4-1-1 Information

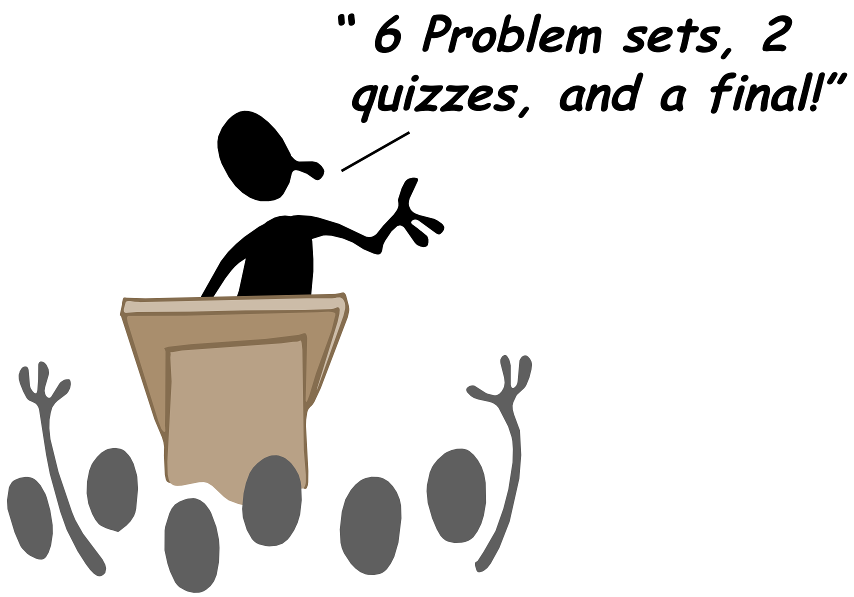
"2 bits, 4 bits, 6 bits a Byte"

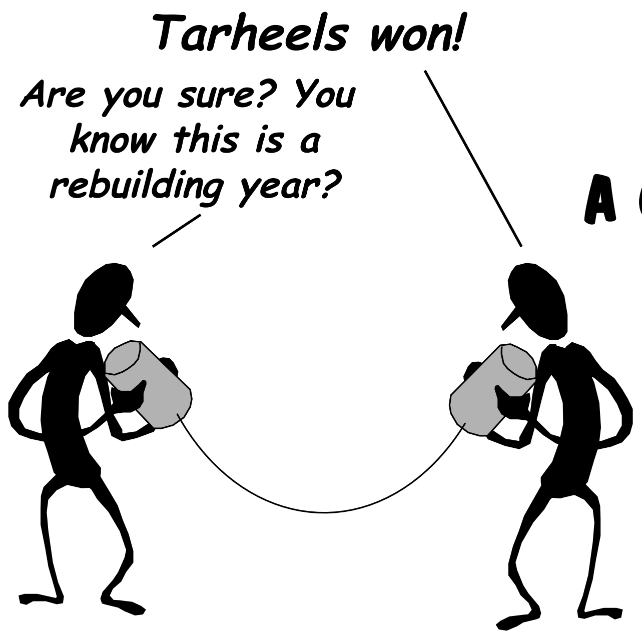
- Representing information using bits
- Number representations
- Some other bits

• Chapter 3.1-3.3

What is "Information"?

information, *n.* Knowledge communicated or received concerning a particular fact or circumstance.





A Computer Scientist's Definition:

Information resolves uncertainty.

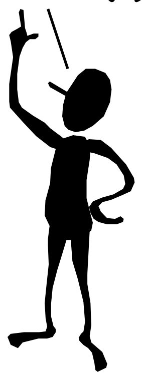
Information is simply that which cannot be predicted. *The less predictable a message is, the more information it conveys!*

Quantifying Information
(Claude Shannon, 1948)

Suppose you're faced with N equally probable choices, and I give you a fact that narrows it down to M choices. Then you've been given:

