Data representation

Floating-point numbers and multiple-precision

Data representation

Real numbers

Example of exact representations:

$$123.456 = 123456 \times 10^{-3} = 1.23456 \times 10^{2}$$

significand x base $^{\text{exponent}}$

Integers

42 : what is the meaning of all this?

how does the computer store these numbers?

what are the limitations?

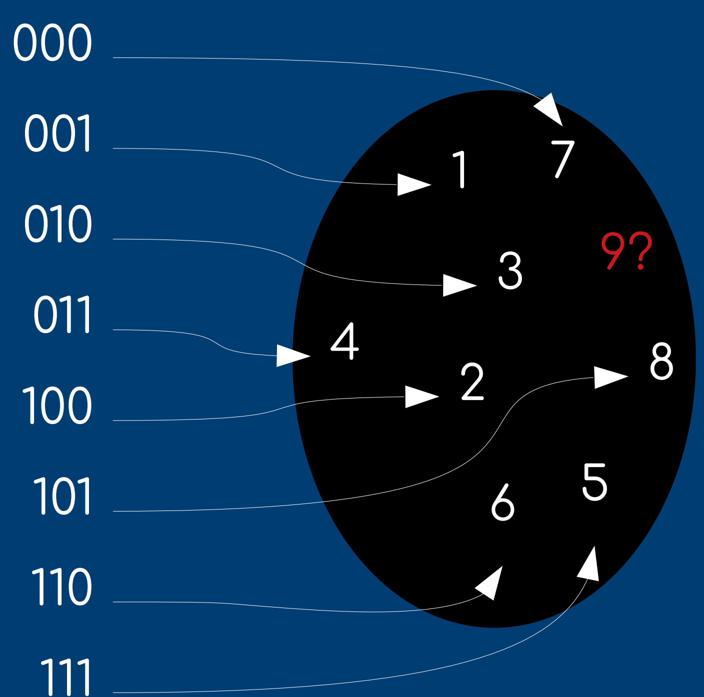
8-bits, 16-bits, 32-bits, 64-bits, 128-bits ... ?

(nes, snes, ps, n64, ps2, xbox, ... ;-)

Each CPU implements native n-bits operations [current PCs: 64-bits]

Example: it limits the memory allocation and data representation

A 3-bit memory controller
can only point to $2^3 = 8$ addresses



A 3-bit memory bank
can only represent 8 abstract entities

000 = apple

001 = avocado

010 = banana

011 = lemon

100 = pineapple

101 = strawberry

110 = kiwi

111 = grape

orange?

Integer representation

From the previous slide, a 3-bit computer can only represent 8 integers

3-bit binary memory

1	0	1
2	1	0

$$101 = 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} = 5$$

Min: (000)b = 0
Max: (111)b = 7

Signed 32-bit integer standard

1	1	1	1	1	1	...	1	1

Sign +/-

+ 31-bits

$$= (-1)^{b31} \times (b_{30} \times 2^{30} + b_{29} \times 2^{29} + \dots b_{0} \times 2^{0})$$

"sign-magnitude representation"

Range 9+ digits

Min: $-2^{31} = -2{,}147{,}483{,}648$
Max: $+2^{31}-1 = +2{,}147{,}483{,}647$

"2's complement representation"

Real numbers : floating point

significand x base $^{\text{exponent}}$

Signed 32-bit float standard

exponent (8 bits)
= signed int

|significand| (23 bits)

sign of the significand

While the set of real numbers is dense / continuous…

only a limited number of entities can be represented on a finite structure

→ loss of precision

Comparing real numbers

Let us try this in C

1) initialize the variables as

```
x = 0.1
y = 3*x
z = 0.3
```

2) check the output of the equality tests

```
if (z == 0.3) { … }
if (z == y) { … }
If (z == 3*0.1) { … }
```

3) What is happening? Let us print x, y, z with many digits

```
printf(‘x = 0.30f\n‘, x)
printf(‘y = 0.30f\n‘, y)
printf(‘z = 0.30f\n‘, z)
```

Why is 0.1 = 0.1000000000000000055511151231 26 ?

