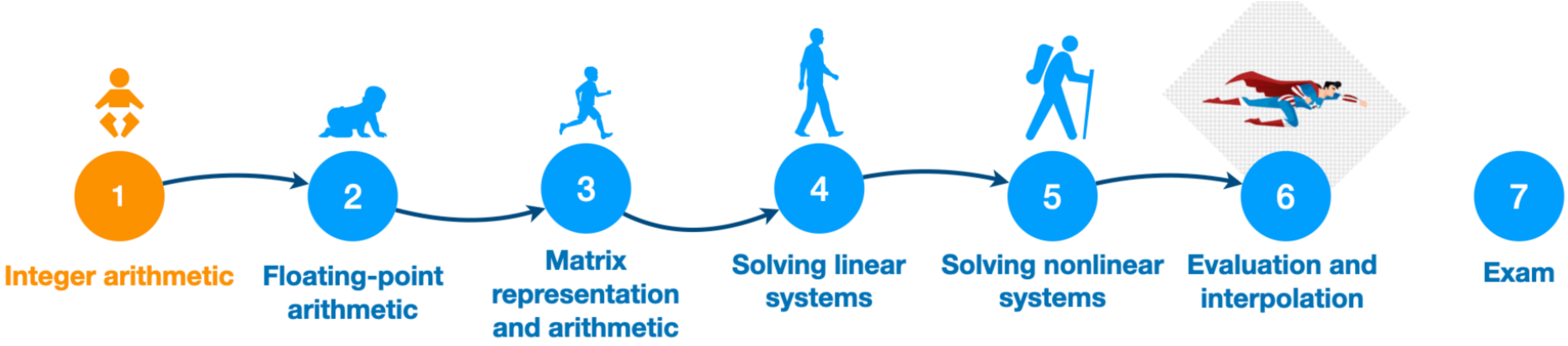
Python & algorithm workshop

Lecture 1 : Integer arithmetic

April 2022 - François HU
https://curiousml.github.io/

Outline


1
Integer arithmetic
2
Floating-point arithmetic
3
Matrix representation and arithmetic
4
Solving linear systems
5
Solving nonlinear systems
6
Evaluation and interpolation
7
Exam

0 Introduction

Introduction

Context

Algorithms

Objective : compute quick and precise algorithms

- Numerically **speed** up the computation process

Representations

Gap between ***mathematical*** and ***computer*** representations of numbers

- May lead to approximation **errors**
- **Objective** : mitigate these errors

Introduction

Context

Algorithms

Objective : compute quick and precise algorithms

- Numerically **speed** up the computation process

Representations

Gap between ***mathematical*** and ***computer*** representations of numbers

- May lead to approximation **errors**
- **Objective** : mitigate these errors

Example 1

Ariane flight V88 : Exploded on June 4, 1996 just after lift-off due to the consequence of an **overflow**.

Example 2

The Patriot Missile : Failed on February 25, 1991 which resulted in 28 deaths
due to poor handling of **rounding errors**.

Introduction

Context

Algorithms

Objective : compute quick and precise algorithms

- Numerically **speed** up the computation process

Representations

Gap between ***mathematical*** and ***computer*** representations of numbers

- May lead to approximation **errors**
- **Objective** : mitigate these errors

Example 1

Ariane flight V88 : Exploded on June 4, 1996 just after lift-off due to the consequence of an **overflow**.

Example 2

The Patriot Missile : Failed on February 25, 1991 which resulted in 28 deaths due to poor handling of **rounding errors**.

