** E errors, we need a minimum Hamming distance of $E+1$ between code words.



Single-bit Error Correction



By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don't overlap. So assuming at most one error, we can perform *error correction* since we can tell what the valid code was before the error happened.

To *correct* E errors, we need a minimum Hamming distance of $2E+1$ between code words.

Summary

- Information resolves uncertainty
- Choices equally probable:
 - N choices down to $M \Rightarrow \log_2(N/M)$ bits of information
 - use fixed-length encodings
 - encoding numbers: 2's complement signed integers
- Choices not equally probable:
 - choice _{i} with probability $p_i \Rightarrow \log_2(1/p_i)$ bits of information
 - average amount of information = $H(X) = \sum p_i \log_2(1/p_i)$
 - use variable-length encodings, Huffman's algorithm
- To detect E -bit errors: Hamming distance $> E$
- To correct E -bit errors: Hamming distance $> 2E$

Next time:

- encoding information electrically
- the digital abstraction
- combinational devices