Floating Point

Agenda

- Fixed point representations
- Big and Small Numbers
- Scientific Notation
- IEEE 754 floating point standard
  - Special symbols
  - Underflow overflow
- Floating point addition and multiplication
- Material from section 3.5 of textbook

How to Represent Real Numbers?

Real Numbers

- Positional notation allows for fractions

$$\mathbf{a}_n\mathbf{a}_{n-1}\ldots\ldots\ldots\mathbf{a}_1\mathbf{a}_0 \,.\, \mathbf{a}_{-1}\mathbf{a}_{-2}\ldots\ldots\ldots\mathbf{a}_{-m}$$

- Let's start with fixed point representation
  - Choose n and m
  - Radix point is always in the same position
- Ex: 000.000 can’t store 0.0002
  - Easy to implement
  - Limited range
- In floating point notation the size of part n and part m can change, however the total length of part_n + part_m is fixed.

Real Numbers

- 152.3_{10}

$152.3 = 1x10^{(2)} + 5x10^{(1)} + 2x10^{(0)} + 3x10^{(-1)}$

- 1011.01_2

$1011.01_2 = 1x2^{(3)} + 0x2^{(2)} + 1x2^{(1)} + 1x2^{(0)} + 0x2^{(-1)} + 1x2^{(-2)}$

$= 1x8 + 0x4 + 1x2 + 1x1 + 0x\ 1/2 + 1x\frac{1}{4}$

$= 11.25_{10}$

Binary to Decimal

- Integers scaled by an appropriate factor
- Direct expansion with positional weights

0.11001_2

$0.11001_2 = 1 * 2^{(-1)} + 1 * 2^{(-2)} + 1 * 2^{(-5)} = ½ + ¼ + 1/32$

$0.11001_2 = 1 * 2^{(-1)} + 1 * 2^{(-2)} + 1 * 2^{(-5)} = (1*2^4 + 1*2^3 + 1*2^0)\ 2^{(-5)}$

Binary to Hexadecimal

- Use the same trick as before

0.110101001_2

$$0.110101001_2 = 0.\underbrace{1101}\underbrace{0100}\underbrace{1000}{}_2 = 0.D48_{16}$$

$0.2BE_{16}$

$$0.2BE_{16} = 0.\underbrace{0010}\underbrace{1011}\underbrace{1110}{}_2$$

Decimal to Binary

- Multiply by 2 and note the integer part
- Subtract integer part and repeat until no fraction left

0.625_{10}

0.625 * 2 = 1.250 --> Keep integer 1
0.250 * 2 = 0.5 --> Keep integer 0
0.500 * 2 = 1.0 --> Keep integer 1

$0.625_{10} = 0.101_2$

Decimal to Binary

- Can all decimal fractions be expressed exactly in Binary?