Main goal : tradeoff between **precision, range** and **speed.**

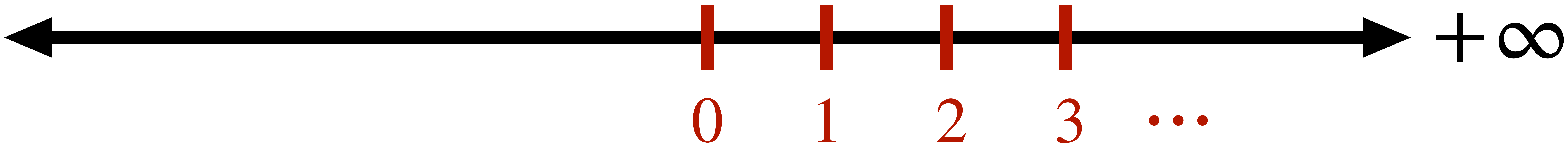
Introduction

Mathematical representation

Maths

$\mathbb{N}$

$\mathbb{Z}$



$\mathbb{Q}$ A rational number includes any whole number, fraction, or decimal that ends or repeats

$\mathbb{R}$ An irrational number is any number that cannot be turned into fraction

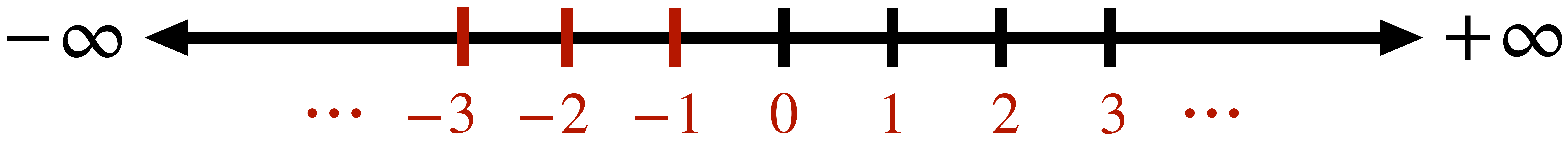
Introduction

Mathematical representation

Maths

$\mathbb{N}$

$\mathbb{Z}$



$\mathbb{Q}$ A rational number includes any whole number, fraction, or decimal that ends or repeats

$\mathbb{R}$ An irrational number is any number that cannot be turned into fraction

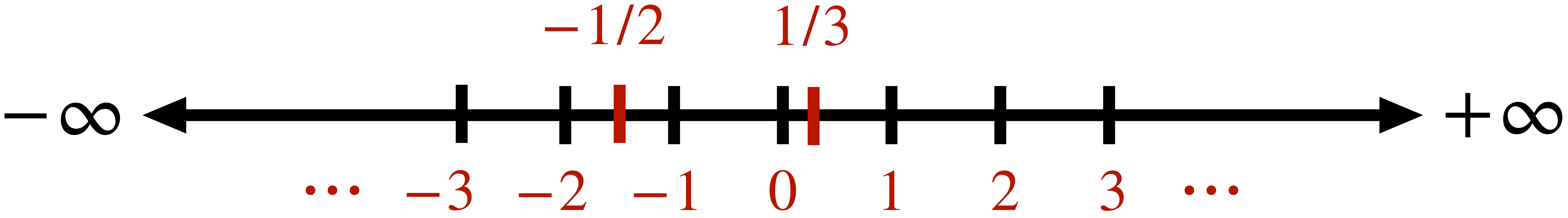
Introduction

Mathematical representation

Maths

$\mathbb{N}$

$\mathbb{Z}$



$\mathbb{Q}$ A rational number includes any whole number, fraction, or decimal that ends or repeats

