Cours 24 – Codage – Partie 1
Représentation en mémoire des nombres

Dans notre usage de l'informatique, les valeurs sont souvent stockées dans la RAM (sous forme de variables), et lors d'opérations entre plusieurs valeurs, le processeur doit pouvoir les interpréter. Dans ce premier cours, nous nous intéresserons aux variables simples, c'est-à-dire les types : *bool, unsigned int, int, float, double,* et *string*. Essentiellement, ce qui nous intéresse ici est donc le codage des nombres.
Dans le second cours, nous aborderons les types « conteneurs », c'est-à-dire les types ou classes : *list, array* et *dict.*

A / « Bestiaire » des différents types de valeurs numériques

$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{D} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$

Entiers naturels ; Entiers relatifs ; Décimaux ; Rationnels ; Réels ; Complexes

0 ou 12 | -3 | 4,17 | $\frac{1}{3}$ | π ou $\sqrt{2}$ | S'écrit $a + ib$ avec $a, b \in \mathbb{R}$ → s'écrit ≈ avec 2 réels

Pas de signe, pas de virgule | Signé, mais, pas de virgule | Signé + nombre fini de chiffres après la virgule | Signé + ∞ de chiffres après la virgule → impossible à ranger dans une mémoire finie → seront **toujours** approchés par le décimal le + proche

→ Codage naturel (mais ∃ d'autres, cf cours de SII) | → Codage par complément (à 1 ou à 2) | → Codage par nombre à virgule flottante

Type : *unsigned int* (*uint*) ; Type : *int* (*int*) | Type : *float* ou *double*

B / Entiers naturels $\mathbb{N}$

B.1 / Écriture d'un entier naturel en base quelconque

Soit un entier $n \in \mathbb{N}$. L'écriture classique que nous utilisons est la base décimale.

À part pour $n = 0$, on dit que « n est un nombre à M chiffres », ou rigoureusement un nombre à M digits s'il existe $M \geq 1$ tel que $10^{M-1} \leq n \leq 10^M - 1$.

→ On peut alors écrire n en base 10, sous forme de « liste » de M digits, où la valeur de chaque digit $n_i \in [\![0,9]\!]$

$$n = \sum_{i=0}^{M-1} n_{M-1-i} \times 10^i$$

Ceci revient à lire un nombre comme une somme de puissances de 10, de **droite à gauche**.
Exemple :

Prenons le nombre : 3 (× 10^3) 0 (× 10^2) 9 (× 10^1) 1 (× 10^0)

$= 3000 + 00 + 90 + 1$ (0 est le neutre)

Étant donné le support physique de la RAM, la base classique pour un ordinateur actuel est dit binaire (2).

À part pour $n = 0$, on dit que « n est un nombre à M chiffres », ou rigoureusement un nombre à M bits s'il existe :

$$M \geq 1 \text{ tel que } 2^{M-1} \leq n \leq 2^M - 1$$

→ On peut alors écrire n en base 2, sous forme de « liste » de M bits, où la valeur de chaque digit $n_i \in \{0,1\}$

$$n = \sum_{i=0}^{M-1} n_{M-1-i} \times 2^i$$

Ceci revient à lire un nombre comme une somme de puissances de 2, de **droite à gauche**.

Exemple :

Prenons le nombre binaire : 1 0 0 1 0 1 0 1

× 128 × 64 × 32 × 16 × 2^3=8 × 2^2=4 × 2^1=2 × 2^0=1

Il vaut donc, en décimal : 128 + 0 + 0 + 16 + 0 + 4 + 0 + 1 = 149

Remarque 1 : On appelle **octet** un nombre binaire, qui est codé sur 8 bits

→ Sur un octet, on peut coder des nombres entiers compris entre 0000 0000 et 1111 1111, càd entre 0 et 255.

Remarque 2 : [*Ruse classique de vendeur*] En anglais, on appelle *byte* un octet, et *bit* un bit.

→ un débit de données de 1 Mbits.s^{-1} correspond à 1/8 Mbytes.s^{-1} = 125 ko.s^{-1}, ($1\ Méga = 10^6$).

Pour représenter des nombres entiers n grands, plutôt que la base décimale on préfèrera la base hexadécimale (16).