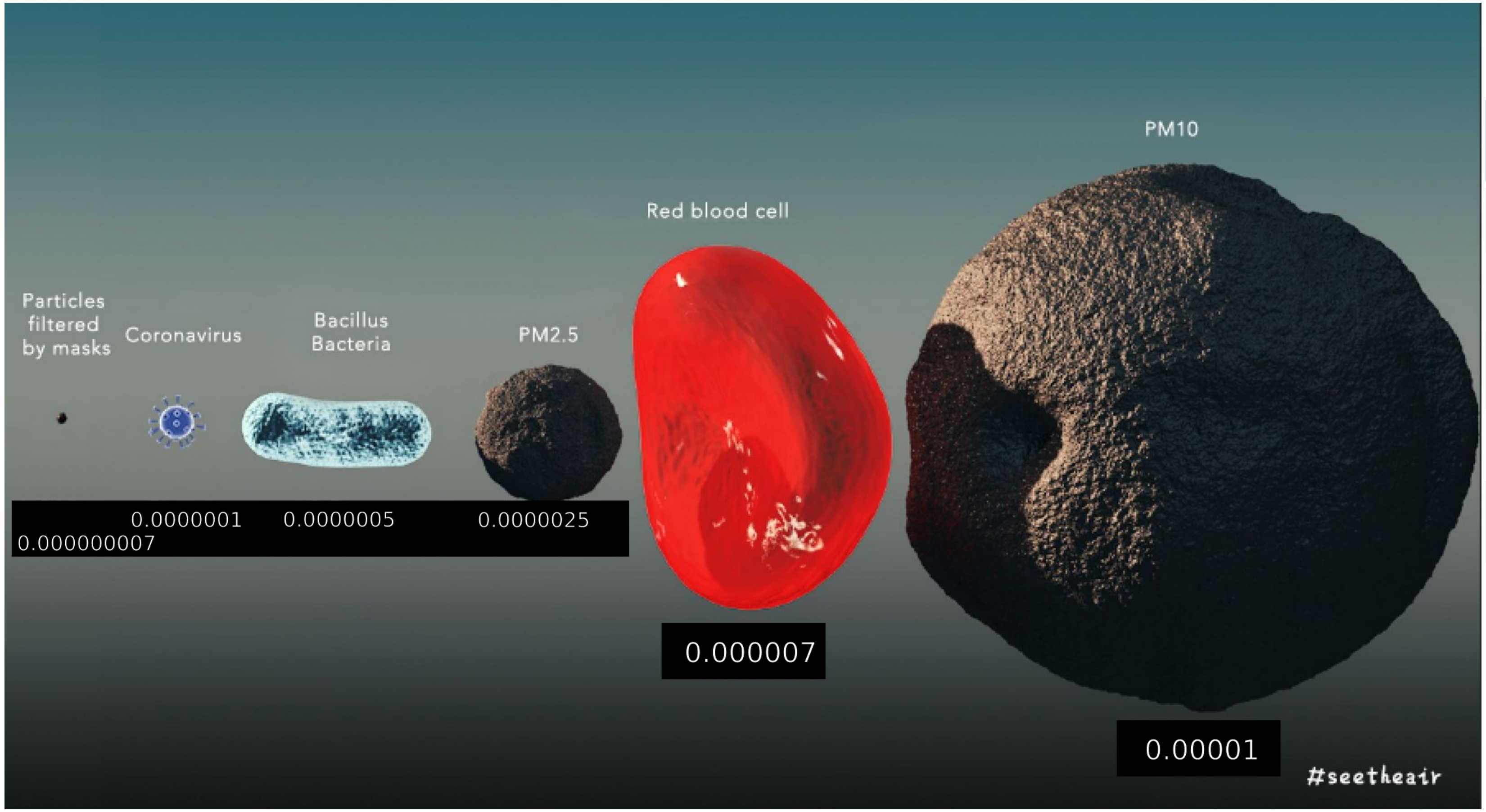
0.1_{10}

$0.1_{10} = 0.0\underline{0011}\,\underline{0011}\,\underline{0011}_2$

- 0.1 doesn’t have an exact binary representation, Just like 1/3 doesn’t have an exact decimal fraction.

How to Represent Small and Big Numbers in Decimal?

How big is Coronavirus?



Particle	Size (meter)
PM10	0.00001
Red Blood Cell	0.000007
PM2.5	0.0000025
Bacteria	0.0000005
Coronavirus	0.0000001
Particles filtered by masks	0.000000007

What numbers do we need?

3.141592…	π
2.71828…	e
1.0×10^{-9}	Seconds per nanosecond
3.15576×10^{9}	Seconds per century
1.47×10^{13}	US National Debt
$2.99792458 \times 10^{10}$	Speed of light in cm/s
6.67300×10^{-11}	Gravitational constant
1.98892×10^{30}	Mass of sun in kilograms
2.08×10^{22}	Distance to Andromeda in m
1.0×10^{-15}	Size of a proton in meters

Scientific Notation for Decimal

- We use **scientific notation** for big and small numbers
  - Use a single digit to the left of the decimal point
  - Multiplied by base (e.g., 10) raised to some exponent
  - Use e or E to denote the exponent part

  1.0×10^{-15} 1.0e-15 1.0E-15

- A ***normalized number*** has no leading zero
  - $1.0_{10} \times 10^{-9}$ normalized
  - $0.1_{10} \times 10^{-8}$ not normalized
  - $10.0_{10} \times 10^{-10}$ not normalized

Scientific Notation for Binary

- How do we represent very small and big numbers in Binary?
- Binary numbers can be written in scientific notation too
  - $1.0_2 \times 2^{-1}$

    $1.0_2 = 1.0_{10} * 2\text{^}(-1) = 0.5$

  - $1.1_2 \times 2^3$

    $1.1_2 * 2\text{^}3 = 1.5_{10} * 8 = 12$

How to Represent Floating Points?

