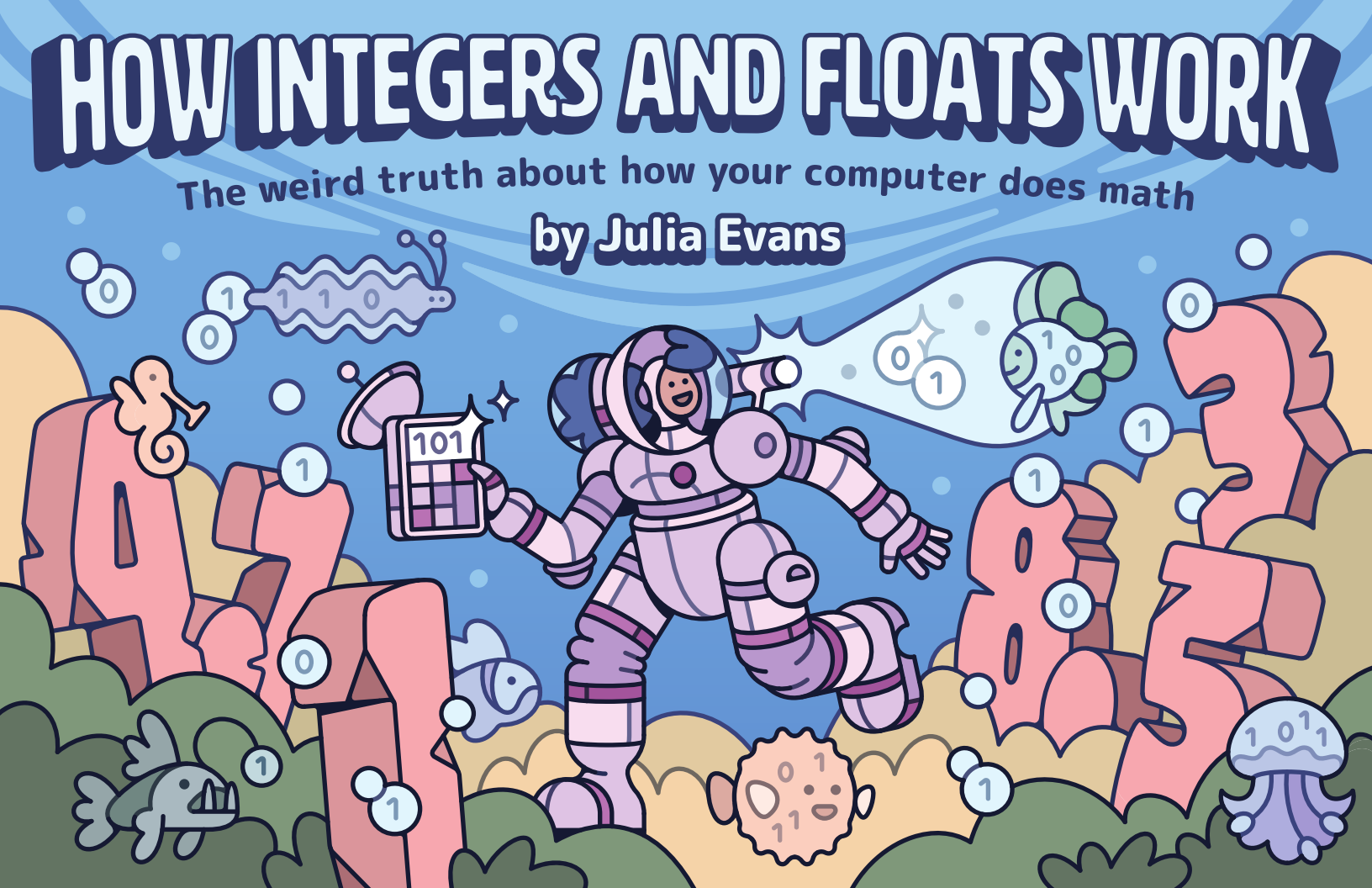
HOW INTEGERS AND FLOATS WORK

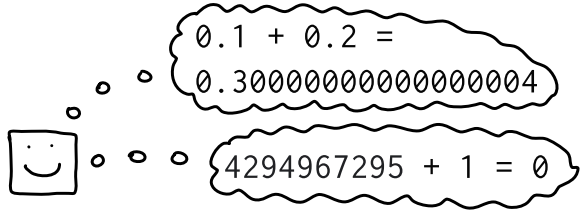
The weird truth about how your computer does math

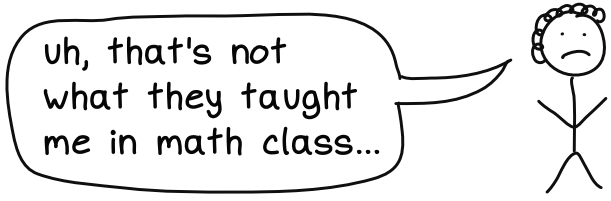
by Julia Evans



computers do math weird

Weird things happen when your computer does math.





