Numerical Representations

Luis Pedro Coelho

Programming for Scientists

September 16, 2012

INSTITUTO DE MEDICINA MOLECULAR

How Are Numbers Represented

It's all 0s & 1s. How do you represent 123?

Binary Notation

$$(b_4b_3b_2b_1b_0)_2 = b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 =$$
$$16b_4 + 8b_3 + 4b_2 + 2b_1 + b_0$$

Common Number Sizes

- Byte: 8 bits, 0 to 255 ($2^8 - 1$).
- Short: 16 bits, 0 to 65535 ($2^{16} - 1$).
- 32-bit int: 32 bits, 0 to 4294967295 ($2^{32} - 1$).
- 64-bit int: 64 bits, 0 to 18446744073709551615 ($2^{64} - 1$).

Bit-wise operations

1. NOT(A): true if A is not true (~A)
2. AND(A,B): true if A is true and B is true (A & B)
3. OR(A,B): true if either A or B are true (A | B)
4. XOR(A,B): true if one is true and the other is false A ^ B

```
def fact(N):
    if N == 0: return 1
    return N * fact(N-1)

print  fact(100)
```

Prints out
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000L

What about negative numbers?

- Sign bit
- Biasing
- Ones' complement
- Twos' complement

Sign Bit

$$(\mathrm{s b_4 b_3 b_2 b_1 b_0})_2 = (-1)^{\mathrm{s}} \left(\mathrm{b_4} 2^4 + \mathrm{b_3} 2^3 + \mathrm{b_2} 2^2 + \mathrm{b_1} 2^1 + \mathrm{b_0} 2^0\right)$$

Biasing

Have a bias B, so that the number n is representated as unsigned(n + B).

One's Complement

If $(\mathrm{b_k b_{k-1} \cdots b_1 b_0})_2$ is some number n, then we represent −n by $(\tilde{\mathrm{b}}_\mathrm{k} \tilde{\mathrm{b}}_\mathrm{k-1} \cdots \tilde{\mathrm{b}}_1 \tilde{\mathrm{b}}_0)_2$ is some number n, then we represent −n by

One's Complement

If $(b_k b_{k-1} \cdots b_1 b_0)_2$ is some number n, then we represent $-n$ by
$(\tilde{b}_k \tilde{b}_{k-1} \cdots \tilde{b}_1 \tilde{b}_0)_2$ is some number n, then we represent $-n$ by

$(00000011)_2$ is 3
$(11111100)_2$ is -3

$(00001111)_2$ is 31
$(11110000)_2$ is -31

One's Complement

If $(b_k b_{k-1} \cdots b_1 b_0)_2$ is some number n, then we represent $-n$ by
$(\tilde{b}_k \tilde{b}_{k-1} \cdots \tilde{b}_1 \tilde{b}_0)_2$ is some number n, then we represent $-n$ by

$(00000011)_2$ is 3
$(11111100)_2$ is -3

$(00001111)_2$ is 31
$(11110000)_2$ is -31

$(00000000)_2$ is 0
$(11111111)_2$ is -0

One's Complement

If $(b_k b_{k-1} \cdots b_1 b_0)_2$ is some number n, then we represent $-n$ by
$(\tilde{b}_k \tilde{b}_{k-1} \cdots \tilde{b}_1 \tilde{b}_0)_2$ is some number n, then we represent $-n$ by

$(00000011)_2$ is 3
$(11111100)_2$ is -3

$(00001111)_2$ is 31
$(11110000)_2$ is -31

$(00000000)_2$ is 0
$(11111111)_2$ is -0

Ones' complement is not actually used in any modern machine.

Twos' Complement

Image from Wikipedia
Metaphor from Steve Heller

Twos' Complement

$(11111111)_2$ is -1
$(11111110)_2$ is -2
$(11111101)_2$ is -3

Ranges

- 8 bits: -128 to 127.
- 16 bits: -32768 to 32767.
- 32 bits: -2147483648 to 2147483647.
- 64 bits: -9223372036854775808 to 9223372036854775807.

Fractional Numbers

What about fractional numbers?

- Fixed point
- Floating point

Fixed Point

Given a fixed base B, then an integer n really represents the number $n * 2^B$.

Floating Point

602214179303030303030303

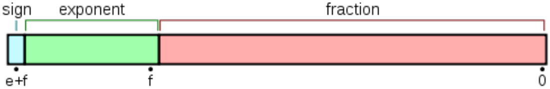
Floating Point

602214179303030303030303

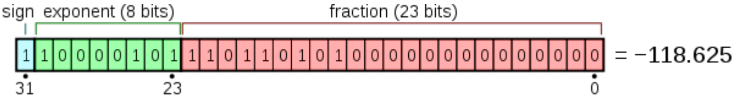
$6.022 * 10^{23}$

Floating Point Representation

$$(-1)^{s} * c * 2^{q}$$



IEEE-754


sign
exponent (8 bits)
fraction (23 bits)
1 1 0 0 0 0 1 0 1 1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
= −118.625
31
23
0

IEEE-754 Formats

- 32-bit floats: 1 sign bit, 23 bit fraction, 8 bit exponent.
- 64-bit floats: 1 sign bit, 52 bit fraction, 11 bit exponent.
- Non-standard 80-bit floats: 1 sign bit, 64 bit fraction, 15 bit exponent.

Ranges

- 32-bit float: $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$.
- 64-bit float: $\pm 2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$.

Limited Precision

```
print 0.3 * 3
print (0.3 * 3) == .9
```

prints

.9
False

```
print 1.1 * 0 == 0.0
print 1.1 * 1 == 1.1
print 1.1 * 2 == 2.2
print 1.1 * 3 == 3.3
print 1.1 * 4 == 4.4
print 1.1 * 5 == 5.5
print 1.1 * 6 == 6.6
print 1.1 * 7 == 7.7
print 1.1 * 8 == 8.8
print 1.1 * 9 == 9.9
print 1.1 *10 == 11
```

```
print 1.1 * 0 == 0.0      #  True
print 1.1 * 1 == 1.1      #  True
print 1.1 * 2 == 2.2      #  True
print 1.1 * 3 == 3.3      #  False
print 1.1 * 4 == 4.4      #  True
print 1.1 * 5 == 5.5      #  True
print 1.1 * 6 == 6.6      #  False
print 1.1 * 7 == 7.7      #  False
print 1.1 * 8 == 8.8      #  True
print 1.1 * 9 == 9.9      #  True
print 1.1 *10 == 11       #  True
```

You never compare two floating-point numbers for equality!

```python
x = 0.0
while x < big_number:
    ... # x is unchanged in here!
    x += 1.
```

Can this go into an infinite loop?

```python
x = 0.0
while x < big_number:
    ... # x is unchanged in here!
    x += 1.
```

Can this go into an infinite loop?

Yes, it can!

Overflow & Underflow

When numbers are too big, we say they overflow.
When they are too small, we say they underflow.

Catastrophic Cancellation

$$\lim_{x \to 0} \frac{1 - \cos x}{x^2}$$

(Example from "Introduction to Programming in Java")

Catastrophic Cancellation

Be Careful

- Use existing implementations of algorithms instead of rolling your own.
- Don't trust your instincts.

Some Special Numbers

- -0: minus zero.
- $\pm\infty$
- NaN: Not a Number

NaN

```python
A = float('NaN')
print A == A
```

prints False!!