À part pour $n = 0$, on dit que n est un nombre à M caractères (ou M digits) s'il existe $M \geq 1$ tel que

$$16^{M-1} \leq n \leq 16^M - 1$$

→ On peut alors écrire n en base 16, sous forme de « liste » de M digits, où la valeur de chaque digit $n_i \in [\![0,15]\!]$

$$n = \sum_{i=0}^{M-1} n_{M-1-i} \times 16^i$$

Pour afficher la valeur des digits, nous n'avons que des chiffres allant de 0 à 9 en base 10, alors qu'il faut aller jusqu'à 15. Par convention, on pose donc : (pour 10 , A); (pour 11 , B); (pour 12 , C); (pour 13 , D); (pour 14 , E); (pour 15 , F).

On va donc lire un nombre comme une somme de puissances de 16, de **droite à gauche**.

Exemple :

Prenons le nombre hexa : A 1 F 7

× 16^3=4096 × 16^2=256 × 16^1=16 × 16^0=1

Il vaut donc, en décimal : $10 \times 4096 + 1 \times 256 + 15 \times 16 + 7 = 41\,463$

Remarque : Pour coder la valeur d'un caractère hexadécimal (entre 0 et 15) en binaire, il faut 4 bits

→ La correspondance entre binaire et hexadécimal est donc facile à faire (c'est pourquoi l'hexadécimal est utilisé)

Exemple prenons le mot binaire : 1 1 0 1 | 0 1 1 1 | 1 1 1 0 | 0 0 1 0 (bits regroupés 4 par 4)

(13) (7) (14) (2) (si besoin, en décimal)

Converti en hexadécimal : D 7 F 2

B.2 / Écriture d'un entier naturel en base 2

Pour convertir un entier naturel depuis la base décimale à la base binaire, je vous propose la méthodologie suivante :

B.2.i / Détermination du nombre de bits nécessaires *(parfois facultatif)*

Si le nombre de bits n'est pas imposé (par le type de processeur ou de mémoire, par exemple : 8, 16, 32 ou 64 bits), on peut avoir la liberté de choisir le nombre de bits nécessaires pour écrire un entier.

La valeur la plus élevée pouvant être écrite sur *N* bits est :

$$\sum_{i=0}^{N-1} 2^i = \frac{1-2^N}{1-2} = 2^N - 1$$

(suite géo-métrique de raison $q = 2$).

En d'autres termes, *N* bits permettent d'écrire une valeur binaire comprise entre 0 et $2^N - 1$

Ainsi, pour exprimer une valeur entière *K*, on recherche le plus petit entier *N* tel que $2^N - 1 \geq K$

Exemples : pour $K_1 = 148$, combien de bits sont nécessaires (au minimum) ?

Avec 7 bits, on peut aller jusqu'à 128 – 1 = 127 (insuffisant).
Avec 8 bits, on peut aller jusqu'à 256 – 1 = 255 → il faut 8 bits au minimum.

Même question pour $K_2 = 16$

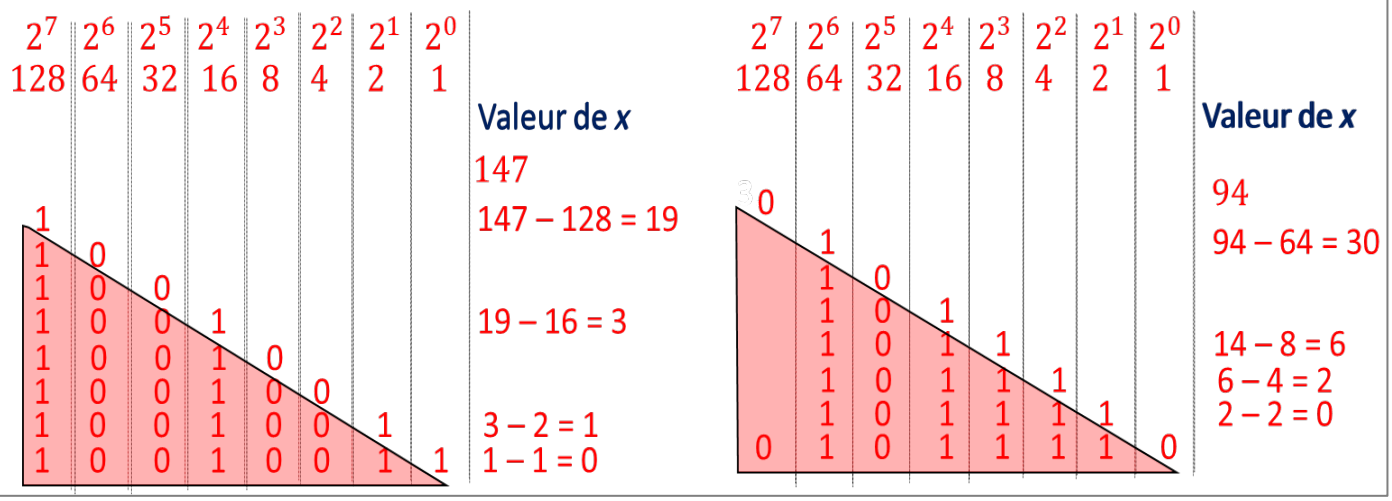
Il faut 5 bits (valeurs comprises entre 0 et 31).