The reason it gets so weird is that your computer has to cram each number into a limited number of bits (8, 16, 32, or 64 bits).

When your computer does math, it's running CPU instructions. And there are only 2 kinds of CPU math instructions: those that work on integers, and those that work on floating point numbers.

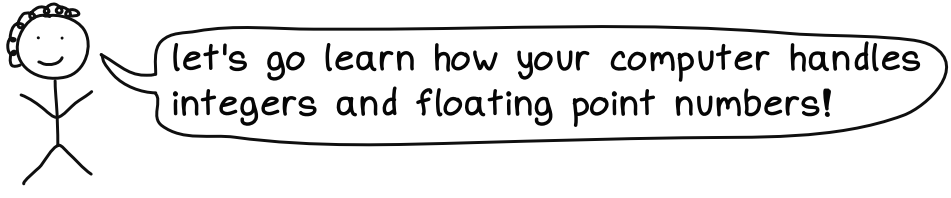


table of contents

integers

floats

meet the byte

You might have heard that a computer's memory is a series of bits (0s and 1s)...

01010011010101010110111

but you only access them in groups of 8 bits — a byte!

01010011 01010101 10110111

2 ways to think about a byte

1. 8 bits
2. an integer from 0 to 255

00000000 = 0
00000001 = 1 ← integer!
00000010 = 2
01011001 = 89

8 bits!

you can't just access 1 bit

Every byte in your computer's memory has an address.

If you want to fetch 1 bit, you need to fetch the whole byte at that address and then extract the bit.

01011001
1 bit

some things that are 1 byte

the boolean `true` (in C)
00000001

the ASCII character `F`
01000110

the red part of the colour #FF00FF
11111111

most things are more than one byte

→ integers and floats are usually 4 bytes or 8 bytes

→ strings are LOTS of bytes (for example, in UTF-8 a heart emoji is 3 bytes)

bytes weren't always 8 bits

In the past, people experimented with lots of different byte sizes (2, 3, 4, 5, 6, 8, and 10 bits!)

But now we've standardized on 8 bits pretty much everywhere.

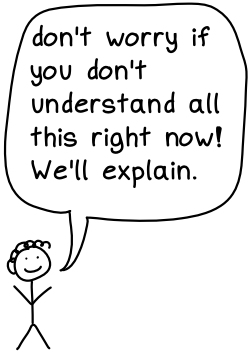
8 bytes, many meanings

Bytes can be decoded in many different ways. Here are 8 bytes and a bunch of things they could potentially mean:

8 bytes (written as integers) → 99 111 109 112 117 116 101 114

c o m p u t e r	8 ASCII characters
99 | 111 | 109 | 112 | 117 | 116 | 101 | 114	8 8-bit integers
28515 | 28781 | 29813 | 29285	4 16-bit integers (little endian)
1886220131 | 1919251573	2 32-bit integers (little endian)
8243122740717776739	1 64-bit integer (little endian)
7165065861944075634	1 64-bit integer (big endian)
99.111.109.112 | 117.116.101.114	2 IPv4 addresses
2.93930e+29 | 4.54482e+30	2 32-bit floating point numbers
1.144493e+243	1 64-bit floating point number
rgba(99, 111, 109, 0.44) | rgba(117, 116, 101, 0.45)	2 RGBA colours
arpl WORD PTR [edi+0x6d],bp | jo 0x7a | je 0x6c | .byte 0x72	x86 machine code

this code is nonsense, but search "ascii shellcode" for x86 code which is valid ASCII



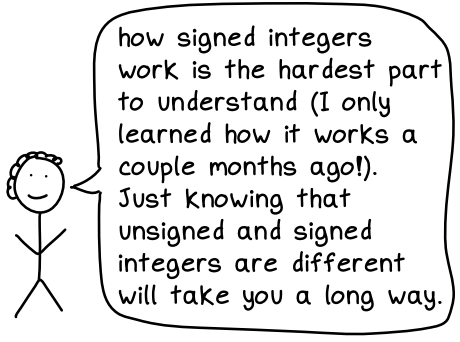
integers

To decode bytes as integers, we need to know 3 things:

① the integer's size (8 bit, 16 bit, 32 bit, or 64 bit)

② is it little or big endian? (see page 7)

③ is it signed or unsigned? (see page 8)



2 bytes, 3 interpretations

We could interpret these 2 bytes as:

① 254 (little endian)
② 65024 (big endian, unsigned)
③ -512 (big endian, signed)

how you decode bytes depends on the context

- in a program's memory, the type of the variable tells you the integer's size and if it's signed/unsigned
- your CPU determines if integers are big or little endian (you don't have a choice)
- for a binary network protocol (like DNS), the specification (for DNS, that's RFC 1035) will tell you how to decode the bytes

examples of types

in Rust, an `i64` is a signed 64-bit integer

in Go, a `uint32` is an unsigned 32-bit integer

in C, a `short` is usually a signed 16-bit integer, depending on the platform

little endian / big endian

we write dates in two main orders

① 2023-03-17 ("big endian")

② 17-03-2023 ("little endian")

? 03-17-2023 ("american" 🤷)

"big endian" means that the big unit (the year) is at the start ("big end first")

similarly: computers order bytes in 2 ways

Here are 2 ways your computer might represent the integer 271:

① big endian: `00000001` `00001111`

② little endian: `00001111` `00000001`

How this corresponds to 271:
`0000000100001111` is 271 in binary

a little history

Before 1980, computers ordered their bytes in different ways.

