EECS-317 Data Management and Information Processing

Lecture 15 – Number representations

Steve Tarzia

Spring 2019

Northwestern

Last Lecture: Web Scraping & Messy Data

- Data can be **scraped** from web pages by writing code that:
  - Downloads HTML pages
  - Picks out data elements using **CSS selectors** (or XPath)
  - Also pick out links to pages with additional data
  - Repeat!
- Data can have missing, incorrect, or inconsistent values for many reasons:
  - Pulled from different sources with different naming or unit conventions
  - Paper scanning (OCR) errors
  - Human input errors
- Variety of tools are needed to deal with messy data:
  - Review summary statistics
  - Synonym tables
  - Named entity matching with ML (dedupe.io and Open Refine)
  - Crowdsourcing: MTurk, home-grown solutions
- Above all, don't blindly trust data you are given!

Computers store information in **binary**

- Ones and Zeros
- …00010010000010010011100110110101010111100000…
- Called "bits," meaning "**b**inary dig**its**"

- Why?
  - Simplicity
  - Noise robustness
  - By convention

- But how do we get meaning from a sequence of ones and zeros?

Data is zeros and ones plus an interpretation/context

- An encoding defines what the zeros and ones represent
- "01000100011000010111010001100001" can represent:
  - The number 1,147,237,473 as an integer
  - The number 901.8184 as a float
  - The four letters "Data" in the ASCII or UTF-8 character encoding
  - This color (at 37% transparency) in RGBA
  - 32 separate True or False values
- Any crazy encoding is possible, but there are some standards.

Decimal numbers in text

- CSV, JSON, and XML files store **text**, usually UTF-8 encoded.
- In that text, you can print decimal numbers using the chars [0-9.eE\-]
- For example:
  - “12” = “1” + “2” = 0x 31 32 = 0011 0001 0011 0010
  - “12.2e-4” = “1” + “2” + “.” + “2” + “e” + “-” + “4”
    = 0x 31 32 2E 32 65 2D 34
    = 0011 0001 0011 0010 0010 1110 0011 0010 0110 0101 0010 1101 0011 0100
- These text-based encodings are **inefficient** because they only make use of a small subset of the characters.
- However, they are easy to read and machine-independent.
- A general-purpose compressor like “gzip” works well on text.
- Other numeric encodings work directly with the bits, not with text.

ASCII TABLE

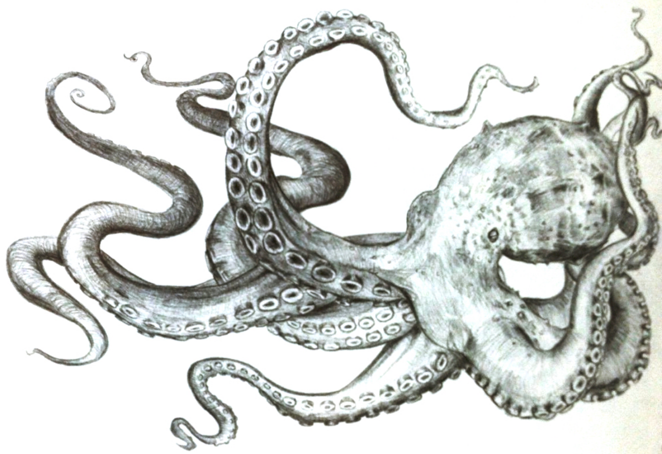
Subset of chars used by numbers are highlighted.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Integers

- Integers are the simplest of all data encodings
- Whole numbers only (no fractions)
- Numbers are represented directly in the "base two" positional notation
- The familiar "base ten" representation of numbers is just a convention due to the fact that humans have ten fingers.
- What number base will octopuses evolve to use?

(drawing from http://drawingpencilarts.com/realistic-octopus-drawing/)



Integers in detail

Decimal 137_{ten}

1	3	7	
x100	x10	x1	← powers of 10

100 + 30 + 7 = 137

Binary $10001001_{two} = 137_{ten}$

1	0	0	0	1	0	0	1	
x128	x64	x32	x16	x8	x4	x2	x1	← powers of 2

128 + 0 + 0 + 0 + 8 + 0 + 0 + 1 = 137

Simple binary integers

$1_{ten} = 1_{two}$

$2_{ten} = 10_{two}$

$4_{ten} = 100_{two}$

$8_{ten} = 1000_{two}$

$16_{ten} = 10000_{two}$

$32_{ten} = 100000_{two}$

$64_{ten} = 1000000_{two}$

$128_{ten} = 10000000_{two}$

$3_{ten} = 11_{two}$

$7_{ten} = 111_{two}$

$15_{ten} = 1111_{two}$

$31_{ten} = 11111_{two}$

$63_{ten} = 111111_{two}$

$127_{ten} = 1111111_{two}$

$255_{ten} = 11111111_{two}$

There are only 10 types of people in this world… those who understand binary and those who don’t.