$\log_2(N/M)$ bits of information

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

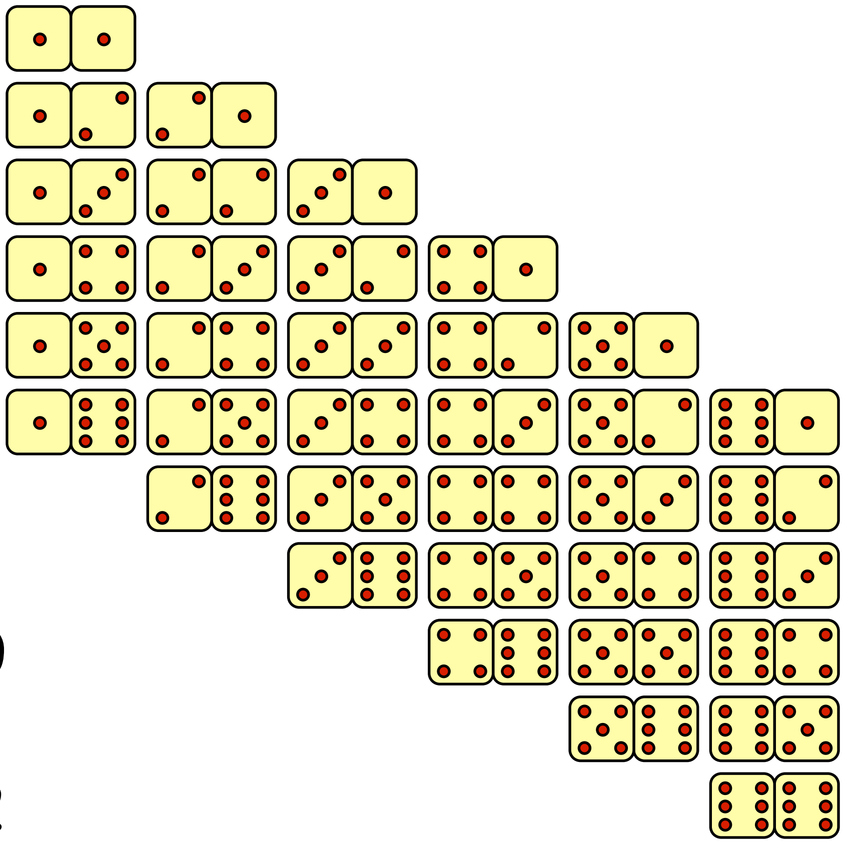


Examples:

- information in one coin flip: $\log_2(2/1) = 1$ bit
- roll of a single die: $\log_2(6/1) = \sim 2.6$ bits
- outcome of a Football game: 1 bit

  (well, actually, "they won" may convey more information if they were "expected" to lose…)

Example: Sum of 2 dice



2	$i_2 = \log_2(36/1) = 5.170$ bits
3	$i_3 = \log_2(36/2) = 4.170$ bits
4	$i_4 = \log_2(36/3) = 3.585$ bits
5	$i_5 = \log_2(36/4) = 3.170$ bits
6	$i_6 = \log_2(36/5) = 2.848$ bits
7	$i_7 = \log_2(36/6) = 2.585$ bits
8	$i_8 = \log_2(36/5) = 2.848$ bits
9	$i_9 = \log_2(36/4) = 3.170$ bits
10	$i_{10} = \log_2(36/3) = 3.585$ bits
11	$i_{11} = \log_2(36/2) = 4.170$ bits
12	$i_{12} = \log_2(36/1) = 5.170$ bits

The average information provided by the sum of 2 dice: **Entropy**

$$i_{ave} = \sum_{i=2}^{12} \frac{M_i}{N} \log_2\left(\frac{N}{M_i}\right) = -\sum_i p_i \log_2(p_i) = 3.274 \text{ bits}$$

Show Me the Bits!

- Is there a concrete ENCODING that achieves the information content?
- Can the sum of two dice REALLY be *represented* using 3.274 bits? If so, how?
- The fact is, the average information content is a strict **lower-bound** on how small of a representation that we can achieve.
- In practice, it is difficult to reach this bound. But, we can come very close.

Variable-Length Encoding

- Of course we can use differing numbers of "bits" to represent each item of data
- This is particularly useful if all items are *not* equally likely
- Equally likely items lead to fixed length encodings:
  - Ex: Encode a "particular" roll of 5?
  - {(1,4), (2,3), (3,2), (4,1)} which are equally likely if we use fair dice
  - Entropy = $-\sum_{i=1}^{4} p(roll_i | roll = 5)\log_2(p(roll_i | roll = 5)) = -\sum_{i=1}^{4} \frac{1}{4}\log_2(\frac{1}{4}) = 2$ bits
  - 00 – (1,4), 01 – (2,3), 10 – (3,2), 11 – (4,1)
- Back to the original problem. Let's use this encoding:

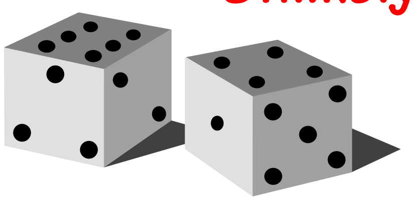
2 - 10011	3 - 0101	4 - 011	5 - 001
6 - 111	7 - 101	8 - 110	9 - 000
10 - 1000	11 - 0100	12 - 10010	

Variable-Length Encoding

- Taking a closer look

2 - 10011	3 - 0101	4 - 011	5 - 001
6 - 111	7 - 101	8 - 110	9 - 000
10 - 1000	11 - 0100	12 - 10010	

Unlikely rolls are encoded using more bits



More likely rolls use fewer bits

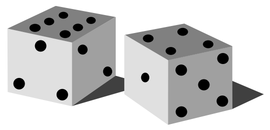
- Decoding

Example Stream:

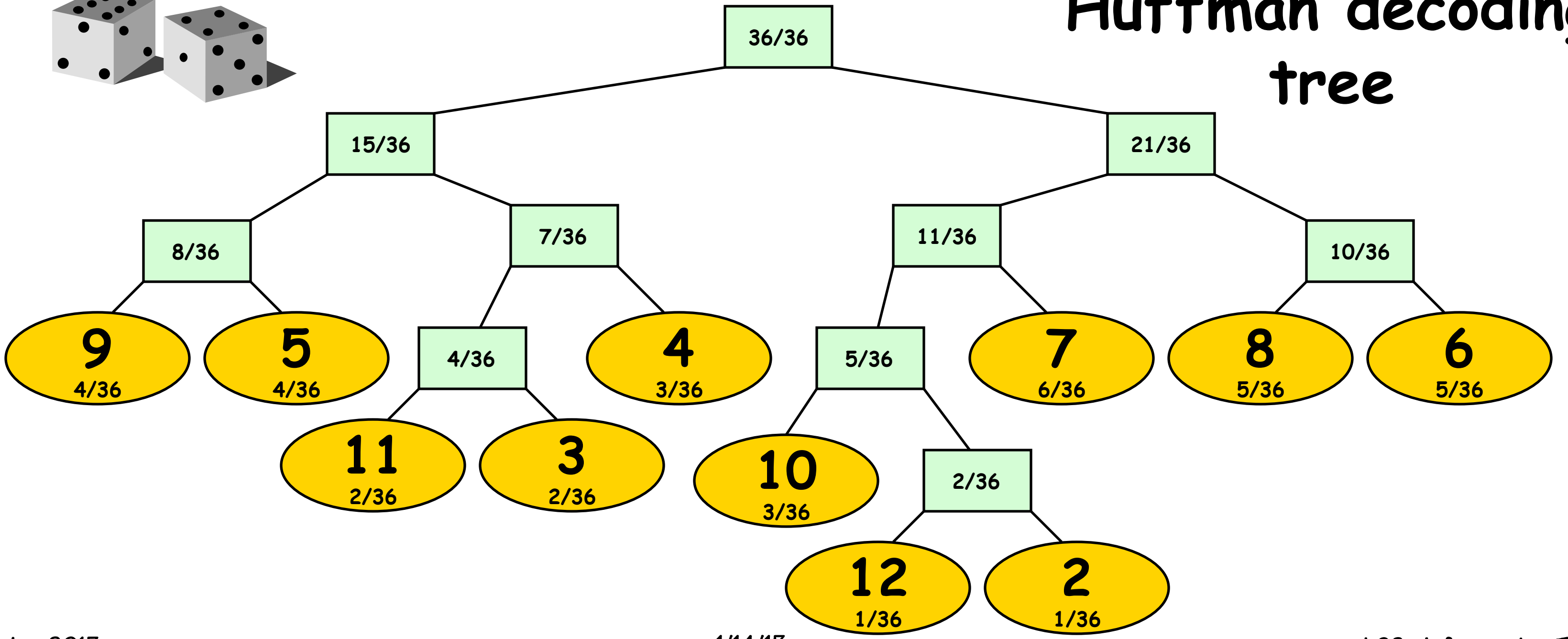
2	5	3	6	5	8	3
10011	001	0101	111	001	110	0101

Huffman Coding

- A simple *greedy* algorithm for approximating an entropy efficient encoding
  1. Find the 2 items with the smallest probabilities
  2. Join them into a new *meta* item whose probability is their sum
  3. Remove the two items and insert the new meta item
  4. Repeat from step 1 until there is only one item