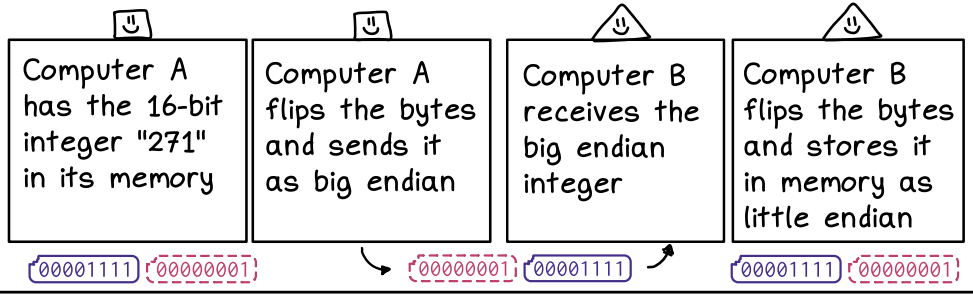
In 1980, the Internet started being standardized, causing a huge fight over which byte order to use on the Internet.

The terms "big/little endian" come from that fight: they were coined in an article called "On Holy Wars and a Plea For Peace" which compares the byte order fight to the Big/Little Endians in Gulliver's Travels.

Big endian won that fight, so most Internet protocols (IPv4, TCP, UDP, etc.) are big endian.

But almost all modern computers are little endian. Some machines, like the Xbox 360, are big endian though.

When you send integers on a computer network, they have to be big endian. Here's how that works:



signed vs unsigned integers

there are 2 ways to interpret every integer

unsigned:

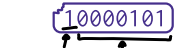
- always 0 or more
- example: 8 bit unsigned ints are `0` to `255`

signed:

- half positive, half negative
- example: 8 bit signed ints are `-128` to `127`

negative integers are represented in a counterintuitive way

You might think that this is `-5`:



sign bit — `101` in binary is `5`

But actually this is `-5`:

this looks weird, but we'll explain why!

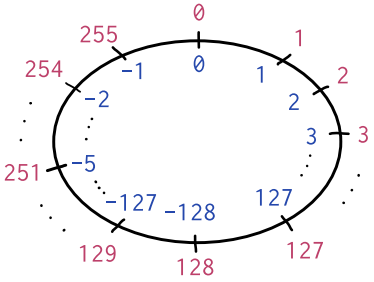
integer addition* wraps around

for example, for 8-bit integers `255 + 1 = 0`

for 16-bit integers, `65535 + 1 = 0`

*by "addition", we mean "what the x86 `add` instruction does"

but if `255 + 1 = 0`, you could also say `255 = -1`



examples of bytes and their signed/unsigned ints

byte	unsigned	signed
00000000	0	0
01111111	127	127
10000000	128	-128
10000001	129	-127
11111011	251	-5
11111111	255	-1

subtract 256 from unsigned numbers to get the signed numbers

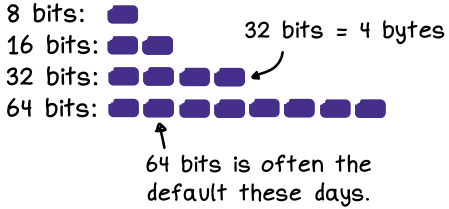
this way of handling signed integers is called "two's complement"

It's popular because you can use the same circuits to add signed and unsigned integers. `5 + 255` has exactly the same result as `5 + (-1)`: they're both `4`!

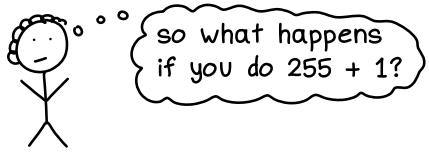
integer overflow

integers have a limited amount of space

The usual sizes for integers are:



the biggest 8-bit unsigned integer is 255



Going above/below the limits is called overflow.

ways overflow is handled

① wrap around

255 + 1 = 0

255 + 3 = 2

② raise an error

③ saturate ← this one is unusual

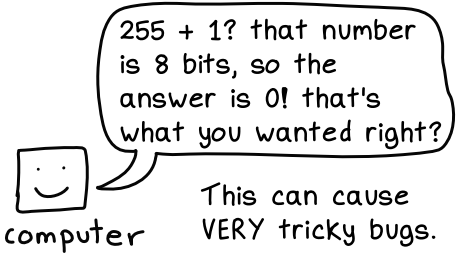
255 + 1 = 255

255 + 3 = 255

maximum numbers for different sizes

bits	signed	unsigned
8	127	255
16	32767	65535
32	~2 billion	~4 billion
64	~9 quintillion	~18 quintillion

overflows often don't throw errors



This can cause VERY tricky bugs.