(Stop and practice)

Binary tricks

- Remember the first eight powers of two:
  - 2, 4, 8, 16, 32, 64, 128, 256
- Remember that $\mathbf{2^{10} = 1024} \approx 1000$
  - Lets you estimate the number of binary digits in a decimal integer: Every three decimal digits gives about ten binary digits
- Remember the important large powers of two:
  - $2^8 = 256$
  - $2^{16} \approx$ 64 thousand
  - **$2^{32} \approx$ 4 billion**
  - $2^{64} \approx$ really big

Addition in binary

$4 + 7 = 11$

$$
\begin{array}{rr}
1 & \leftarrow \text{carry} \\
 & 4 \\
+ & 7 \\
\hline
1 & 1
\end{array}
$$

$100 + 111 = 1011$

$$
\begin{array}{rrrr}
1 & \leftarrow & \text{carry} & \\
 & 1 & 0 & 0 \\
+ & 1 & 1 & 1 \\
\hline
1 & 0 & 1 & 1
\end{array}
$$

More binary addition

63 + 98 = 161

11111 + 110010 = 1010001

```
1 1 ← carry
  6 3
+ 9 8
-----
1 6 1
```

```
1 1 1 1 1 ← carry
    1 1 1 1 1
+ 1 1 0 0 1 0
-------------
1 0 1 0 0 0 1
```

Subtraction: addition's tricky pal

161 – 98 = 63

↝ ↝ borrow

0~~1~~ 15~~6~~ 11

\- 9 8

6 3

1010001 - 110010 = 11111

↝ ↝ ↝ ↝ ↝ borrow

0~~1~~ 1~~0~~ 10~~1~~ 1~~0~~ 1~~0~~ 10 1

\- 1 1 0 0 1 0

1 1 1 1 1

What about negative integers?

- Signed integers can represent both positive and negative integers
- We need an extra bit to represent the sign of the number
- But we don't just use a simple sign bit
- We use two's complement to represent negative numbers, because it
  - Simplifies the computer's addition and subtraction circuitry, and
  - And it has just one representation of zero
- Negative numbers "roll over" from the top of the binary range.

Works like an old-style car odometer

Two's complement for three-bit numbers

```
 3: 011
 2: 010
 1: 001
 0: 000
-1: 111
-2: 110
-3: 101
-4: 100
```

↩ rollover

-2 + 1 = -1

`110 + 001 = 111`

- Subtraction is done in the exact same way as addition!
- No need to learn how to "borrow."

Subtraction works just like addition!

No need to learn how to "borrow."

Just negate the second number and add.

$3 - 2 = 3 + (-2) =$

```
1 1   ← carry
  0 1 1
+ 1 1 0
-------
  0 0 1   ← our answer!
```

We ignore the final carry because it falls outside of the 3-bits we are working with. That's how we roll-over between negative and positive.

```
 3: 011
 2: 010
 1: 001
 0: 000
-1: 111
-2: 110
-3: 101
-4: 100
```

Two's complement negation

To negate a number:

- Flip all the bits. Ones become zeros and zeros become ones.
- Add one

For example -3

- Start with the bits for three: **`011`**
- Flip the bits: **`100`**
- Add one: **`101`**

```
 3: 011
 2: 010
 1: 001
 0: 000
-1: 111
-2: 110
-3: 101
-4: 100
```

Overflow: when numbers don't fit

For example, 2 + 2 = 4

4 cannot be represented in a three-bit signed integer.
What happens when we try this addition?

```
    1 ← carry
    0 1 0
+   0 1 0
---------
    1 0 0  ← answer looks like -4!
```

3:	011
2:	010
1:	001
0:	000
-1:	111
-2:	110
-3:	101
-4:	100

- The computer will throw an exception if the signs of the operands were the same, but the sign of the result is different.
  - positive + negative cannot overflow.
  - positive + positive should give a positive
  - negative + negative should give a negative
- Remember that the left-most bit indicates the sign.