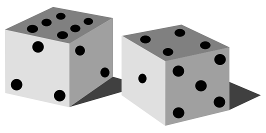
Huffman decoding tree



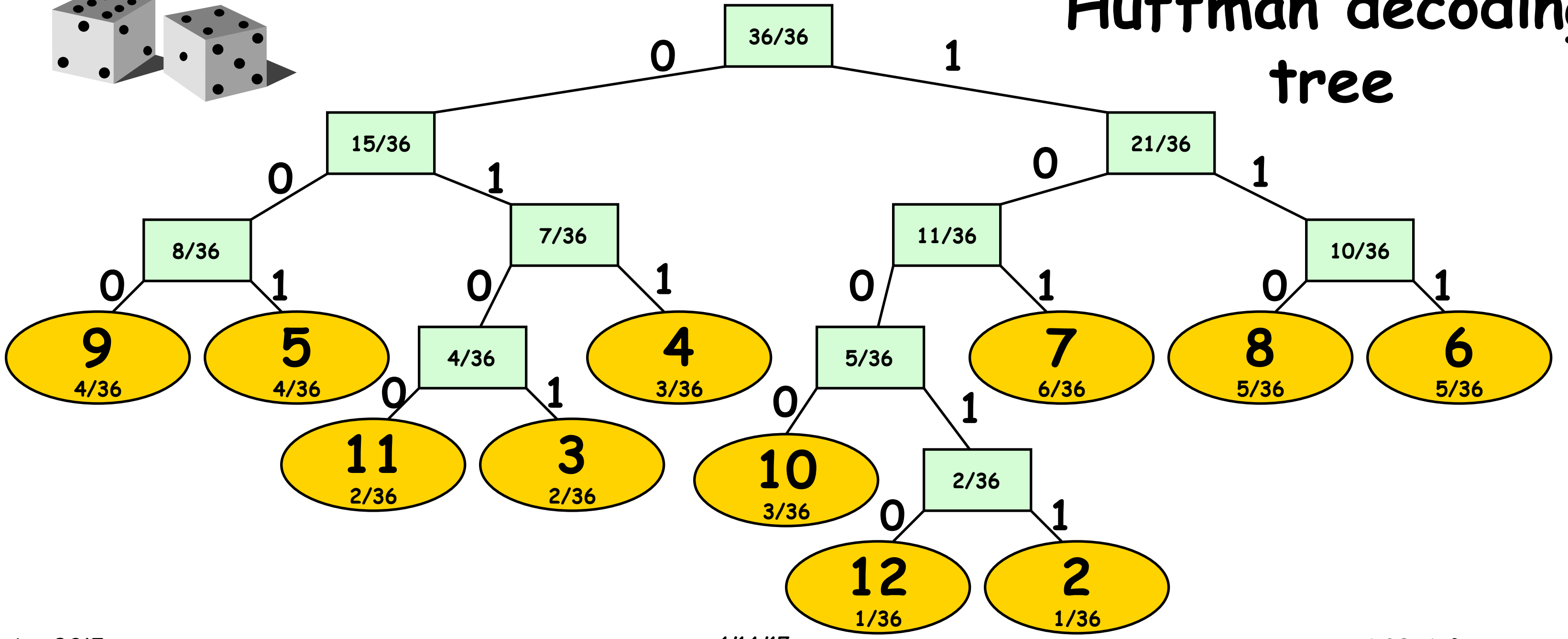
Converting Tree to Encoding

- Once the *tree* is constructed, label its edges consistently and follow the paths from the largest *meta* item to each of the real item to find the encoding.

2 - 10011	3 - 0101	4 - 011	5 - 001
6 – 111	7 – 101	8 - 110	9 - 000
10 - 1000	11 - 0100	12 - 10010	



Huffman decoding tree



Encoding Efficiency

- How does this encoding strategy compare to the information content of the roll?

$$b_{ave} = \frac{1}{36}5 + \frac{2}{36}4 + \frac{3}{36}3 + \frac{4}{36}3 + \frac{5}{36}3 + \frac{6}{36}3$$

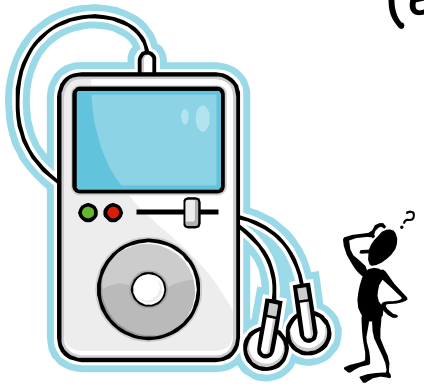
$$+ \frac{5}{36}3 + \frac{4}{36}3 + \frac{3}{36}4 + \frac{2}{36}4 + \frac{1}{36}5$$

$$b_{ave} = 3.306$$

- Pretty close. Recall that the lower bound was 3.274 bits. However, an efficient encoding (as defined by having an average code size close to the information content) is not always what we want!

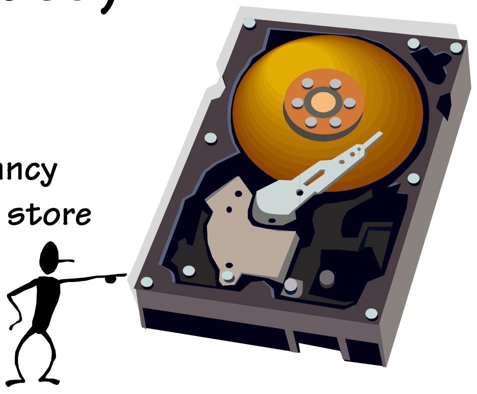
Encoding Considerations

- Encoding schemes that attempt to match the information content of a data stream remove redundancy. They are *data compression* techniques.
- However, sometimes our goal in encoding information is *increase redundancy*, rather than remove it. Why?
- Make the information easier to manipulate (fixed-sized encodings)
- Make the data stream resilient to noise (error detecting and correcting codes)