B.2.ii / Écriture du nombre *K* avec le nombre de bits déterminé *(ou avec le nombre de bits imposé)*

Algorithme conseillé (la base binaire étant canonique, vous retrouverez ici un algorithme de type glouton, très similaire à celui découvert sur le rendu de monnaie) :

- Écrire la suite des 2^i de **droite** à **gauche** : 2^{N-1} ; 2^{N-2} ; … ; 4 ; 2 ; 1 → ceci crée *N* colonnes
- Définissons un reste *r* , initialisé à *K*, qui déterminera « combien il reste à écrire »
- Puis, en balayant les colonnes de **gauche** à **droite** (ou mieux : tant que le reste *r* est non nul)
  - Si $r < 2^{i-1}$ *(on ne peut pas « prendre » la valeur 2^{i-1})*
    - On inscrit 0 dans la colonne correspondante
  - Sinon *(on peut « prendre » la valeur 2^{i-1})*
    - On inscrit 1 dans la colonne correspondante
    - Le nouveau reste à écrire est $r \leftarrow r - 2^{i-1}$

Exemples : écrire, sur 8 bits, les valeurs 147 et 94.



B.3 / Opérations en binaire

Les deux seules opérations possibles entre nombres binaires sont l'addition et la multiplication.

B.3.i / Addition en binaire

L'addition se fait comme ce que vous faisiez au primaire, mais il faut garder en tête que les bits valent 0 ou 1, ils ne peuvent donc pas valoir 2 ni 3 : 1 + 1 = 10, et 1 + 1 + 1 = 11 en binaire. Les retenues sont donc très courantes.

Exemples :

```
  1                    Vérif :
  1 0 1                  5
+ 1 1 0                + 6
-------                ----
1 0 1 1                = 11  ✓

  1 1                  Vérif :
  1 1 1 0                14
+   1 1 0              + 6
---------              ----
1 0 1 0 0              = 20  ✓
```

B.3.ii / Multiplication en binaire (limite programme)

Si une valeur entière positive K s'écrit en base 2 : $\sum_i a_i 2^i$, alors $2 \times K$ s'écrit $\sum_i a_i 2^{i+1}$

Exemple : 5 s'écrit : 0 0 0 1 0 1 et 10 s'écrit : 0 0 1 0 1 0 et 20 s'écrit : 0 1 0 1 0 0 et 40 s'écrit : 1 0 1 0 0 0

Ainsi : multiplier un nombre binaire par 2 revient à décaler tous les bits de 1 vers la gauche

Poursuivons : multiplier K par 2^j donne $\sum_i a_i 2^{i+j}$

Ainsi : multiplier un nombre binaire par 2^j revient à décaler tous les bits de j vers la gauche

Conclusion : étant donné que tout autre entier naturel se décompose en base 2 : $K' = \sum_j b_j 2^j$, le produit $K \times K'$ se calcule en distribuant (formule du binôme faisant apparaître une somme) et on voit qu'une multiplication est une combinaison de décalages de bits à gauche et d'additions. Avantage : temps de calcul très court pour le processeur !

B.4 / Écriture d'un entier naturel en base hexadécimale

Notons que $16^1 = 2^4$ ce qui rend très facile le passage de base binaire à base hexadécimale (1 hexa = 4 bits). On fait donc un passage décimal ↔ binaire ↔ hexadécimal.