Floating Point

- The binary point is not fixed, but instead can move based on the exponent

Normalized Binary number always has the form:

$$\mathbf{1.xxxxxxx_2 \times 2^{yyyy}}$$

  - x is the ***fraction / significand / coefficient / mantissa***
  - y is the ***exponent***
  - always has a one to the left of the binary point

Floating Point Standards

- Many options for representing floating point
  - Number of bits for the fraction
  - Number of bits for the exponent
  - How to represent zero?
  - How to represent negative numbers?
- Standards are important for exchanging data

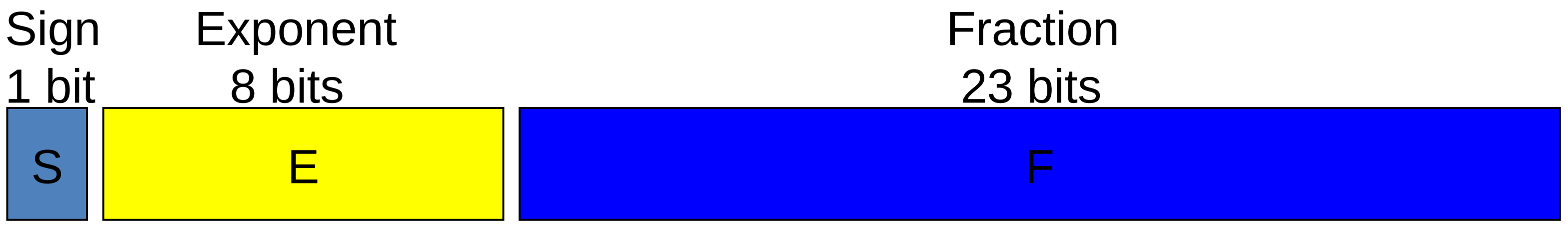
Floating Point Standards

- IEEE 754 used in nearly all computers today
  - Defines two representations
    - single precision (32 bits)
    - double precision (64 bits)

In high level languages, data of this type is called

- *float (single precision)*
- *double* (for double precision)

Single Precision



A real number can be described as

$$(-1)^S \times (1+F) \times 2^E$$

- IEEE 754 does **not** use 2's complement
- Clarification:
  - Fraction refers to the 23-bit number F
  - Mantissa refers to the 24-bit number 1+F

Single Precision

Sign 1 bit	Exponent 8 bits	Fraction 23 bits
S	E	F

A real number can be described as $(-1)^S \times (1+F) \times 2^E$

- Numbers are in normalized form. Why?

Base 2	S	Exponent	Mantissa
0.0011×2^{0}	0	0000000	001100...
0.011×2^{-1}	0	1111111	011000...
0.11×2^{-2}	0	1111110	110000...
All equivalent to the same real number. The encoding is wasteful			

Not normalized

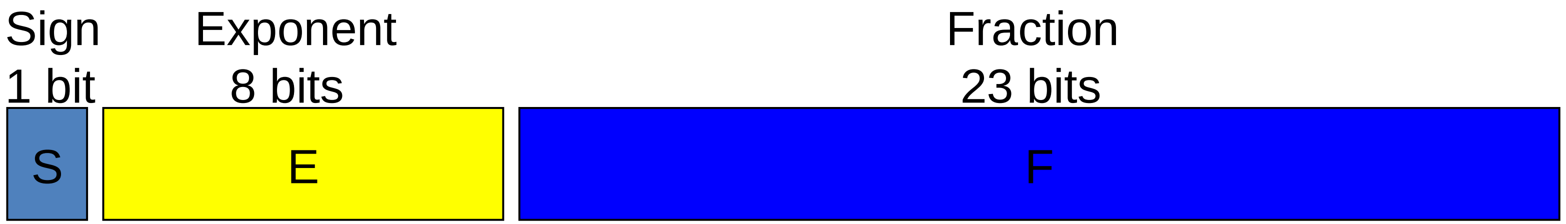
Biased Notation

- In IEEE 754, actual representation is

$$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(E - \text{Bias})}$$

  - Exponent(actual one) = E - bias
  - E = exponent (actual one) + bias
- In single-precision, bias = 127
- Represent negative exponents
- Want easy integer style comparison / sorting

Single Precision Floating Point



$$(-1)^S \times (1+F) \times 2^E$$

- IEEE 754 does not use 2’s complement
- The first bit is the sign:
  - S=0: positive number
  - S=1: negative number
- How many numbers can we represent?