-Data compression allows us to store our entire music and video collections in a pocketable device

-Data redundancy enables us to store that *same* information *reliably* on a hard drive



Information Encoding Standards

- Encoding describes the process of ***assigning representations to information***
- Choosing an appropriate and efficient encoding is a real engineering challenge (and an art)
- Impacts design at many levels
  - Mechanism (devices, # of components used)
  - Efficiency (bits used)
  - Reliability (noise)
  - Security (encryption)

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 should use at least enough bits to represent the information content.

ex. Decimal digits 10 = {0,1,2,3,4,5,6,7,8,9}

4-bit BCD (binary code decimal)

$\log_2(10/1) = 3.322 < 4$ bits

BCD		
0	-	0000
1	-	0001
2	-	0010
3	-	0011
4	-	0100
5	-	0101
6	-	0110
7	-	0111
8	-	1000
9	-	1001

ex. ~84 English characters = {A-Z (26), a-z (26), 0-9 (10), punctuation (8), math (9), financial (5)}

7-bit ASCII (American Standard Code for Information Interchange)

$\log_2(84/1) = 6.392 < 7$ bits

ASCII

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
000	NUL	SOH	STX	ETX	EOT	ACK	ENQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
010		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
110	`	a	b	c	d	e	f	g	h	l	j	k	l	m	n	o
111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- For letters upper and lower case differ in the 6th "shift" bit
    - 10aaaaaa is upper, and 11aaaaa is lower
- Special "control" characters set upper two bits to 00
    - ex. cntl-g → bell, cntl-m → carriage return, cntl-[→ esc
- This is why bytes have 8-bits (ASCII + optional parity). Historically, there were computers built with 6-bit bytes, which required a special "shift" character to set case.

Unicode

- ASCII is biased towards western languages. English in particular.
- There are, in fact, many more than 256 characters in common use:

  â, m, ö, ñ, è, ¥, 揗, 敇, 흽, 力, א, ג, ж, š, ค
- Unicode is a worldwide standard that supports all languages, special characters, classic, and arcane
- Several encoding variants 16-bit (UTF-8)

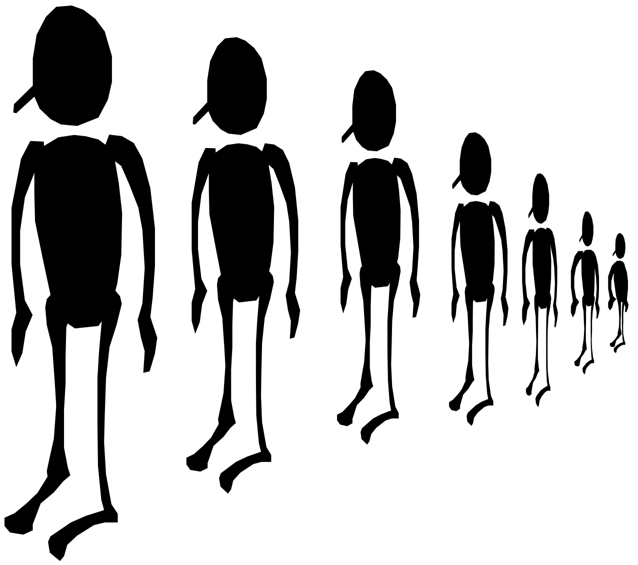
ASCII equiv range:				0xxxxxxx
Lower 11-bits of 16-bit Unicode			110yyyyx	10xxxxxx
16-bit Unicode		1110zzzz	10zyyyyx	10xxxxxx
	11110www	10wwzzzz	10zyyyyx	10xxxxxx

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



$$\begin{array}{rrr} & 2^4 = & 16 \\ + & 2^6 = & 64 \\ + & 2^7 = & 128 \\ + & 2^8 = & 256 \\ + & 2^9 = & 512 \\ + & 2^{10} = & \underline{1024} \\ & & 2000_{10} \end{array}$$

Some Bit Tricks

- You are going to have to get accustomed to working in binary. Specifically for Comp 411, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

1. Memorize the first 10 powers of 2

$2^0 = 1$	$2^5 = 32$
$2^1 = 2$	$2^6 = 64$
$2^2 = 4$	$2^7 = 128$
$2^3 = 8$	$2^8 = 256$
$2^4 = 16$	$2^9 = 512$

More Tricks with Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 411, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

2. Memorize the prefixes for powers of 2 that are multiples of 10

2^{10} = Kilo (1024)
2^{20} = Mega (1024*1024)
2^{30} = Giga (1024*1024*1024)
2^{40} = Tera (1024*1024*1024*1024)
2^{50} = Peta (1024*1024*1024 *1024*1024)
2^{60} = Exa (1024*1024*1024*1024*1024*1024)