$\mathbb{R}$

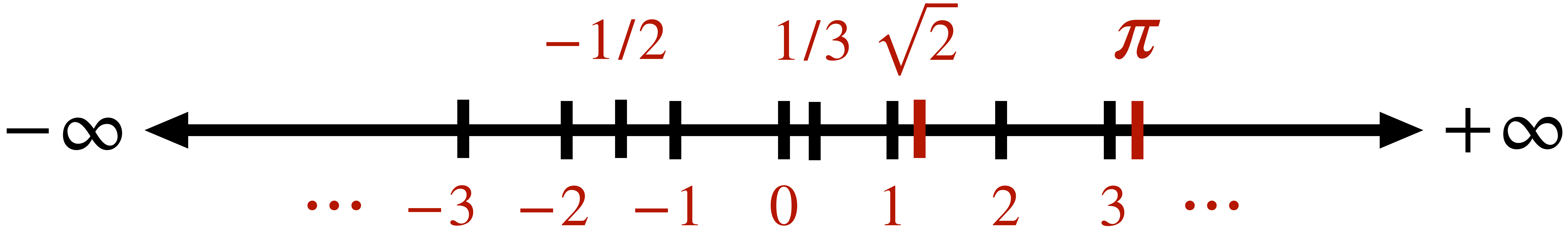
Introduction

Computer representation

Maths

$\mathbb{N}$

$\mathbb{Z}$



$\mathbb{Q}$ A rational number includes any whole number, fraction, or decimal that ends or repeats

$\mathbb{R}$ An irrational number is any number that cannot be turned into fraction

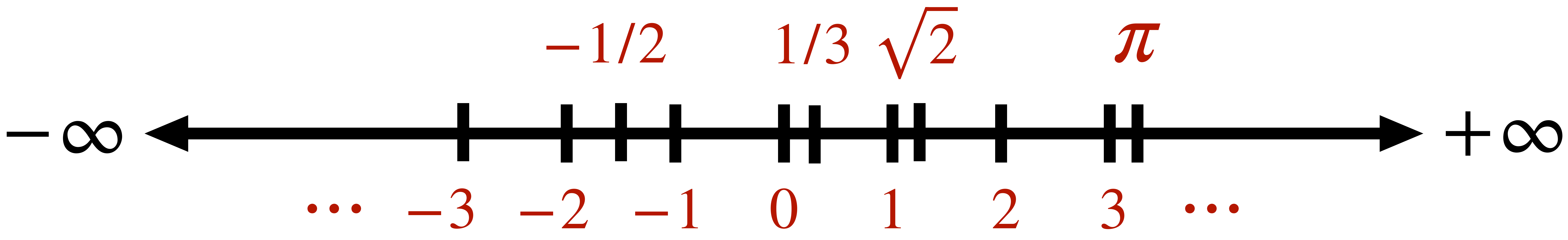
Introduction

Computer representation

Maths

$\mathbb{N}$

$\mathbb{Z}$



$\mathbb{Q}$ A rational number includes any whole number, fraction, or decimal that ends or repeats

$\mathbb{R}$ An irrational number is any number that cannot be turned into fraction

Lecture 1 topics
- **integer representation**
- **integer arithmetic**

1 Integer representation

1. Integer representation

Base

(Positional) integer representation

Choose a **base (or radix)** β and an **alphabet** $\mathscr{A} = \{0,1,2,\ldots,\beta-1\}$

The number n is written as $\left(n_k\ldots n_2 n_1 n_0\right)_\beta = \left(n_0\beta^0 + n_1\beta^1 + \ldots + n_k\beta^k\right)_{10} = \left(\sum_{i=0}^{k} n_i\beta^i\right)_{10}$ with $n_i \in \mathscr{A}$

Example: base 2

$\mathscr{A} = \{0,1\}$

$1010001 = 1\times 2^6 + 1\times 2^4 + 1$

Example: base 10

$\mathscr{A} = \{0,1,\ldots,9\}$

$2271 = 2\times 10^3 + 2\times 10^2 + 7\times 10 + 1$

Example: base 16

$\mathscr{A} = \{0,1,\ldots,9,A,B,C,D,E,F\}$

$E20F = \cdots$

Euclidean division based algorithm: representation of an integer n in base β

input : n
$k \leftarrow 0$
while $n > 0$ do :
    $n_k \leftarrow n \,\%\, \beta$
    $n \leftarrow n/\beta$
    $k \leftarrow k+1$
output : $(n_k\ldots n_2 n_1 n_0)_\beta$