Exemple : 1011 0101 1110 0010 (2 octets)

8+2+1 4+1 2+4+8 2

→ en hexa : B 5 E 2

ou en sens inverse FA (250) en hexa

→ en binaire : 15=8+4+2+1 10=8+2

1111 1010

C / Entiers relatifs $\mathbb{Z}$

Pour pouvoir coder des nombres entiers négatifs en binaire, une idée naïve est de simplement séparer les *N* bits en :

- Le 1er bit (le plus à gauche, dit « **bit de poids fort** » - BPF) : qui sert uniquement à exprimer le signe : -1^{BPF}
  - Ainsi, si le 1er bit vaut 0, l'entier codé est positif
  - Sinon (si le 1er bit vaut 1), l'entier codé est négatif.
- Les $N - 1$ autres bits servent alors à coder la valeur absolue, qui est forcément positive et peut donc être codée en binaire naturel.

Ainsi, avec cette représentation, sur 8 bits, on peut coder des nombres compris entre -127 et 127 (car 7 bits permettent de coder des nombres entre 0 et 128 – 1).

Exemple : 1 0 0 1 0 1 0 1

- (16 + 4 + 1) = -21

ou, dans le sens inverse : + 15 et – 3 sur 8 bits

0 0001111 et 1 0000011

Mais alors, quel est le « problème » avec cette représentation ?

Le nombre 0 peut être codé par +0 et par -0. Exemple, sur 4 bits : 1000 et 0000 codent la même valeur.
Or, on souhaite l'unicité d'écriture d'une valeur dans une base (pour pouvoir établir des bijections).
→ En pratique, cette représentation **n'est pas utilisée**, et on lui préfère les 2 représentations suivantes.

C.1 / Écriture en binaire « complément à 2 »

Outre le fait qu'elle règle le problème d'unicité de la représentation de la valeur 0, l'écriture en **complément à 2** présente aussi l'avantage de simplifier les calculs de soustraction, en les ramenant à des additions en binaire.

Soit une valeur *K* et un nombre de bits *N* « suffisamment grand » (nous verrons ce que ça signifie plus bas). L'algorithme pour écrire *K* en binaire complément à deux, sur *N* bits est le suivant :

- Si $K \geq 0$
  - Écrire K en binaire naturel, normalement
  - (Remarque : le 1er bit sera alors nécessairement = 0, ce qui est cohérent car la valeur codée est positive)

- Sinon $K < 0$
  - Écrire $|K+1|$ (ou, c'est équivalent, $|K|-1$) en binaire naturel
  - Faire le complémentaire de chacun des bits ! **Exemple :** $0\,1\,0\,1\,1 \rightarrow 1\,0\,1\,0\,0$
  - (Remarque : le 1er bit sera alors nécessairement = 1, ce qui est cohérent car la valeur codée est négative)

Exemple : en remarquant que $89 = 64 + 16 + 8 + 1$, écrire -89 en binaire complément à 2 sur 8 bits.

- 89 – 1 = 88 = 0 1 0 1 1 0 0 0
- Complémentaire : 1 0 1 0 0 1 1 1

De manière réciproque, l'algorithme permettant d'interpréter en décimal une valeur écrite en complément à 2 est le suivant :

- Si le bit de poids fort (le plus à gauche) vaut 0 (c'est que K est positif)
  - Interpréter K en binaire naturel, normalement

- Sinon (bit de poids fort = 1, donc $K < 0$)
  - Faire le complémentaire de chacun des bits
  - Interpréter ce nombre en binaire naturel
  - Ajouter 1
  - Mettre un signe – devant ce nombre.

Exemple : interpréter en décimal la valeur de 1 1 1 1 1 0 1 0

- Complémentaire : 0 0 0 0 0 1 0 1
- Interpréter : 1 + 4 = 5. Ajouter 1 : 6
- Valeur : -6

Remarque 1 : sur 1 octet, + 0 se code : 0000 0000 et -0 se code: 1111 1111 + 0000 0001 = 1 0000 0000.
Cette dernière valeur est codée sur 9 bits, elle sort de l'octet (ne peut pas être codée), il n'y a plus la perte de valeur.