Even More Tricks with Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 411, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

$$01\boxed{0001100000}\boxed{0001100000}\boxed{0000101000}$$

3. When you convert a binary number to decimal, first break it down into *clusters* of 10 bits.
4. Then compute the value of the leftmost remaining bits (1) find the appropriate prefix (GIGA) (Often this is sufficient)
5. Compute the value of and add in each remaining 10-bit cluster

Other Helpful Clusters

Oftentimes we will find it convenient to cluster groups of bits together for a more compact written representation. Clustering by 3 bits is called Octal, and it is often indicated with a leading zero, 0. Octal is not that common today.

$$v = \sum_{i=0}^{n-1} 8^i d_i$$

$$\begin{array}{cccccccccccc} 2^{11} & 2^{10} & 2^{9} & 2^{8} & 2^{7} & 2^{6} & 2^{5} & 2^{4} & 2^{3} & 2^{2} & 2^{1} & 2^{0} \\ \hline \end{array}$$

$$\underbrace{0\,1\,1}_{3}\;\underbrace{1\,1\,1}_{7}\;\underbrace{0\,1\,0}_{2}\;\underbrace{0\,0\,0}_{0} = 2000_{10}$$

03720

Octal - base 8

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$$\begin{array}{rcr} 0*8^0 & = & 0 \\ +\ 2*8^1 & = & 16 \\ +\ 7*8^2 & = & 448 \\ +\ 3*8^3 & = & \underline{1536} \\ & & 2000_{10} \end{array}$$

One Last Clustering

Clusters of 4 bits are used most frequently. This representation is called hexadecimal. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16. Commonly indicated with a leading "0x".

$$v = \sum_{i=0}^{n-1} 16^i d_i$$

0x7d0

$$\begin{array}{cccccccccccc} 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \underbrace{0\;1\;1\;1}_{7} & & & & \underbrace{1\;1\;0\;1}_{d} & & & & \underbrace{0\;0\;0\;0}_{0} & & & \end{array} = 2000_{10}$$

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

$$\begin{aligned} 0*16^0 &= 0 \\ +\ 13*16^1 &= 208 \\ +\ 7*16^2 &= \underline{1792} \\ & \quad 2000_{10} \end{aligned}$$

Signed-Number Representations

- There are also schemes for representing signed integers with bits. One obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.

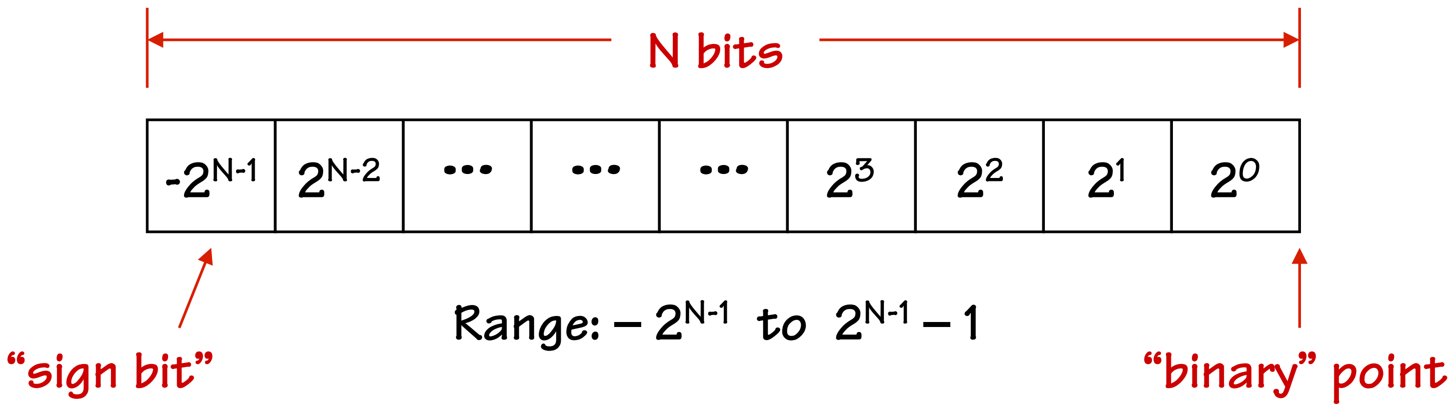
$$v = -1^{S} \sum_{i=0}^{n-2} 2^{i} b_{i}$$

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

2000 -2000

- The Good:
  - Easy to negate, find absolute value
- The Bad:
  - Add/subtract is complicated; depends on the signs
  - Two different ways of representing a 0
- Not used that frequently in practice
  - with one important exception

2's Complement Integers



The 2's complement representation for signed integers is the most commonly used signed-integer representation. It is a simple modification of unsigned integers where the most significant bit is considered negative.