Euclidean division

if (a,d) are integers then there exists integers (q,r) such that

$$a = d\times q + r \quad \text{with} \quad 0 \le r < |d|$$

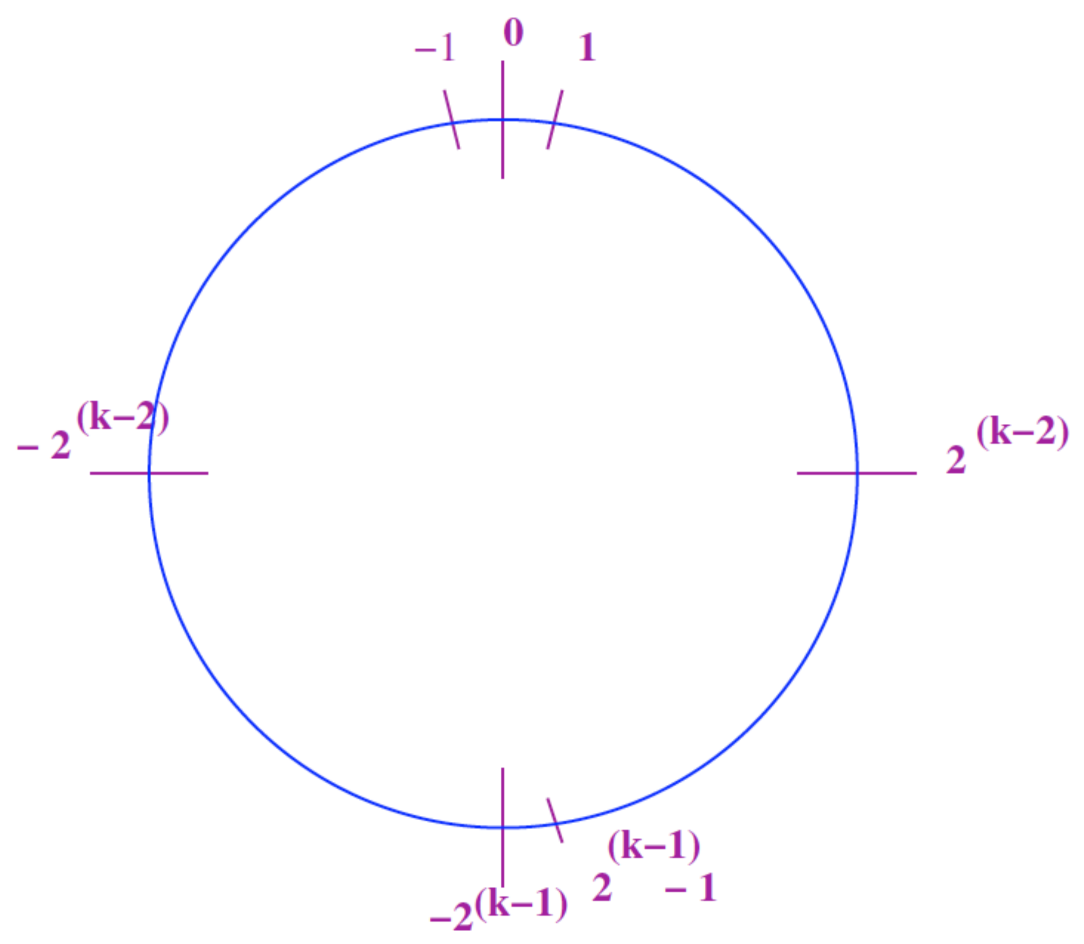
we denote :

- $q = a/d$ the **quotient**
- and $r = a \,\%\, d$ the **remainder**

1. Integer representation

Binary digit (bit)

Bit : for a computer, numbers are represented in base 2 and each binary digit is called « **bit** »



Programming languages such as Java or C/C++ :

- An integer is coded on a fixed number of k bits (usually $k = 32$ or $k = 64$)
- So only integers smaller than 2^k are represented
- **Remark :** for these programming languages the arithmetic is <u>modular</u> (mod 2^k) !