If a float in simply stored as significand x base $^{\text{exponent}}$

shouldn't we have $0.1 = 1 \times 10^{-1}$?

… but computers use base 2 !!!

Examples

→ the fraction 1/10 in base 10 is 0.1 (exact)

→ the fraction 1/3 in base 10 is $0.\overline{3}$ = 0.333333… ~ 0.333334

→ the fraction 1/10 in base 2 is $0.0\overline{0011}$ = $0.0\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}\ldots$

An exact fraction in base 10 might be a repeating fraction in base 2

Truncation yields: $(0.0\underbrace{0011}\underbrace{0011}\underbrace{0011})_2 = (0.0999755859375)_{10}$

To convert from decimal to binary: [https://www.rapidtables.com/convert/number/binary-to-decimal.html]

Multiple, or arbitrary-precision

Default types:

Allows you to freely choose the precision

Floats: 32 bits ~ 7 digits, 64 bits ~ 16 digits, 128 bits ~ 34 digits

C

[int] = 32 bits
[float] = 32 bits
[double] = 64 bits
...

GMP: GNU Multiple Precision Arithmetic Library

[https://gmplib.org/]

MPFR: multiple-precision floating-point computations with correct rounding

[https://www.mpfr.org/]

Python

[int] = arbitrary
[float] = 64 bits

Python mpmath [http://mpmath.org/]

→ for multiple-precision in Python, uses GMP if available

Python SymPy [https://www.sympy.org/]

→ for Symbolic mathematics in Python

Example in Python

From "Why and How to Use Arbitrary Precision",

Ghazi et al, Computing in Science & Engineering 12, 62-65 (2010)

Let us try to calculate $d = 173746a + 94228b - 78487c$, with

$a = \sin(10^{22})$, $b = \log_{10}(17.1)$, $c = \exp(0.42)$

Standard Python

```
import numpy as np
a = np.sin(1e22)
b = np.log(17.1)
c = np.exp(0.42)
d = 173746*a + 94228*b - 78487*c
print('d =', d)
     2.9103830456733704e-11
```

Exact value:
$-1.341818958\ldots \times 10^{-12}$

The same Python example, but using mpmath

```
from mpmath import mp
mp.dps = 40 # defines the decimal precision
print(mp) # to check again

a = mp.sin('1e22')
b = mp.log('17.1')
c = mp.exp('0.42')
d = 173746*a + 94228*b - 78487*c
print('d = ', d)
      -1.341818957829610467062152588 14e-12
```

Exact value:
$-1.341818958\ldots \times 10^{-12}$

Notice that the real numbers are set by strings, otherwise Python would first convert to floating-point and lose precision. TRY IT!

Overall features of these libraries

	python numpy+scipy	python mpmath	C+GMP+MPFR
Float precision	64 bits	arbitrary	arbitrary

Trigonometric functions + Special functions	`np.sin(…)`
	`np.exp(…)`
	`sp.special.jv(…)`
	`sp.special.zeta(…)`
		`mp.sin(…)`
		`mp.exp(…)`
		`mp.besselj(…)`
		`mp.zeta(…)`
			`mpfr_sin(…)`
			`mpfr_exp(…)`
			`mpfr_jn(…)`
			`mpfr_zeta(…)`

Numerical calculus	Root finding, sum, quadrature (integrals), differentiation, ODE (RK4), Taylor, Fourier, …		Check other libraries: GSL mpack = blas+lapack Boost C++ [www.boost.org]
Linear algebra	Matrix/vector operations (products, inverse, determinant), SVD, linear systems, eigenproblems, matrix functions (exp, cos, …), …		

Exercise: factorial

Let us go back to our factorial implemented on a for loop:

```
def myfactorial(n):
    f = 1
    for i in range(1, n+1):
        f *= i
    return f
```

CENSORED

1) Try to run it for n=30

→ expected result: 265252859812191058636308480000000

2) Now, change your implementation to have the initial f as a float:

→ `f = 1.0`

3) And check again for n=10 and n=30. Float → loss of precision!

4) Change the code to initialize f as a multiple precision float