Remarque 2: toujours sur 1 octet, 0111 1111 code la valeur décimale : $2^6 + 2^5 + \cdots + 2^0 = 2^7 - 1 = 127$
et 1000 0000 code $-(111\,1111 + 1) = -128$

→ **Généralisons :** le complément à 2 permet, sur *N* bits, de coder 2^N valeurs, entre -2^{N-1} et $2^{N-1} - 1$

C.2 / Calcul d'une soustraction à l'aide du complément à 2

On a vu en partie **B.3.i** comment réaliser une addition en binaire entre deux valeurs K_1 et K_2. Lorsqu'on aura une soustraction à faire, on fera $K_1 - K_2 = K_1 + (-K_2)$: il ne reste qu'à écrire $-K_2$ en complément à 2.

Remarque 1 : il faut que les deux nombres K_1 et K_2 soient écrits sur le même nombre de bits. Si ce n'est pas le cas, compléter le plus court des deux avec des 0 à gauche.

Remarque 2 : si les nombres binaires sont initialement écrits en binaire naturel sur *N* bits, il faut ajouter un *N + 1* ième bit à gauche pour le signe.

Exemple : calculer 1010 1111 – 11

- On complète à 8 bits + 1 bit de signe : 0 1010 1111 – 0 0000 0011
- Pour passer le second terme en complément à 2 sur 9 bits :
  - On fait le complément bit à bit : 1 1111 1100
  - On ajoute 1 (ça revient au même que si on avait soustrait 1 avant le complément) : 1 1111 1101
- On pose l'addition :

```
  1 1111 111
  0 1010 1111
+ 1 1111 1101
---------------
 10 1010 1100
```

10ème bit > 8 bits d'expression + 1 bit de signe → ignoré

- Vérifions : $(1010\ 1111)_2$ → $(175)_{10}$; $(11)_2$ → $(3)_{10}$ et $(0\ 1010\ 1100)_2$ → $(172)_{10}$ (bit signe ⊕)

C.3 / Écriture en binaire par excès

Une autre manière de coder des entiers relatifs est l'écriture binaire par excès. Elle est souvent utilisée pour les Convertisseurs Analogiques – Numériques (CAN), lorsque la valeur analogique à coder peut être négative comme positive (exemple : sur un capteur potentiométrique, sur une cellule de charge en sortie du pont de Wheatstone, d'une génératrice tachymétrique, etc). L'avantage est que la conversion est très facile à faire et à interpréter.

L'algorithme permettant de convertir une valeur *K* en binaire par excès sur *N* bits est le suivant :

- On ajoute $2^{N-1} - 1$ à K : de la sorte, la valeur à coder est forcément ≥ 0.
- Et on code cette valeur en binaire naturel.

Exemple : convertir – 87 en binaire par excès sur 8 bits.

- Sur *N* = 8 bits, la valeur « médiane inférieure » est 127
- $-87 + 127 = 40 = 32 + 8$
- Donc 0 0 1 0 1 0 0 0

De façon réciproque, pour interpréter la valeur décimale d'un nombre codé en binaire par excès, on applique :

- On interprète la valeur comme si elle était codée en binaire naturel.
- On soustrait $2^{N-1} - 1$ à cette valeur.

Exemple : interpréter la valeur décimale de 1 0 0 0 0 0 1 1 0 1 0 0 (12 bits)

- Sur *N* = 12 bits, la valeur « médiane inférieure » est 2047
- $2048 + 32 + 16 + 4 = 2100$
- $2100 - 2047 = 53$

Remarque : Sur 1 octet, la valeur la plus basse est codée par : $0000\ 0000$ qui correspond à $0 - 127 = -127$

La valeur la plus haute est codée par : $1111\ 1111$ qui correspond à $255 - 127 = 128$

→ **Généralisons :** le binaire par excès permet de coder des valeurs entre $-2^{N-1} + 1$ et 2^{N-1}

D / Cas particulier des entiers multi-précision

La plupart des ordinateurs que nous utilisons aujourd'hui travaillent avec un processeur en 64 bits. On appelle **entiers machine**, ou **petits entiers** les nombres entiers pouvant être représentés directement sur ces 64 bits, en complément à 2, c'est-à-dire des valeurs *K* telles que :

$$-2^{63} \leq K \leq 2^{63} - 1$$

Pour ordre de grandeur, les valeurs absolues maximales sont de l'ordre de 10^{19}. Pour de tels nombres, l'architecture du processeur permet de réaliser les opérations simples (addition, soustraction, multiplication) avec des instructions dédiées, et ce à coût constant (quelques coups d'horloge). En d'autres termes, le coût d'une addition est, pour un processeur 3 GHz, de l'ordre d'1 ns, quelles que soient les tailles (nombres de bits) des deux petits entiers à additionner.