Examples with 4 and 8 bits

4-bit is between -8 and 7

8-bit is between -128 and 127

(Stop and practice)

Just for fun over the weekend

- This video shows how addition is actually implemented in hardware: https://www.youtube.com/watch?v=1I5ZMmrOfnA
  Search YouTube for "PBS ALU"

- If you're interested in learning more, take COMP_ENG-203 Intro to Computer Engineering

A few more things about integers

- Multiplication: two's complement works magically here too
- Positive division works as expected
- *Sign extension*: when increasing the "bit size" of a negative number, add leading ones.
  - Eg., -2 is **`1110`** as a 4-bit signed integer and **`11111110`** in 8 bits.
- Computers typically use 32 or 64 bit integers.

Limitations of Integers

Integers are great for **counting**, but sometimes we need to **measure** fractional quantities.

Binary numbers can have "decimal" places, too

- $\mathtt{0.1111111111}_{two}$ is slightly smaller than 1
- $\mathtt{0.0000000001}_{two}$ is slightly larger than 0
- $\mathtt{0.1}_{two}$ is one half

- $\mathtt{10.101}_{two} = \mathbf{1} \times 2^{1} + \mathbf{0} \times 2^{0} + \mathbf{1} \times 2^{-1} + \mathbf{0} \times 2^{-2} + \mathbf{1} \times 2^{-3}$
  $= 2 + 0 + 1/2 + 0 + 1/8 = 2\,\frac{5}{8}$

How shall we represent fractional number in the computer?

Fixed point: *Integers 2.0*

- Simplest solution is to just stick an implicit **radix point** somewhere.
  - We don't call it a *decimal point* because we're not in base ten.
- Examples of fixed point numbers in base ten:
  - Represent the cost of a purchase with an integer number of cents.
    - The cost of a sandwich is 625 cents.
  - Represent the distance between cities by counting the hundredths of a mile.
    - Evanston is 1321 hundredths of a mile from Chicago
    - and 79,543 hundredths of a mile from Philadelphia

Fixed point example in 16 bits

Let's store the chemical elements' atomic weights.

- Smallest value (hydrogen) is 1.00784
- Largest value (uranium) is 238.02891
- Negative values are not possible
- We can reserve 8 bits for the fractional part and 8 bits for the part > 1
- In this particular binary fixed point representation, weight of uranium is:

  The radix point is implicit, not stored in the computer.

  `11101110.00000111`

  $= 238\,\frac{7}{256} = 238.02734375$ (We had to round off, so this is not precisely accurate)

- And the weight of hydrogen is:

  `00000001.00000010`

  $= 1\,\frac{2}{256} = 1.0078125$

Fixed point limitations

- Fixed point is simple & efficient, but…
- Range is very limited
  - Multiplication overflows easily – can double the number of bits
    - Eg., if working in 32-bits, then we can only multiply 16-bit values without overflow
  - Division **underflows** easily (small values are rounded to zero)
- Precision varies across the range:
  - Small numbers have few significant figures:
  - For example, `00000000.00000010` is not very precise

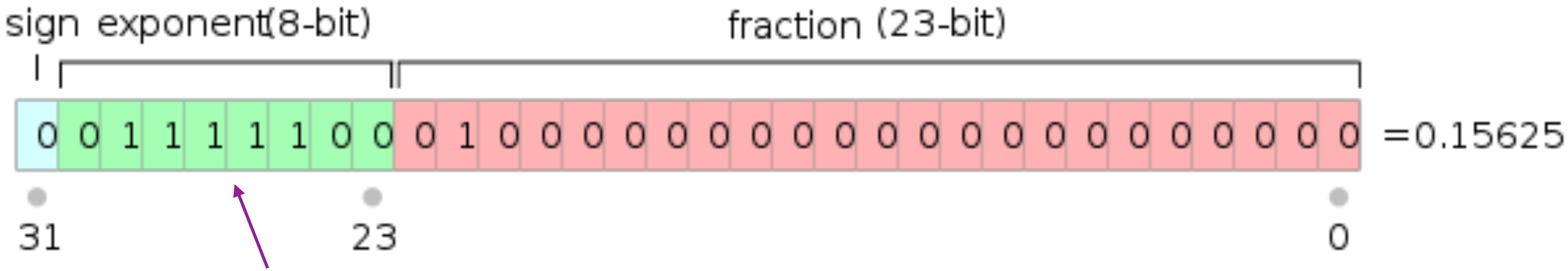
Floating point

- Based on **scientific notation**:
  - $10{,}340 = 1.034 \times 10^4$
  - $0.00424 = 4.24 \times 10^{-3}$
- Scientific notation gives a compact representation of extreme values:
  - $1{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000 = 1.0 \times 10^{24}$
  - $0.000\ 000\ 000\ 000\ 000\ 000\ 000\ 001 = 1.0 \times 10^{-24}$
- In binary:
  - $100010_{two} = 1.0001_{two} \times 2^5{}_{ten} = 1.0001 \times 10^{101}{}_{two}$
  - $0.00101_{two} = 1.01_{two} \times 2^{-3}{}_{ten} = 1.01 \times 10^{-11}{}_{two}$