some languages where integer overflow happens

Java/Kotlin C/C++ Rust

SQL C# Swift Go R

Dart

Some throw errors on overflow, some don't, for some it depends on various factors. Look up how it works in your language!

big integers

integers don't have to overflow

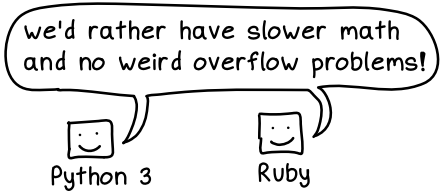
Instead, integers can expand to use more space as they get bigger. Integers that expand are called "big integers".

big integer math is slower

It's slower because it's implemented in software, not hardware.

So a big integer addition is actually turned into lots of smaller additions.

some languages only have big integers



This works because people don't do a lot of math in Ruby/Python (except with numpy, which doesn't use big integers).

some languages offer big integers as an option

Go, Javascript, Java, and lots more.

Each language has its own big integer implementation.

how big integers are represented

(in Go, as of 2023)

```
type Int struct {
    neg bool    // sign
    abs []uint // absolute value
}
```

array of "digits", each digit is 64 bits

You can think of this array of 64-bit integers as being the number written in base 2^{64}.

when are big integers useful?

- they're used in cryptography (e.g. for large key sizes)
- for math on really big integers

32 bits is small

using 32-bit integers is dangerous

Let's see some examples of how it can go wrong and why it's often better to use 64-bit integers instead!

(32-bit floats have similar problems)

32 bit integers are at most 4 billion

unsigned 32-bit ints go from
0 to 4,294,967,295 ← 4 billion

signed 32-bit ints go from
-2,147,483,648 to 2,147,483,647

times "4 billion" wasn't enough

Database primary keys: 4 billion records really isn't that much.

IPv4 addresses: turns out we want more than 4 billion computers on the internet. Oops.

Registers: in the 90s, registers were 32 bits. 4 billion bytes of RAM is 4GB. We need more than that.

Unix timestamps: 2 billion seconds after Jan 1, 1970 is Jan 19, 2038. That's going to be an exciting day. (look up "2038 problem"!)

64 bits is usually big enough

For example, 2^{64} seconds after Jan 1, 1970 is over 100 billion years in the future: well after the death of the sun.

So a 64-bit timestamp is definitely enough space.

be wary of using 32-bit integers by accident

Systems that were designed in the 90s often have a 32-bit integer as the default.

For example, in MySQL an `INTEGER` is 32 bits.

bases

We usually write numbers in base 10, but you can write numbers in any base.

Let's write the number 103 in 3 different bases:

base 10: 103

1	0	3
×	×	×
100	10	1

← powers of 10

100 + 0 + 3 = 103

base 2: 1100111

1	1	0	0	1	1	1
×	×	×	×	×	×	×
64	32	16	8	4	2	1

← powers of 2

64 + 32 + 0 + 0 + 4 + 2 + 1 = 103

base 16: 67

6	7
×	×
16	1

← powers of 16

96 + 7 = 103

base 2, 10, and 16 are the main bases we use on computers

base 2 is called binary

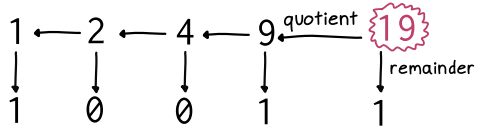
base 10 is called decimal

base 16 is called hexadecimal

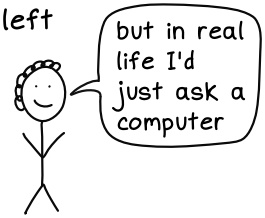
how to convert from base 10 to base 2

Let's convert 19! We'll start on the right and move left.

1. Divide by 2: 19/2 = 9 remainder 1
2. Write the remainder (1) below, and 9 on the left
3. Repeat



answer: 10011!

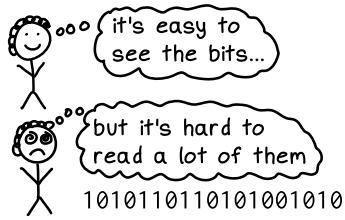


hexadecimal

let's talk about how to write binary data

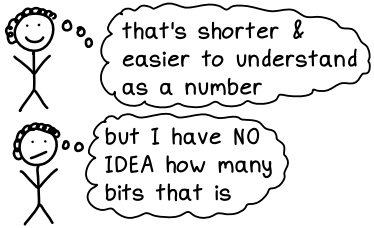
one way: binary

01111111 11111111 11111111



another way: base 10

83888607

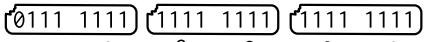


now the best way to write binary data: base 16!