Il arrive qu'on ait besoin de décrire des valeurs entières qui sortent de cet intervalle. On parle d'**overflow**. *Ci-contre une photo de la plus chère erreur d'overflow (dépassement d'entier)…*


Crash *ARIANE* 501 – 04/06/1996

Python permet de le faire au moyen d'**entiers multi-précision**. L'entier est alors stocké sur un espace mémoire contigu qui prend le nombre de bits nécessaires pour en décrire la valeur. Pour distinguer les entiers multi-précision des petits entiers, *Python* ajoute le suffixe *L* à la fin. Exemple :

```
>>> 2**10    # petit entier
1024
>>> 2**62    # idem
4611686018427387904
>>> 2**63    # entier multiprécision car > 2^63 - 1
9223372036854775808L
>>> 2**100   # idem
1267650600228229401496703205376L
```

L'inconvénient notable est que l'architecture du processeur (et de sa cache) ne lui permet pas de traiter directement ces valeurs. *Python* utilise des sous-programmes pour traiter ces valeurs élevées, mais puisqu'elles sont stockées « comme une liste » sur *N* bits (*N* > 64), le temps d'exécution de la moindre opération est $O(N)$, alors qu'il était en $O(1)$ pour des petits entiers.

E / Codage à virgule flottante (*float*)

E.1 / Pour mieux comprendre, revenons sur l'écriture scientifique (en base décimale)

En base décimale, un nombre décimal (au sens qu'il a un nombre entier de chiffres après la virgule) $x \in \mathbb{D}$ peut s'écrire : $s \cdot (c, m) \cdot 10^e$, ou encore

$$(-1)^s \cdot \left[c + \sum_i m_i 10^{-i} \right] \cdot 10^e$$

- s : est le signe (codé sur 1 bit : 0 si + ; 1 si –)
- c : est la caractéristique (1er chiffre significatif, avant la virgule). c ne peut pas être = 0 , donc $c \in [\![1,9]\!]$
- m : est la mantisse (liste finie de chiffres après la virgule). On peut se représenter que $m \in \mathbb{N}$ (comme si c'était un seul nombre entier)
- e : est l'exposant. $e \in \mathbb{Z}$

Exemples :

x	Écriture scientifique	s	c	m	e	(chiffres significatifs)
– 224	$-\ 2{,}24 \cdot 10^2$	1	2	24	2	(3)
0,03554	$+\ 3{,}554 \cdot 10^{-2}$	0	3	554	- 2	(4)
– 0,56	$-\ 5{,}6 \cdot 10^{-1}$	1	5	6	- 1	(2)

Dans cet exemple, on a vu deux décompositions différentes : l'écriture classique (à **virgule fixe**) et l'écriture scientifique (à **virgule flottante**).

- **Virgule fixe** : un nombre s'écrit $\pm aaa, bbb$ avec *aaa* liste des nombres avant, et *bbb* liste de ceux après la virgule.

Exemples :	94,72	10 000 000	0,000 000 54
Entiers à stocker : (hors bit de signe)	94 et 72	10 000 000 et 0	0 et 000 000 54
Digits nécessaires :	2 + 2	8 + 1	1 + 8

- **Virgule flottante** : un nombre s'écrit $\pm c, mmm \cdot 10^e$

Exemples :	94,72	10 000 000	0,000 000 54
Écriture scientifique :	$9{,}472 \cdot 10^2$	$1{,}0 \cdot 10^7$	$5{,}4 \cdot 10^{-7}$
Entiers à stocker : (hors bit de signe)	9 et 472 et 2	1 et 0 et 7	5 et 4 et -7
Digits nécessaires :	1 + 3 + 1	1 + 1 + 1	1 + 1 + 1

Conclusion : On constate que l'écriture à virgule flottante est beaucoup plus concise (nécessite de stocker beaucoup moins de digits, donc de bits, gestion optimisée de la mémoire) que la virgule fixe. D'autant plus vrai que le nombre est « éloigné » de 1 ($|x| \gg 1$ ou $|x| \ll 1$).

