Numbers

Arnaud Malapert, Gilles Menez, Marie Pelleau

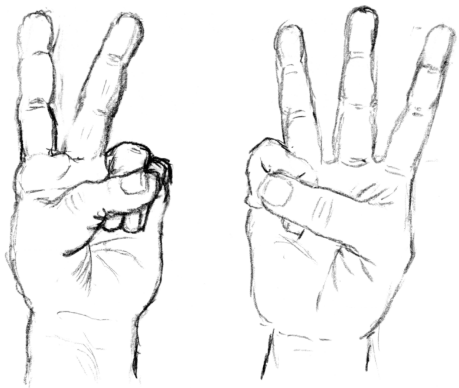
Master Informatique, Université Côte d'Azur

Computers and Data abstraction

- An old history
- Abstraction remains abstractions
- How to make by itself?

Counting

Soon after language develops, it is safe to assume that humans begin counting to represent amount of cows, volume of wine, area of fields, ...
It is safe to assume too that fingers and thumbs provide nature's abacus



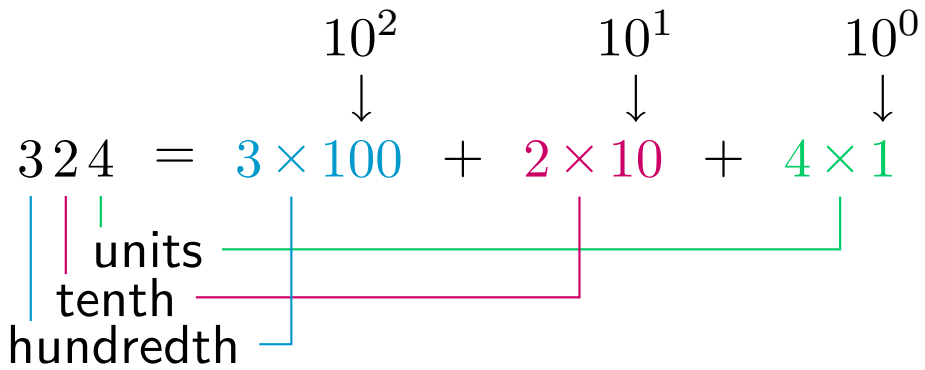
Remarque

- The decimal and duodecimal systems are no accidents, ten and twelve have been the bases of most counting systems in history
- Abacus from greek "dust table" is the generic name given to "planar mechanical instrument" for computation and counting

Modern numbers

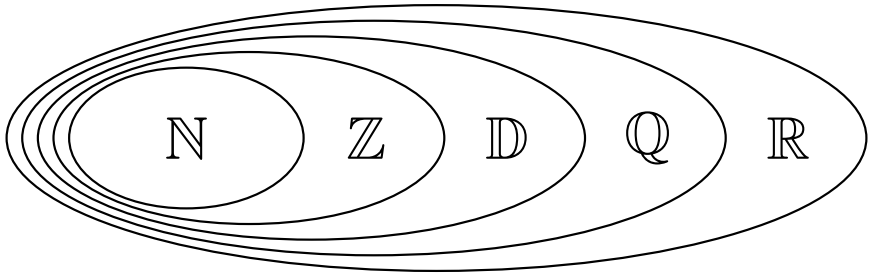
Arab numbers (and 10 basis digit system) are a modern representation (4th century)

	0	1	2	3	4	5	6	7	8	9
Arabic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Devanagari	०	१	२	३	४	५	६	७	८	९
Bengali	০	১	২	৩	৪	৫	৬	৭	৮	৯
Gurmukhi	੦	੧	੨	੩	੪	੫	੬	੭	੮	੯

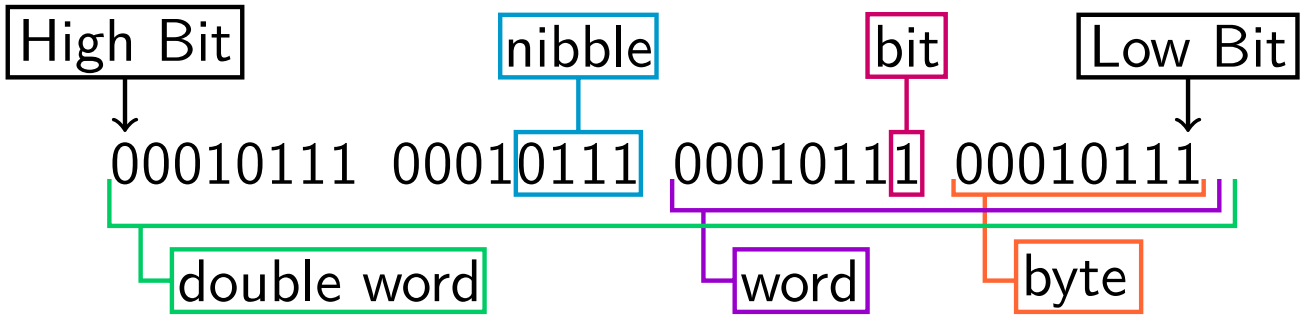


Data words

Machine numbers are abstractions of mathematical numbers :