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

8-bit 2's complement example:

11010110 $= -2^7 + 2^6 + 2^4 + 2^2 + 2^1$
$= -128 + 64 + 16 + 4 + 2 = -42$

Why 2's Complement?

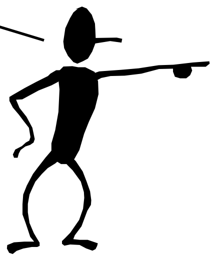
If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

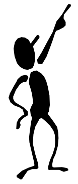
When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.

Example:

$$\begin{array}{rcl} 55_{10} & = & 00110111_2 \\ +\ 10_{10} & = & 00001010_2 \\ \hline 65_{10} & = & 01000001_2 \end{array}$$

$$\begin{array}{rcr} 55_{10} & = & 00110111_2 \\ +-10_{10} & = & 11110110_2 \\ \hline 45_{10} & = & 100101101_2 \end{array}$$





ignore this overflow

2's Complement Tricks

- Negation – changing the sign of a number
  - First invert every bit (i.e. 1 → 0, 0 → 1)
  - Add 1

  Example: 20 = 00010100, -20 = 11101011 + 1 = 11101100

- Sign-Extension – aligning different sized 2's complement integers
  - Simply copy the sign bit into higher positions
- 16-bit version of 42 = 0000 0000 0010 1010
- 8-bit version of -2 = 1 1 1 1 1 1 1 1 1 1 1 1 1 110

CLASS EXERCISE

10's-complement Arithmetic

(You'll never need to borrow again)

Step 1) Write down 2 3-digit numbers that you want to subtract

Step 2) Form the 9's-complement of each digit in the second number (the subtrahend)

Step 3) Add 1 to it (the subtrahend)

Step 4) Add this number to the first

Step 5) If your result was less than 1000,
        form the 9's complement again and add 1
        and remember your result is negative
    else
        subtract 1000

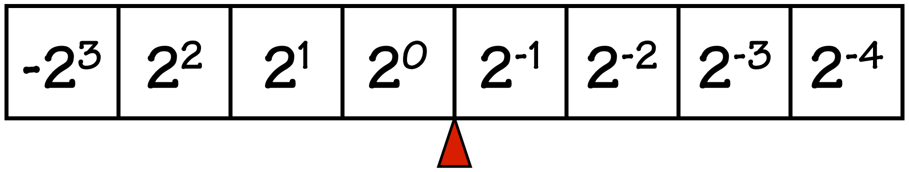
Helpful Table of the 9's complement for each digit

0 → 9	
1 → 8	
2 → 7	
3 → 6	
4 → 5	
5 → 4	
6 → 3	
7 → 2	
8 → 1	
9 → 0	

What did you get? Why weren't you taught to subtract this way?

Fixed-Point Numbers

By moving the implicit location of the “binary” point, we can represent signed fractions too. This has no effect on how operations are performed, assuming that the operands are properly aligned.



$$
\begin{aligned}
1101.0110 &= -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\
&= -8 + 4 + 1 + 0.25 + 0.125 \\
&= -2.625
\end{aligned}
$$

OR

$$1101.0110 = -42 * 2^{-4} = -42/16 = -2.625$$

Repeated Binary Fractions

Not all fractions can be represented exactly using a finite representation. You've seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent ($0.333\overline{3}\ldots$).

In Binary, a great many fractions that you've grown attached to require an infinite number of bits to represent exactly.

EX: $1 / 10 = 0.1_{10} = .00011\overline{0011}\ldots_2$

$1 / 5 = 0.2_{10} = .\overline{0011}\ldots_2 = 0.\overline{3}33\ldots_{16}$

Bias Notation

- There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called "Bias Notation".

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

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1	1	0

EX: (Bias = 127)

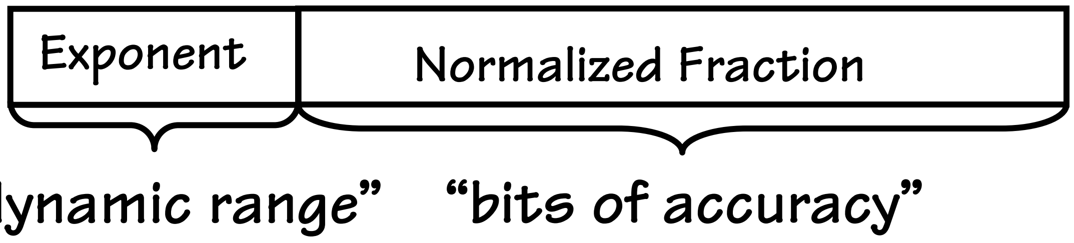
$$\begin{array}{rcr} 6 * 1 & = & 6 \\ 13 * 16 & = & 208 \\ & & -127 \\ \hline & & 87 \end{array}$$