For Python :

- **No integer overflow** : Python uses a **variable** (not fixed !) number of bits to represent integers
- **Python integers are objects** $\Longrightarrow$ need for additional fixed number of bits
- The maximum integer representation depends on the memory available

```python
from sys import getsizeof

n = 1024
size = getsizeof(n)
print(size)  # 28 bytes, so 28*8 bits

n = 2**64
size = getsizeof(n)
print(size)  # 36 bytes, so 36*8 bits

n = 2**288
size = getsizeof(n)
print(size)  # 64 bytes, so 64*8 bits
```

```
28
36
64
```

2 Integer addition

2. Integer addition

Naïve addition (works only for an arbitrary-precision integers)

Integer addition algorithm:

input : $A = (a_{k-1}\dots a_1 a_0)_\beta$,

$B = (b_{k-1}\dots b_1 b_0)_\beta$

$c \leftarrow 0$

for $i = 0$ to $k-1$ do :

  $s_i \leftarrow a_i + b_i + c$

  if $s_i \geq \beta$ then :

    $c \leftarrow 1$

    $s_i \leftarrow s_i - \beta$

  else $c \leftarrow 0$

$s_k \leftarrow c$

output : $S = (s_k\dots s_2 s_1 s_0)_\beta$

$$\begin{array}{cr} & 00001100 \\ + & 01001001 \\ \hline & 01010101 \end{array} \qquad \begin{array}{cr} & 10001100 \\ + & 01001001 \\ \hline & 11010101 \end{array} \qquad \begin{array}{cr} & 01001100 \\ + & 01001001 \\ \hline & 10010101 \end{array}$$

$$\begin{array}{cr} & 10001100 \\ + & 11001001 \\ \hline 1 & 01010101 \end{array} \qquad \begin{array}{cr} & 10001100 \\ + & 11110100 \\ \hline 1 & 10000000 \end{array}$$

3 Integer multiplication

3. Integer multiplication

Long (or grade-school) multiplication

Long multiplication algorithm:

Used in Python for **small** numbers

input : $A = (a_{k-1} \ldots a_1 a_0)_\beta$,

$B = (b_{k-1} \ldots b_1 b_0)_\beta$

$P \leftarrow 0$

for $i = 0$ to $k - 1$ do :

  $T \leftarrow 0$

  for $j = 0$ to $k - 1$ do :

    $T \leftarrow T + a_j \times b_i \times \beta^{i+j}$

  $P \leftarrow P + T$

output : $P = (p_{2k-1} \ldots p_2 p_1 p_0)_\beta$

```
      1010
   ×  1011
   -------
      1010
     1010
    0000
   1010
   -------
   1101110
   -------
```

- The multiplication by β^{i+j} is a shift of $i + j$ positions to the left (adding $i + j$ zeros at the right of the integer)
- **Algorithm complexity : quadratic** with the size of positions $o(k^2)$

3. Integer multiplication

Divide and conquer

Karatsuba algorithm:

Used in Python for **big** numbers

input : $A = (a_{k-1}\ldots a_1 a_0)_\beta$,
$B = (b_{k-1}\ldots b_1 b_0)_\beta$

if $k = 1$ then :
$P \leftarrow a_0 \times b_0$

else :
$k_0 \leftarrow \lfloor k/2 \rfloor$ and $k_1 \leftarrow k - k_0$
$A_1 \leftarrow a_{k-1}\ldots a_{k_0}$ and $A_0 \leftarrow a_{k_0-1}\ldots a_0$
$B_1 \leftarrow b_{k-1}\ldots b_{k_0}$ and $B_0 \leftarrow b_{k_0-1}\ldots b_0$
$Sa \leftarrow 1$ and $Sb \leftarrow 1$
if $A_0 \geq A_1$ then $D \leftarrow A_0 - A_1$
else $D \leftarrow A_1 - A_0$ and $Sa \leftarrow -1$
if $B_0 \geq B_1$ then $E \leftarrow B_0 - B_1$ and $Sb \leftarrow -1$
else $E \leftarrow B_1 - B_0$
$T \leftarrow$ Karatsuba(A_1, B_1, k_1)
$U \leftarrow$ Karatsuba(A_0, B_0, k_0)
$V \leftarrow$ Karatsuba(D, E, k_1)
$V \leftarrow (Sa \times Sb) \times V + T + U$
$P \leftarrow T \times \beta^k + V \times \beta^{k/2} + U$

output : $P = (p_{2k-1}\ldots p_2 p_1 p_0)_\beta$

1. Multiplication « **divide and conquer** »

Assume that $k = 2^t$
we can write
$$A = A_1 \beta^{k/2} + A_0$$
$$B = B_1 \beta^{k/2} + B_0$$
Therefore
$$A \times B = (A_1 \times B_1)\beta^k + (A_1 \times B_0 + A_0 \times B_1)\beta^{k/2} + A_0 \times B_0$$
However
$$(A_1 \times B_0 + A_0 \times B_1) = (A_1 - A_0)(B_0 - B_1) + (A_1 \times B_1) + A_0 \times B_0$$
$\Rightarrow$ instead of 1 mult of k bits numbers we have 3 mult of $k/2$ bits numbers

2. **Algorithm complexity :** $o(k^{\log_2(3)}) \approx o(k^{1.585})$

Let us denote $T(k)$ the complexity
$$T(k) = 3 \times T(k/2) + \alpha k \quad (\alpha k \text{ additions complexity})$$
$$3 \times T(k/2) = 3^2 T(k/4) + 3\alpha k/2$$
$$\vdots$$
$$3^{t-1} T(k/2^{t-1}) = 3^t T(k/2^t) + 3^{t-1} \alpha k / 2^{t-1}$$
Therefore
$$T(k) = 3^t T(1) + \alpha k \frac{(3/2)^t - 1}{3/2 - 1}$$
$$= 3^t T(1) + 2\alpha (3^t - k) \quad \text{because } k = 2^t$$
$$= k^{\log_2(3)} T(1) + 2\alpha k^{\log_2(3)} - 2\alpha k \quad \text{because } 3^t = k^{\log_2(3)}$$
$\Rightarrow o(k^{\log_2(3)})$

4 Bitwise operations

4. Bitwise operations

Masks and shifts

Masks:
- ‘&’ (*bitwise and*),
- ’|’ (*bitwise or*),
- ’^’ (*bitwise xor*)

  can be used to perform Boolean logic on individual bits

$$\begin{array}{rcrcl} a & = & 42 & = & 00101010 \\ b & = & 22 & = & 00010110 \\ \hline a \ \& \ b & = & 2 & = & 00000010 \end{array}$$

$$\begin{array}{rcrcl} a & = & 42 & = & 00101010 \\ b & = & 22 & = & 00010110 \\ \hline a \mid b & = & 62 & = & 00111110 \end{array}$$

Shifts:
- ‘>>’ (*bitwise right shift*) shifts the bits to the right by the number of places provided
- ‘<<’ (*bitwise left shift*) shifts the bits to the left by the number of places provided

  they are commonly used to **boost the speed** of specific mathematical procedures. Can be used to **multiply** or to **divide** the first operand by two at the power of the second operand.

$$\begin{array}{rcrcl} a & = & 68 & = & 01000100 \\ \hline a >> 2 & = & 17 & = & 00010001 \end{array}$$

$$\begin{array}{rcrcl} a & = & 42 & = & 00101010 \\ \hline a << 2 & = & 168 & = & 10101000 \end{array}$$

Other bitwise boolean operations:
- ‘~’ (*bitwise not*)

  can be used to perform Boolean logic on individual bits