Representing floating point in bits

$$0.15625_{\text{ten}} = 0.00101_{\text{two}} = 1.\underline{01} \times 10^{\underline{-11}}{}_{\text{two}}$$

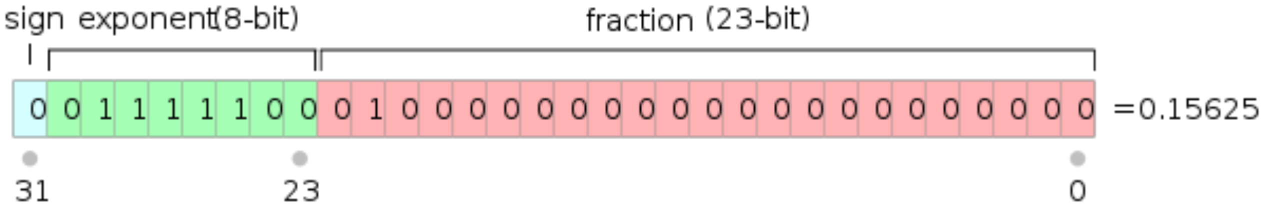
- Three essential parts are the **sign**, **fraction**, & **exponent**
  - Notice that the first significant figure is always "1" so we don't have to store it
- In the mid 1980s, the IEEE standardized the floating point representation of 32 and 64 bit numbers:
  - The exponent has a sign too, but the standard says to add a "bias" of 127

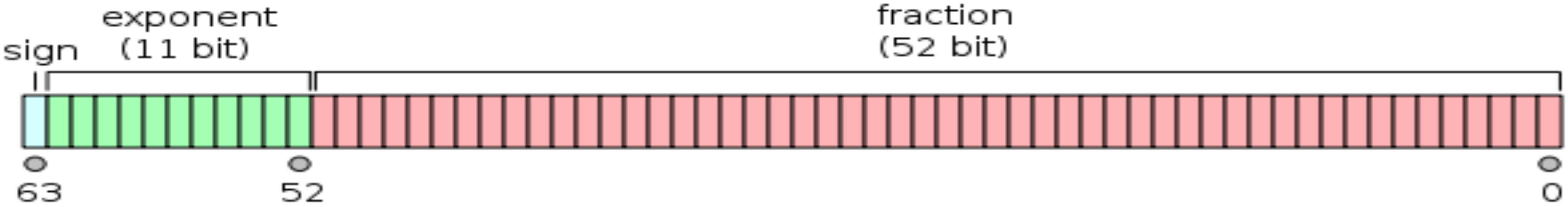


1111100 = 124   124-127 = -3 exponent

64-bit floating point

- Similar to 32-bit, but we have more precision in the fraction and larger exponents are possible.
- 32-bit is called **single precision** and 64-bit is called **double precision**.
- Double precision can represent larger, smaller, and more precise numbers.





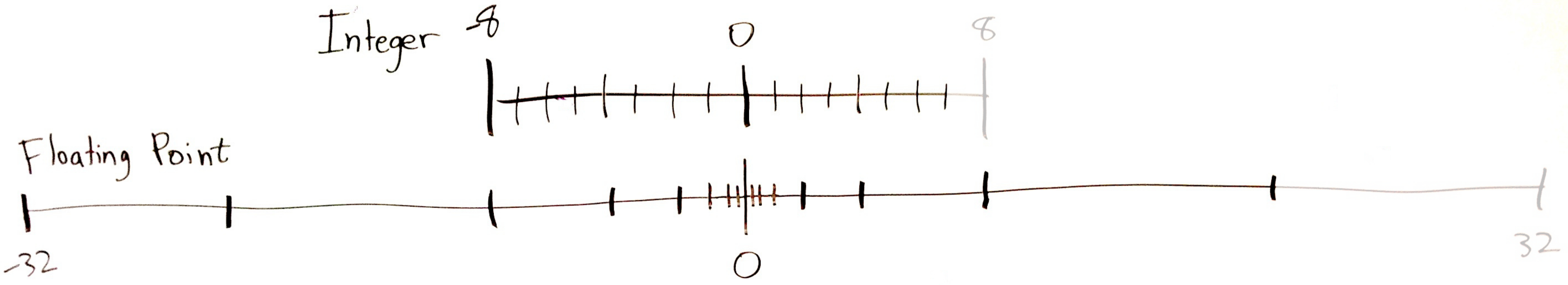
A few special floats

- The IEEE standard allows for a few special values to be stored
  - Positive and negative zero (We normally start with an implied "1" which doesn't work for zero)
  - Positive and negative infinity (the result of divide by zero)
  - Not a number (the result of zero divided by zero)
- These all have the exponent bits set to all ones or all zeros