Single Precision Floating Point

- Convert -0.75 from decimal to single precision

1. convert 0.75_{10} to binary

```
0.75        0.5
X   2       X 2
-----       -----
1.50        1.0
```

$0.75_{10} = 0.11_2$

2. normalize

$0.11_2 = 1.1 \times 2\text{^}(-1)$

3. sign = 1

Exponent = -1 + 127 = 126_{10} = $0111\ 1110_2$

Fraction 100000.....0

$$(-1)^S \times (1+\text{significand}) \times 2^{(E\text{-bias})}$$

$$(-1)^1 \times (1+ .10000000000000000000000) \times 2^{(126-127)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit | 8 bits | 23 bits

Single Precision Floating Point

- Convert 110000001 01000…0000 from single precision to decimal:

Sign:
S = 1

Exponent:
$E = 10000001_2 = 129_{10}$.
129-127 =2

Mantissa:
$F = 01000…0000_2 = 0.25_{10}$

(-1)^s * 1.25 * 2^2 = -5

What can go wrong?

Overflow / Underflow

- Largest number that can be represented in single precision:

  1.111 ...1 ~= 2

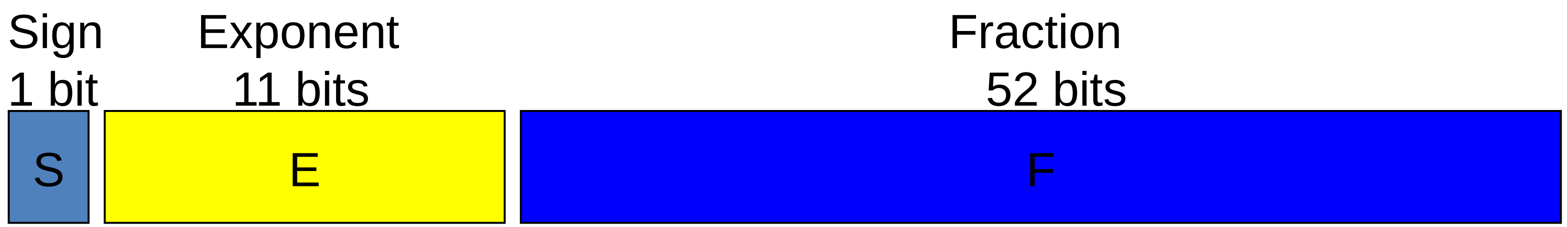
  255(max. exponent value)-127= 128

  Approximately $\pm 2.0 \times 2^{128} = 2.0 \times 10^{38}$
- Smallest fraction that can be represented in single precision:

  Approximately $\pm 2.0 \times 2^{-128} = 2.0 \times 10^{-38}$
- Overflow: representing a number larger than the one above;
- Underflow: representing a number smaller than the one above

These are approximate values as we've not yet talked about how we store special values.

Double Precision



- More bits!
- More precision
- Double precision uses a bias of 1023
- Can do more before underflow / overflow
  - Approximately 1E-308 to 1E308

Double Precision

- Convert 3.25 from decimal to double precision

1. convert decimal to binary
   3 -> 11
.25 -> .01
$3.25_{10} = 11.01_2$

2. normalize
11.01 x 2^0 = 1.101 x 2^1

3. extract the values
Sign: 0
Exponent: 1+1023 = 1024_{10} = 10000000000_2
Fraction: 101000000 …….00 (52 bits)

Floating Point Arithmetic

Floating Point Addition

- **Align** the radix points
  - Make the smaller number to match the larger
- **Add** the significands
- **Normalize** the result
  - What if one number is positive and the other negative?
  - May need to shift a lot!
  - Check for overflow or underflow when shifting!
- **Round** so number fits in available digits/bits
  - If bad luck when rounding, renormalize

Floating Point Addition

9.999e1 + 1.610e-1 with 4 digits precision

1. Align the decimal points
1.610e-1 = 0.0161 e1

2. Add the significant
 0.0161 e1
 9.999   e1
==========
10.0151 e1

3. Normalize
 1.00151e2

4. Round
If digit to right is 0 through 4, truncate
If digit to right is 5 through 9, then add 1
So 1.002e2 is our answer

Addition Example

Example Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 4.44.