They are supported by a bit, a group/words (finite) of bits whose logical nature is compatible with electronic processing (on-off)



Representation is finite, we can only represent a subset of values : 2^n

Coding numbers

Most of machines will offer two well known coding methods

Two's Complement for signed integers

$$x = -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{i<n-1} b_i \times 2^i$$

$$12_{10} - 5_{10} = 12_{10} + (-5_{10})$$

$$\begin{array}{rll} & 0000\ 1100 & (12_{10}) \\ + & 1111\ 1011 & (5_{10}) \\ \hline = & 0000\ 0111 & (7_{10}) \end{array}$$

$$-2^{n-1}$$

$$\rightarrow$$

$$2^{n-1} - 1$$

IEEE 754 for floats

$$x = (-1)^s \times 1.m \times 2^{(e-bias)}$$

- s : Sign bit (1 bit)
- m : Mantissa (23 bits)
- e : Exponent (8 bits)

$$1,175\ 494\ 35 \times 10^{-38}$$

$$\rightarrow$$

$$3,402\ 823\ 46 \times 10^{38}$$

The range problem

- $2^{32} = 4,294 \times 10^9$
- $2^{64} = 1,844 \times 10^{19}$
- $2^{128} = 3,402 \times 10^{38}$

- $12! = 4,790 \times 10^8$
- $13! = 6,227 \times 10^9$
- $20! = 2,432 \times 10^{18}$
- $21! = 5,109 \times 10^{19}$
- $34! = 2,952 \times 10^{38}$
- $35! = 1,033 \times 10^{40}$

- Programmer must guarantee that the integer in a specific application do not cause an overflow
- More and more modern applications, need to get off the range of the machine and/or of the language
  - Estimated number of atoms in the observable universe 10^{80}
  - Earth's mass consist of about 4×10^{51} nucleons
  - Estimated number of neuronal connections in the human brain 10^{14}
  - Estimated lower bound on the game-tree complexity of chess ("Shannon number") 10^{120}

Big numbers

- Several modern programming languages have built-in or options to support for bignums : Lisp, Python, Perl, Haskell and Ruby
- Others (C, C++, Java) have libraries available for arbitrary-precision integer and floating-point math

Rather than store values as a fixed number of binary bits related to the size of the processor register, these implementations typically use variable-length arrays of digits

- Reduces performances
- Eliminates simple overflow
- Guarantees the results on all machines

Exercise 1: Big addition

Statement

- Addition of large numbers (unsigned integers) stored in strings
- Numbers may be very large (may not fit in `long long int`)
- Reads (from stdin) sequences of two strings containing big numbers
- Computes the sum and writes (on stdout) the string of the resulting number
- If one of the two input strings is not a number (*i.e.* all letters of the string are not digits or the string is empty) output should be the string "?"

Exercise 1: Big addition

Solution 1: Using languages with bignums

read s_1 and s_2 on the standard input

print $s_1 + s_2$

Solution 2: Using a language **without** bignums

read s_1 and s_2 on the standard input

res $\leftarrow$ computes addition of s_1 and s_2

print res

Exercise 1: Big addition

Example

Input:

```
333331111111111
44422222221111

4
0005682
0529
L33T
12
0538
2
```

Output:

```
3377733333332222
?
6211
?
540
```

Exercise 2: Bit Reverse

Statement

Given an hexadecimal integer x and a decimal integer n, reverse the nth lowest bit of x

$n = 3$

b_{31}	$\dots$	b_3	000	becomes	b_{31}	$\dots$	b_3 000
b_{31}	$\dots$	b_3	001	becomes	b_{31}	$\dots$	b_3 100
b_{31}	$\dots$	b_3	010	becomes	b_{31}	$\dots$	b_3 010
b_{31}	$\dots$	b_3	011	becomes	b_{31}	$\dots$	b_3 110
b_{31}	$\dots$	b_3	100	becomes	b_{31}	$\dots$	b_3 001
b_{31}	$\dots$	b_3	101	becomes	b_{31}	$\dots$	b_3 101
b_{31}	$\dots$	b_3	110	becomes	b_{31}	$\dots$	b_3 011
b_{31}	$\dots$	b_3	111	becomes	b_{31}	$\dots$	b_3 111

$\Rightarrow$ if $n \notin [2, 32]$ x is not modified