It's short AND maps well to bits!

7fffff

Every hexadecimal digit represents 4 bits. So 1 byte (8 bits) is always 2 hexadecimal digits.



there are 16 hex digits

0 → f

hex	decimal	binary	hex	decimal	binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	a	10	1010
3	3	0011	b	11	1011
4	4	0100	c	12	1100
5	5	0101	d	13	1101
6	6	0110	e	14	1110
7	7	0111	f	15	1111

0x means it's hex

In many languages, the `0x` prefix lets you write numbers in hexadecimal.

For example, in C:

```
0x20 == 32      (base 16)
0b10100 == 20   (base 2)
061 == 49       (base 8)
```

↖ be careful: the 0 prefix meaning "base 8" can really trip you up!

things hexadecimal is used for

- → color codes! (e.g. `#FF00FF`)
- → memory addresses!
- → hashes! (like git commit IDs)
- → displaying binary data! (like with `hexdump`)

bitwise operations

bitwise operations operate one bit at a time

The results can be surprising when you write them in base 10:

8 & 3 = 0

but in binary it makes more sense:

```
  00001000 ← 8
& 00000011 ← 3
----------
  00000000
```

&

Bitwise and: the result is 1 if BOTH bits are 1

```
1 & 1 = 1
1 & 0 = 0
0 & 0 = 0
11 & 10 = 10
```

|

Bitwise or: the result is 1 if EITHER bit is a 1

```
1 | 1 = 1
1 | 0 = 1
0 | 0 = 0
11 | 10 = 11
```

^

Exclusive or: the result is 1 if EXACTLY ONE bit is a 1

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 0 = 0
11 ^ 10 = 01
```

~

Bitwise not: FLIP all the bits

```
~0 = 1
~1 = 0
~10 = 01
```

<<

Left shift: add 0s to the end

```
1110 << 3
= 1110000
```

<< n is the same as multiplying by 2^n

>>

Right shift: chop bits off the end

```
01100001 >> 2
= 00011000
```

>> n is the same as dividing by 2^n (rounded down)

there are actually two right shifts

unsigned right shift

253 >> 1 = 126

11111101 → 01111110

always pad on the left with a 0

signed right shift

-3 >> 1 = -2

11111101 → 11111110

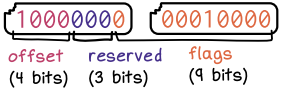
if the number is negative, pad on the left with 1 instead of a 0

In some languages, unsigned right shift is >>>. In other languages, both right shifts are >> and the integer's type determines which is used.

how bitwise operations are used

Binary formats often pack information into bytes very tightly to save space.

For example, here are 2 bytes from a real TCP packet:

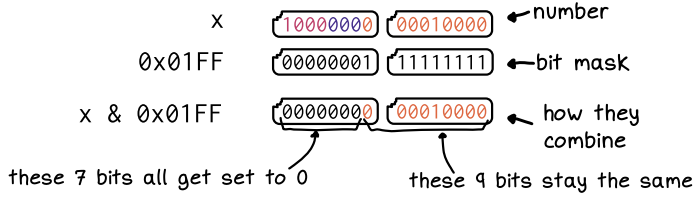


Here's how &, |, <<, >> can be used to pack/unpack data into bytes.

bit masking

Let's say we have the 2 bytes from the previous panel, and we want to extract just the flags part. Here's how to do it with & (bitwise and):

The idea is that you put a mask "on top" of the bytes to erase bits:



check/set bit flags

(see page 16 for more)

set a bit flag with or:

```
x = x | 0b010000;
```

check a bit flag with and:

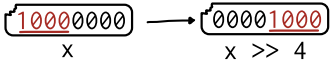
```
if ((x & 0b010000) != 0) {
...
}
```

this example is in C

unpack/pack bits

Now let's talk about the offset from the first panel. We can't do calculations in it with the packed form, so we need to unpack it.

You can unpack with >>:



and pack with <<:

1000 in binary is 8, which in this case is the TCP offset value.

bit flags

bit flags are a clever way to store lots of information in one integer

If you have many options which are true or false, you can encode them all into an integer, with 1 bit for each option. 32 bits = 32 options!

For example, some of the bit flags the `open` function in C uses:

(this is on Linux)

where you'll see bit flags

In libc, the `open`, `socket`, and `mmap` functions use bit flags to pass options.

The TCP and UDP protocol headers both have a `flags` field which has bit flags.

bit flags are used a lot in C code

Here's some C code that opens a new file:

```
fd = open("file.txt", O_RDWR | O_CREAT, 0666);
```

bitwise or!

the file permissions (in base 8)

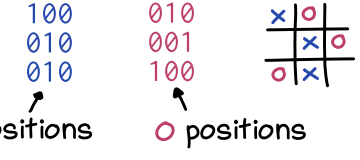
O_RDWR is:	0000010
O_CREAT is:	0100000
O_RDWR | O_CREAT is:	0100010

You can check if a bit flag is set in C like this:

```
if (flags & O_RDWR) { ... }
```

fun example: tic tac toe!

Here's a way to encode the state of a tic tac toe game in 18 bits:



floating point is weird

floating point 10.0 is not the same as the integer 10

10 (64-bit integer):

`0x000000000000000a`

10.0 (64-bit float):

`0x4024000000000000`

what's this `4024` doing???

computer integers work almost exactly the way you'd expect

`1 + 2 - 3 = 0`

but floating point numbers don't:

```
(0.1 + 0.2) - 0.3 =
0.00000000000000000555
```

checking for float equality is dangerous

`if x == 0.3:` ← bad!

`(0.1 + 0.2)` is not equal to `0.3`!

Instead, check if `x` is very close to `0.3`, something like this:

```
if abs(x - 0.3) < 0.0000001:
```

in floating point, very large integers get rounded

For example: `10000000000000001.0 == 10000000000000000.0`

16 zeros

(try comparing those 2 numbers in your favourite language! they're the same!)

(x + y) + z is not the same as x + (y + z)

For example: `(9007199254740992.0 + 1.0) - 1.0 = 9007199254740991.0`

(the math term for this problem is "floating point addition isn't associative")

some intuition for precision

32-bit floats have about 8 digits of precision

64-bit floats have about 16 digits of precision

the gaps between floats

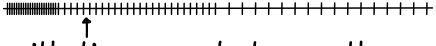
floating point numbers have to fit into 32 or 64 bits

This means there are only 2^{64} 64-bit floats, the same way there are only 2^{64} 64-bit integers.

(terminology note: a 64-bit float is often called a "double")

this means floating point numbers have to be spread out

You can imagine them all spaced out on a number line, like this:



with tiny gaps between them

the gaps start small

the next 64-bit float after `1.0` is `1.0000000000000022204`

the gap between those two floats is `0.0000000000000022204`, or 2^{-52}

(gaps are always a power of 2)

the gaps get bigger as the numbers get bigger

The next 64-bit float after `10000000000000000000.0` is `10000000000000016384.0`.

So the gap is 16384, or 2^{14}!

the gaps make calculations inaccurate

When you do math on floating point numbers, often you have to round the result to the nearest float.

Usually this doesn't make a big difference, but small mistakes can add up.

this inaccuracy is inevitable

Floating point was actually designed very thoughtfully. But if you cram infinite numbers into 64 bits, you have to sacrifice some accuracy.

science ♥ floating point

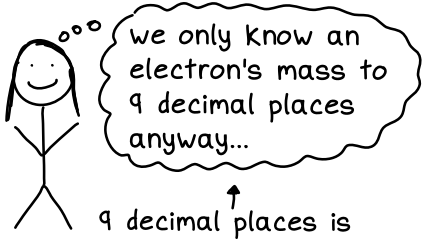
floating point was invented to do scientific computation

weather simulations!

earthquake modeling!

orbital mechanics!

scientists don't need unlimited precision...



... but they do need TINY numbers and GIANT numbers

mass of hydrogen atom:

$1.6735575 * 10^{-24}$ grams

distance to Andromeda galaxy:

$2.4 * 10^{22}$ meters

floating point is inspired by scientific notation

$1.6735575 \times 10^{-24}$ g

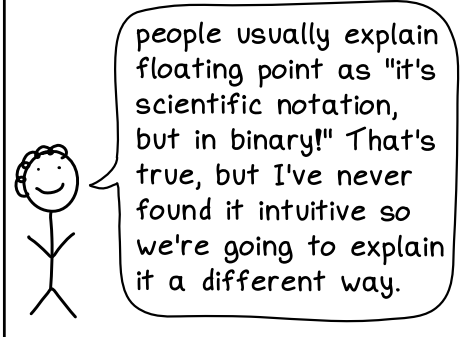
The idea in floating point is to store a number by splitting it into:

- the exponent (like -24)
- the multiplier (like 1.6735575)
- and its sign (+ or -)

floating point isn't just used for science though

For example, Javascript's `number` type is floating point. Before it added `BigInt` in 2021, Javascript didn't have integers at all!

Similarly, numbers in JSON are often interpreted as floating point numbers.

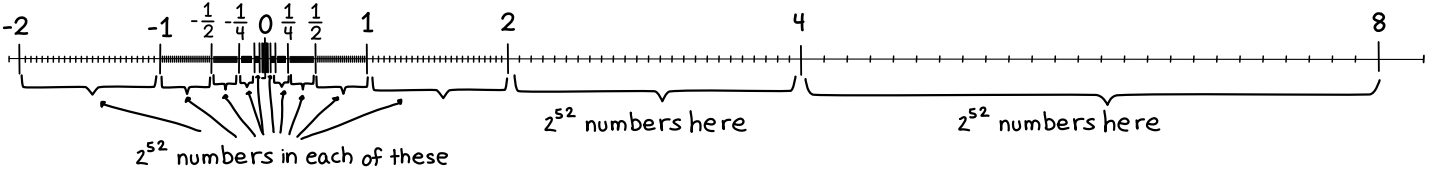


the floating point number line

the (64-bit) floating point number line

Floating point numbers aren't evenly distributed. Instead, they're organized into windows: [0.25, 0.5], [0.5, 1], [1,2], [2,4], [4,8], [8,16], all the way up to $[2^{1023}, 2^{1024}]$.

Every window has 2^{52} floats in it.



the windows go from REALLY small to REALLY big

The window closest to 0 is $[2^{-1023}, 2^{-1022}]$.
This is TINY: a hydrogen atom weighs about 2^{-76} grams.

The biggest window is $[2^{1023}, 2^{1024}]$
This is HUUUGE: the farthest galaxy we know about is about 2^{90} meters away.

the gaps between floats double with every window

window	gap	
[1, 2]	2^{-52}	
[2, 4]	2^{-51}	
[4, 8]	2^{-50}	
[8, 16]	2^{-49}	

why does 10000000000000000.0 + 1 = 10000000000000000.0?

- In the window $[2^n, 2^{n+1}]$, the gap between floats is 2^{n-52}
- 10000000000000000.0 is in the window $[2^{53}, 2^{54}]$, where the gap is 2^1 (or 2)
- So the next float after 10000000000000000.0 is 10000000000000002.0

the bits

Floats need to fit into 64 bits. But how do we actually convert a number like 10.87 into 64 bits?

First, we split the number into 3 parts: the sign, a power of 2 and an offset

$$10.87 = +(8 + 2.87)$$

biggest power of 2 that's less than 10.87

the usual term is "significand", but I find that term confusing so we're calling it "offset"

Next, we encode the sign, power of 2, and offset into bits!

encoding the sign (1 bit)

\+ is 0

\- is 1

floating point encoding is defined in the IEEE 754 standard

since it's standardized, it works the same way on every computer!

it was originally defined in 1985

encoding the exponent (11 bits, 2^{-1023} to 2^{1023})

8

↓ $2^3 = 8$

3

↓ add 1023 (this makes sure that the result is positive)

1026

↓ write it in binary, in 11 bits

10000000010

encoding the offset (52 bits)

2.87

↓ divide by the gap size, 2^{-49} in this case ($2^{\text{exponent}-52}$)

1615666366319165.3

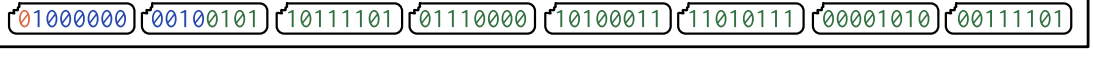
↓ round

1615666366319165

↓ write it in binary, 52 bits

0101101111010111000010100011111010111000010100011101

And here's 10.87!



NaN and infinity

NaN stands for "not a number"

It means the result of the calculation is undefined.