Answer Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$
\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^1{}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^0 &&= 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^4{}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^0 &&= -1.110_{two} \times 2^{-2}
\end{aligned}
$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

1.001 (2's comp. of -0.111)

Step 2. Add the significands:

$$1.0_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{two} \times 2^{-1} &= 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} \\ &= 1.000_{two} \times 2^{-4} \end{aligned}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{two} \times 2^{-4} &= 0.0001000_{two} = 0.0001_{two} \\ &= 1/2^4{}_{ten} = 1/16_{ten} = 0.0625_{ten} \end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Floating Point Multiplication

- **Adding** exponents
- **Multiply** the significands
- **Normalize** the result (check for overflow)
- **Round** to fit in available digits/bits
  - Normalize again if necessary
- Compute **sign** of result
  - Positive if signs of operands match, negative otherwise

Floating Point Multiplication

1.110e10 times 9.200e-5   with 4 digits precision

New exponent is 10-5 = 5

```
     1.110
     9.200
     -------
     0000
   0000
  2220
9990
----------
10212000
```

Note 3 decimal places in each number, so decimal now at 6th spot
10.212000e5

Normalizing give 1.0212e6

Rounding gives 1.021 e6

Multiplication Example

Example Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten} using the steps in Figure 4.46.

Answer In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{two} \times 2^{-3}$$

Converting to decimal to check our results:

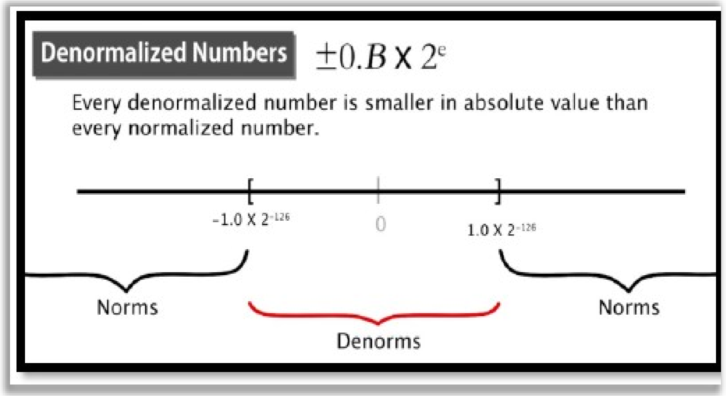
$$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^5{}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

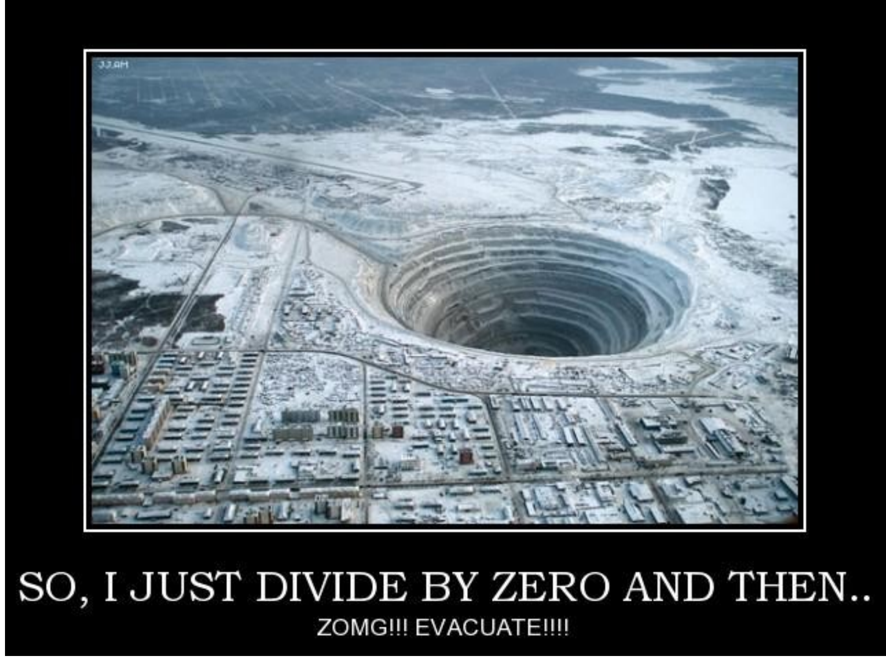
Special Cases?