The Flexibility and Flaws of Floats

- A 32-bit signed integer can represent all the whole numbers between -2,147,483,648 and 2,147,483,647
- A 32-bit floating point number can be as large as $\pm 3.402823 \times 10^{38}$ $= 340{,}282{,}300{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$
- or as tiny as $5.8774718 \times 10^{-39}$ $= 0.000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 005\ 877\ 471\ 8$
- But, single-precision floats have only 24 bits of precision:
  - Can only precisely store **integers** up to 2^{24} = **16,777,216**
- Floats can store larger numbers than integers of the same bit-length, but with less precision because 8 bits are set aside for the exponent.

Floats just distribute numbers differently



- Above, the dashes represent possible numbers using 4 bits.
- Both of the above number lines have 16 dashes (possible numbers)
  - Actually, there are 17 dashes, and we have to leave out the largest number (8, 32).
- The only difference is the spacing.
  - Integer spacing is constant but floats are ***exponentially spaced***

Catastrophic Cancellation

- Subtraction of similar-sized numbers leads to a **loss of precision:**

  $$0.1234567891 - 0.1234567890 = 0.0000000001$$

  $$1.234567891 \times 10^{-1} - 1.234567890 \times 10^{-1} = 1.\underbrace{000000000}_{\textit{result has 9 insignificant figures}} \times 10^{-10}$$

- We started with 10 significant figures but the result has just one sig fig!
  - Note that I'm giving an example in decimal, but the same idea applies to floating point's binary representation.
- What about:
  - addition? multiplication? division?
    - Actually, only subtraction can lead to a loss of precision.
  - Integers?
    - Integers may *overflow*, precision is not really defined for integers.

Numerical Methods

- Math on computers (especially with floats) has limited precision.
- The field of **Numerical Methods** (within Applied Math) studies:
  - The **errors** introduced by numeric representations and calculations,
  - Optimizes numerical calculations so as to minimize errors, runtime, etc.

- For example, the quadratic formula you learned in high school is theoretically correct:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- But catastrophic cancellation occurs when $b \cong \sqrt{b^2 - 4ac}$
- A better numerical method for finding roots of quadratic functions is as follows, though there is still a catastrophic cancellation when $4ac \cong b^2$:

$$x_1 = \frac{-b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{2c}{-b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}} = \frac{c}{ax_1}$$

When to use the various number representations

- When **counting** or labelling things, always use integers
- When **measuring** things, usually use floating point
  - May use fixed point if speed/simplicity is more important than accuracy
- If your machine does not support floating point (eg., a toaster):
  - Use fixed point representation for fractional quantities
- If rounding is desired then use fixed point
  - U.S. currency values usually should be rounded to the nearest cent
- Use 64-bit integers when you need values > 2 billion
- Floating point rules of thumb:
  - Single precision gives ~7 decimal digits of precision, max of $\sim 10^{38}$
  - Double precision gives ~16 decimal digits of precision, max of $\sim 10^{308}$

How do computers work with floats?

- It’s complicated and slow!
- Have to manipulate both the fraction and the exponent.
- Addition is no longer simple, as it was for integers & fixed point.

Recap

- Computers represent numbers with different binary encodings
- **Text** can represent decimal numbers in various formats (eg., CSV, JSON).
- **Integers** represent whole numbers
  - Remember that $\mathbf{2^{10} = 1024} \approx 1000$, $\mathbf{2^{32} \approx 4}$ **billion**
  - Signed integers use two's complement
  - Used for *counting* and *identifying* records.
- **Fixed point** adds an implicit radix point to an integer.
  - Allows representing fractional quantities as integers, but with limited range.
  - Used for numbers that *should round off*, like prices.
- **Floating point** is a binary scientific notation representation
  - Can represent tiny fractional values and huge values with equal precision
    - Single precision ≈ 7 decimal digits, Double precision ≈ 16 decimal digits of precision
  - Used for *measurements* and *calculations*.
  - Float subtraction can lead to *catastrophic cancellation*.