```
0/0 = NaN
sqrt(-1) = NaN
log(-1) = NaN
```

infinity

"Infinity" just means "this number is too big for floating point to handle." There are two infinities: one positive, one negative.

```
2.0**1024 = inf
-1 / 0 = -inf
inf - 10 = inf
inf - inf = NaN
```

`2.0**1024` means 2^{1024}

NaNs spread

As soon as one NaN gets in, it gets everywhere

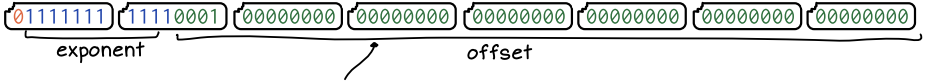
```
NaN * 5 = NaN
NaN + 2 = NaN
```

NaN != NaN

NaN isn't equal to anything (including itself)

NaN and infinity: the bits

A floating point value is `NaN` or `infinity` if the bits in the exponent are all 1. For example, this is a `NaN`:



It's `infinity` if the offset bits are all 0, otherwise it's `NaN`.

There are 2^{52} values like this: 2 of them are `±infinity` and the other $2^{52}-2$ are `NaN`.

We usually treat `NaN` like a single value though.

a note on byte order

All of the floating point examples in this zine use a big endian byte order, because it's easier to read. But most computers use a little endian byte order (see page 7).

You can see this in action at `https://memory-spy.wizardzines.com`

how floats are printed

computers lie when they print out floats

(by rounding)

For example `0.12` isn't `0.12`, it's actually (roughly):

`0.11999999999999995559`

the string -> float translation

If your program says:

```
x = 0.12
```

your interpreter / compiler needs to translate `"0.12"` into the float `0.11999999999999995559`. Most languages will use the `strtod` ("string to double") function from libc to do that translation.

the float -> string translation

This is where the rounding comes in. Computers round to make the numbers shorter and easier to read.

`1.1999999999999995559`
↳ `1.2`

float -> string translation is actually super complicated

Every floating point number needs a unique string representation.

There are a bunch of academic papers about how to do this well, search "Printing floating point numbers accurately" to read more about it.

some examples of printing floats

`1.1990000000000006573`
↳ `1.199`

`1.1999999900000001637`
↳ `1.19999999`

`1.1999999999998996358`
↳ `1.19999999999999`

`1.1999999999999995559`
↳ `1.2`

you can also print floats in base 16 or base 2

For example, `0.1` as a 32-bit float is:

`p-4` is the base 16 version of `e-4`

base 16: `0x1.99999ap-4`

base 2: `1.10011001100110011001101p-100`

The base 2 / base 16 representations are not rounded, but they're rarely used.

floating point math

let's deconstruct `0.1 + 0.2`

① The closest 64-bit float to `0.1` is (roughly)
`0.1000000000000000055511151231`

② For `0.2`, it's (roughly)
`0.2000000000000000111022302462`

③ `0.1000000000000000055511151231 +`
`0.2000000000000000111022302462 =`
`0.3000000000000000166533453693`

④ Inconveniently,
`0.3000000000000000166533453693`
is exactly in between 2 floating point numbers:
`0.29999999999999988977` and
`0.30000000000000004440892`

⑤ How do we pick the answer?
`0.30000000000000004440892` has an even offset (see page 21), so we round to that one

losing a little precision is okay

`0.1 + 0.2 = 0.30000000000000004`
is usually no big deal. Do you REALLY need your answer to be accurate to 16 decimal places? Probably not!

the more numbers you add, the more precision you lose

This Go code:

```
var meters float32 = 0.0
for i := 0; i < 100000000; i++ {
    meters += 0.01
}
fmt.Println(meters)
```

prints out `262144`, not `1000000` because `262144.0 + 0.1 = 262144.0`

adding a number to a MUCH smaller number is bad

For example:

```
2**53 + 1.0 = 2**53

1.0 + 2**-57 = 1.0
```

(try it!)

use scientific computing libraries if you can

There are special algorithms for adding up lots of small floating numbers without losing accuracy!

For example `numpy` implements them.

fixed point

just because you see 0.23, doesn't mean it's floating point

For example, in this RGBA color:

```
rgba(211, 7, 23, 0.23)
                 ↳ opacity
```

`0.23` isn't a float at all, it's the 8-bit integer `59`. Let's see how that works!

fixed point numbers are integers

You interpret them as the integer divided by some fixed number (like 255 or 10000)

For example, that opacity should be divided by 255

```
59 / 255 = 0.23ish
```

things fixed point is often used for

money	`$1.23 => 123`
time	`0.1 seconds => 100000 microseconds`
opacity	`0.23 => 59`

fixed point is the most common alternative to floating point

It's very simple and it's pretty easy to implement!

implementing fixed point is easy

(especially if you only need to add and subtract)

You just need:

- → an integer
- → some code to display it (by dividing by 255 or something)

fixed point can help avoid accuracy issues

If you try to represent the current Unix epoch in nanoseconds as a 64-bit float, you'll lose accuracy.

But if it's a 64-bit integer, it'll be fine.

more floating point alternatives

there are many alternative ways to represent numbers

These are all implemented in software (not hardware) so they're a lot slower, and different languages have different libraries.

alternative 1: decimal floating point

This is like regular floating point, but in base 10 instead of base 2. It's also standardized in IEEE 754.

Examples: Python's `decimal` module or Java's `BigDecimal`

alternative 2: fractions

This lets you do exact calculations with fractions (1/10 + 2/10 = 3/10)

Examples: Python's `fractions` module in the standard library, Lisps have first-class support

alternative 3: symbolic computation

For example, `sqrt(2)` instead of `1.414`.

You'll see this in computer algebra systems like Mathematica, Maple, or sympy.

alternative 4: interval arithmetic

The idea is to store every number as a range so that you can precisely track your error bars.

Probably the least mainstream of these alternatives.

alternative 5: binary-coded decimal

This is how floating point numbers (and integers) were stored on IBM computers in the 60s, and you can still occasionally see it today in old formats like ISO 8583 for financial transactions.

thanks for reading

If you want to learn more, my favourite ways to learn about integers and floats have been to:

- experiment to find out how my favourite programming languages handle some of the edge cases in this zine (like integer overflow!)
- play around with the bits in a float at `https://float.exposed` (or the bits in an integer at `https://integer.exposed`)
- use a debugger to see how integers and floats are actually represented in a program's memory. You can try this at `https://memory-spy.wizardzines.com`

credits

Pairing: Marie LeBlanc Flanagan
Cover illustration: Vladimir Kašiković
Editing: Dolly Lanuza, Kamal Marhubi
Copy editing: Gersande La Flèche
Technical review: an anonymous friend

♡ this?
more at
★ wizardzines.com ★