Denormalized Numbers

- The exponent 00000000 is used to represent a set of numbers in the tiny interval (-2^{-126}, 2^{-126})
- This includes the number 0
- Called denormalized numbers
  - Smallest normalized is $1.0 \times 2^{-126} = 2^{-126}$
  - Smallest denormalized is $0.000 \blacksquare \blacksquare \blacksquare 01 \times 2^{-126} = 2^{-149}$
- Allows us to squeeze more precision out of a floating point operation
- Tricky to implement. We will come back to this topic later



Unusual events

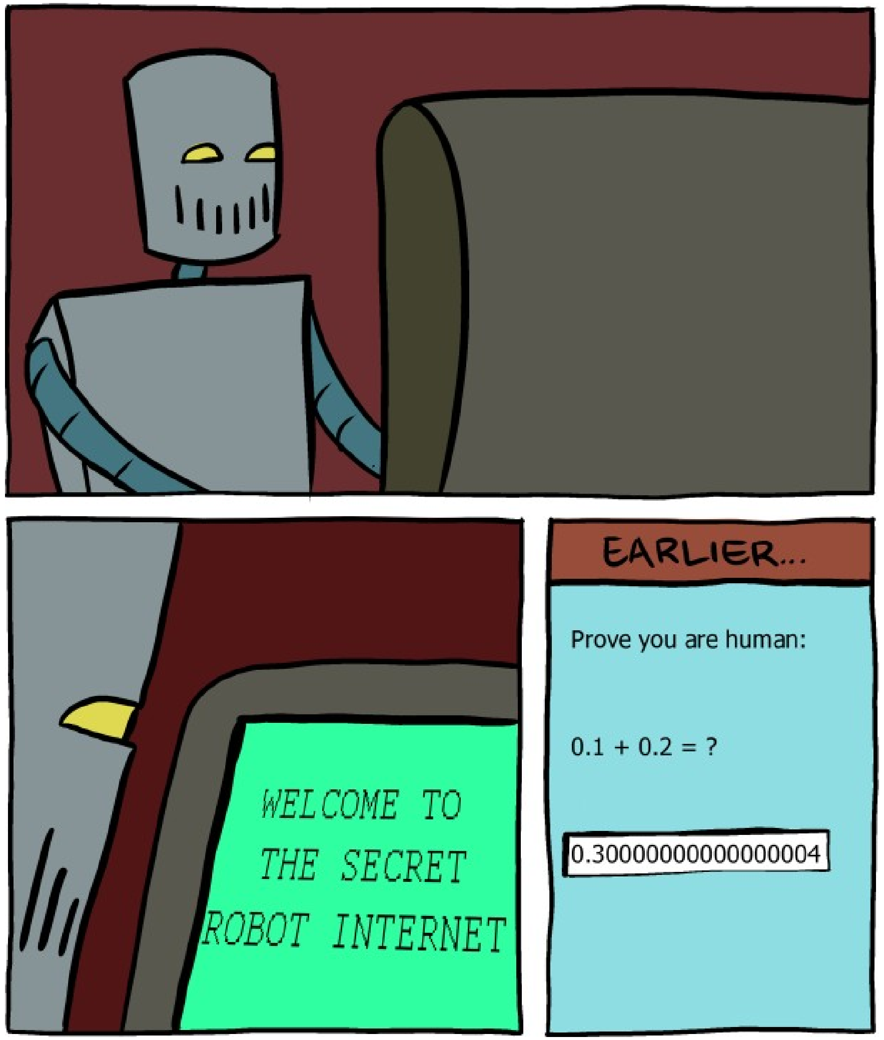


- Nonzero divided by zero
  - Not the end of the world!
  - Results in positive or negative infinity
- 0/0 (invalid), or subtracting infinity from infinity
  - Results in NaN
- Notes on NaN
  - Using NaN in math always results in NaN
  - Allows us to avoid tests or decisions until a later time in our program