Why? Monotonicity

Floating Point Numbers

Another way to represent numbers is to use a notation similar to Scientific Notation. This format can be used to represent numbers with fractions (3.90×10^{-4}), very small numbers (1.60×10^{-19}), and large numbers (6.02×10^{23}). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the “floating” binary point).

$$Normalized \quad Fraction \times 2^{Exponent}$$

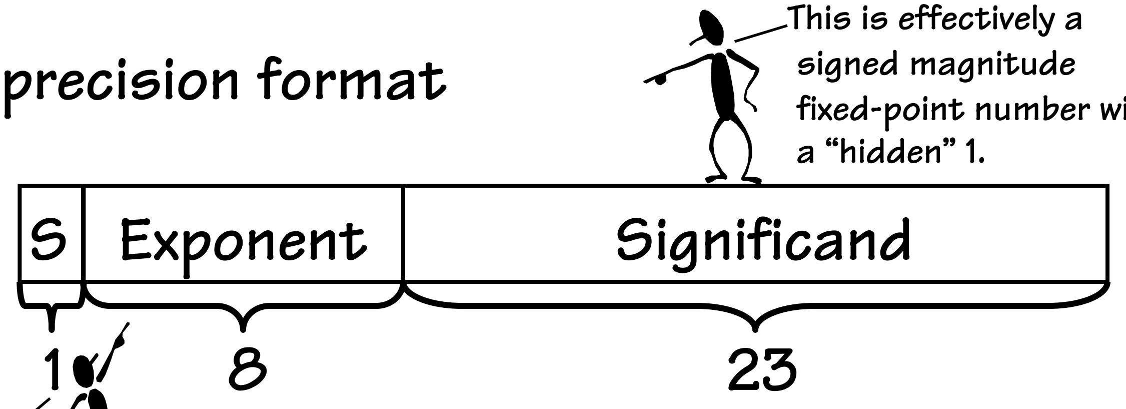


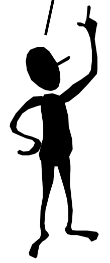
IEEE 754 Format

- Single precision format

This is effectively a signed magnitude fixed-point number with a “hidden” 1.

The 1 is hidden because it provides no information after the number is “normalized”





The exponent is represented in bias 127 notation. Why?

$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-127}$$

- Example

$42.75 = 00101010.11000000_2$

Normalize: $001.010101100000_2 \times 2^{5}$

(127+5)

$0\ 10000100\ 01010110000000000000000_2$

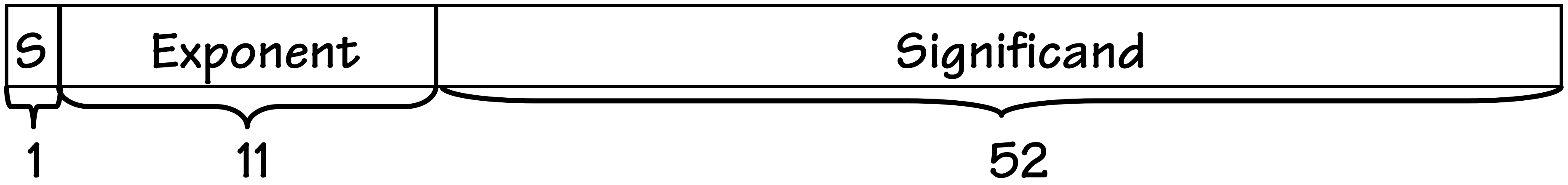
$0100\ 0010\ 0010\ 1011\ 0000\ 0000\ 0000\ 0000_2$

$42.75 = \text{0x422B0000}_{16}$

IEEE 754 Format

- Single precision limitations
  - A little more than 7 decimal digits of precision
  - minimum positive normalized value: ~1.18×10^{-38}
  - maximum positive normalized value: ~3.4×10^{38}
- Inaccuracies become evident after multiple single precision operations

- Double precision format



$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-1023}$$

Summary

Information resolves uncertainty

- Choices equally probable:
  - N choices narrowed down to M $\rightarrow$
    $\log_2(N/M)$ bits of information
- Choices not equally probable:
  - choice_i with probability $p_i \rightarrow$
    $\log_2(1/p_i)$ bits of information
  - average number of bits = $\Sigma p_i \log_2(1/p_i)$
  - variable-length encodings

Next time:

- How to encode thing we care about using bits, such as numbers, characters, etc…
- Bit's cousins, bytes, nibbles, and words

Summary (continued)

1) Selecting the encoding of information has important implications on how this information can be processed, and how much space it requires.
2) Computer arithmetic is constrained by finite representations, this has advantages (it allows for complement arithmetic) and disadvantages (it allows for overflows, numbers too big or small to be represented).
3) Bit patterns can be interpreted in an endless number of ways, however important standards do exist
   - Two's complement
   - IEEE 754 floating point