Numerical Representations

Luis Pedro Coelho

Programming for Scientists

October 22, 2012
It’s all 0s & 1s. How do you represent 123?
Binary Notation

\[(b_4b_3b_2b_1b_0)_2 = b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0 = 16b_4 + 8b_3 + 4b_2 + 2b_1 + b_0\]
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
```python
def fact(N):
    if N == 0: return 1
    return N * fact(N-1)

print fact(100)
```

Prints out

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000L
```
What about negative numbers?
- Sign bit
- Biasing
- Ones’ complement
- Twos’ complement
Sign Bit

\[(s_b_4 b_3 b_2 b_1 b_0)_2 = (-1)^s (b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0)\]
Biasing

Have a bias $B$, so that the number $n$ is represented as unsigned$(n + B)$. 
One’s Complement

If \((b_k b_{k-1} \cdots b_1 b_0)_2\) is some number \(n\), then we represent \(-n\) by 
\((\overline{b_k} \overline{b_{k-1}} \cdots \overline{b_1} \overline{b_0})_2\) 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 

\[(\sim b_k \sim b_{k-1} \cdots \sim b_1 \sim b_0)_2\] is some number \(n\), then we represent \(-n\) by

\[
\begin{align*}
(00000011)_2 & \text{ is 3} \\
(11111100)_2 & \text{ is } -3
\end{align*}
\]

\[
\begin{align*}
(00001111)_2 & \text{ is 31} \\
(11110000)_2 & \text{ is } -31
\end{align*}
\]

Ones’ complement is not actually used in any modern machine.
One’s Complement

If \((b_k b_{k-1} \cdots b_1 b_0)_2\) is some number \(n\), then we represent \(-n\) by

\((\overline{b_k} \overline{b_{k-1}} \cdots \overline{b_1} \overline{b_0})_2\) 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 $(\overline{b_k b_{k-1} \cdots b_1 b_0})_2$ 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

\[(11111111)_2 \text{ is } -1\]
\[(11111110)_2 \text{ is } -2\]
\[(11111101)_2 \text{ is } -3\]
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
Given a fixed base B, then an integer n really represents the number \( n \times 2^B \).
Floating Point

602214179303030303030303030303
Floating Point

60221417930303030303030303030303
6.022 \times 10^{23}
Floating Point Representation

\[ (-1)^s \times c \times 2^q \]
IEEE-754

Numerical Representations

October 22, 2012 (19 / 32)
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 $\pm 1.8 \times 10^{308}$. 
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!
```
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!
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”)
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
$\text{NaN}$

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

print A == A
```

prints `False`!!