Special symbols

Exponent	Fraction	Object represented
0	0	0
0	Nonzero	± denormalized number
1-254	Anything	± floating point number
255	0	± infinity
255	Nonzero	NaN (Not a Number)

Loss of Precision



https://imgur.com/r/totallynotrobots/lsNcv

Compare these for loops

```
for ( int i = 0; i <= 10; i += 1 ) {
   System.out.println( i/10f );
}

for ( float y = 0; y <= 1; y += 0.1f ) {
   System.out.println( y );
}
```

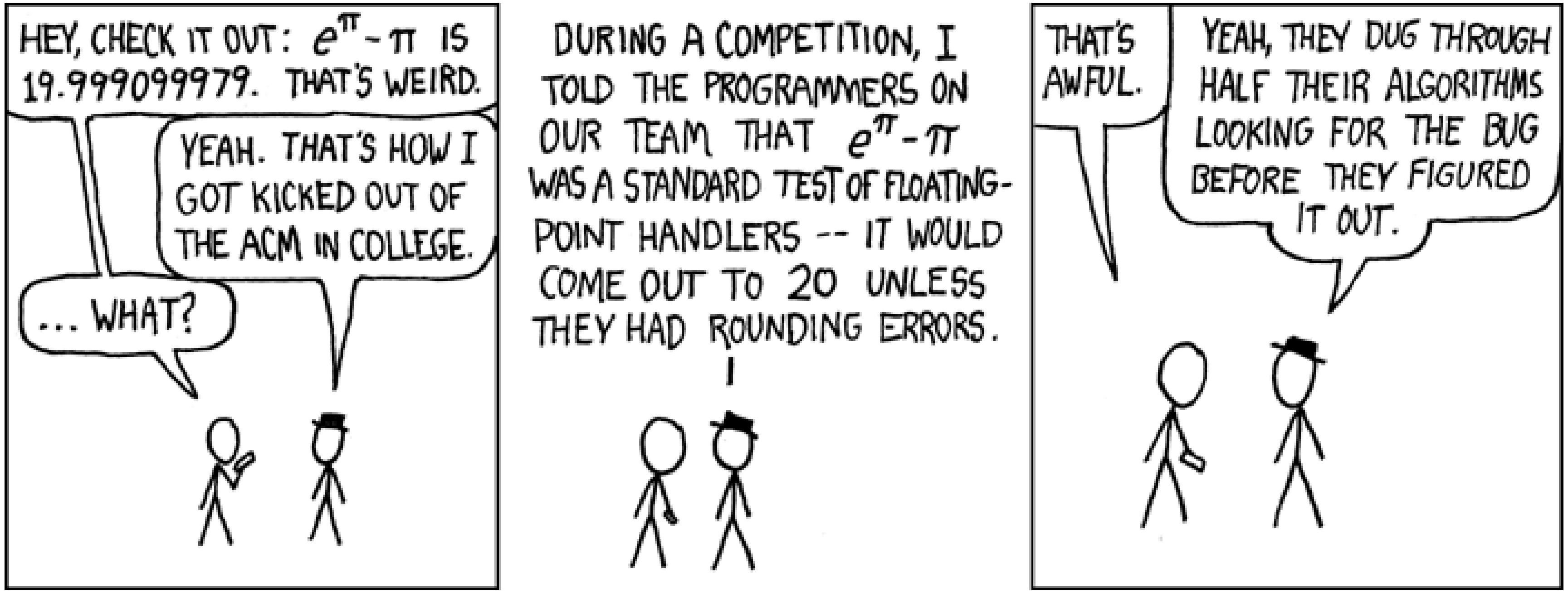
Same or different?

Questions

- Represent 0.1_{10} in IEEE 754 single precision floating point

- Represent 1.1_{10} in IEEE 754 single precision floating point?

Review and more information



- Big and Small Numbers
- Scientific Notation
- IEEE 754 floating point standard
- Floating point addition and multiplication
- Material from Section 3.5 of textbook