Remarque : Si la mantisse est de taille quelconque (nombre fixe mais non borné de chiffres après la virgule), alors on peut coder tout nombre décimal. En pratique, nous allons « réserver » un nombre donné de digits (bits, en base 2) pour coder la mantisse.

- Alors, l'ensemble des *float* (valeurs pouvant être codées par virgule flottante) est un sous-ensemble de D (cette remarque n'est pas anodine, nous y reviendrons).

E.2 / Écriture d'un *float* (en binaire)

En informatique, les nombres à virgule seront donc codés en **virgule flottante binaire**, suivant la norme IEEE 754. Au programme de CPGE, vous devez être capables d'interpréter la norme, mais n'avez pas à la connaître par cœur.

Si on écrit un nombre binaire en virgule flottante : $x = (-1)^s \cdot (c, m) \cdot 2^e$

- s : est le signe (codé sur 1 bit : 0 si + ; 1 si –)
- c : est la caractéristique : $c \in \{0,1\}$ (car binaire)
  or la caractéristique doit être $\neq 0$
  → inutile de stocker *c*, il vaut forcément 1
- m : est la mantisse $m \in \mathbb{N}$: *N* bits en codage naturel
- e : est l'exposant. $e \in \mathbb{Z}$: *M* bits en codage binaire par excès

} Valeurs de N et M imposées par la norme

La norme IEEE 754 permet, selon la précision désirée (taille de mantisse) de choisir entre 2 codages :

- *float* **simple** précision : *N* = 23 bits (mantisse) *M* = 8 bits (exposant) +1 (signe) → total 32 bits (4 octets)
- *float* **double** précision : *N* = 52 bits (mantisse) *M* = 11 bits (exposant) +1 (signe) → total 64 bits (8 octets)

		Nombre de bits	Bit signe	*c* (Bit implicite)	Mantisse *m* (*N* Bits)	Exposant *e* (*M* Bits)	Décalage
Float	Simple précision	32	1	(1)	23	8	127
Double	Double précision	64	1	(1)	52	11	1023

Remarque 1 : en décimal, 0.1 = 0.1000. De même en binaire : pour atteindre le bon nombre de bits, la mantisse doit être complétée **par la droite** !

Remarque 2 : par défaut, *Python* utilise des *float* double précision (car les processeurs sont en 64 bits), mais pour la plupart des applications temps-réel, les *float* simple précision sont suffisants.

Seule la méthode est à connaître. Découvrons-la au travers de deux exemples :

Exemples : Écrire $x = (10.25)_{10}$ en float simple, sous la forme normalisée $(s|e|m)_2$, de même pour $y = (-0.625)_{10}$

1. Écrire le bit de signe : Pour $x > 0$ → $s = 0$ Pour $y < 0$ → $s = 1$
2. Décomposer la valeur absolue dans la base des 2^N :

	… 2^4	2^3	2^2	2^1	2^0	,	2^{-1} (=½)	2^{-2} (=¼)	2^{-3} (=1/8)	…
	16	8	4	2	1	,	0.5	0.25	0.125	
10.25 = 8 + 2 + 0.25 =		1	0	1	0	,	0	1		
0.625 = 0.5 + 0.125 =					0	,	1	0	1	

3. Multiplier par 2^k (décalage de *k*) pour la mettre sous forme scientifique binaire $(1, m) \cdot 2^e$

   $(10.25)_{10} = (1010{,}01)_2 \cdot 2^0 = (1{,}01001)_2 \cdot 2^3$

   $(0.625)_2 = (0{,}101)_2 \cdot 2^0 = (1{,}01)_2 \cdot 2^{-1}$

4. Écrire l'exposant en binaire par excès (ici, en simple précision, sur 8 bits) :

   3 + 127 = 130 avec $(130)_{10} = (128)_{10} + (2)_{10} = (1000\ 0010)_2$
   -1 + 127 = 126 avec $(126)_{10} = (127)_{10} - (1)_{10} = (0111\ 1111)_2 - (1)_2 = (0111\ 1110)_2$

5. Conclure, en complétant si besoin la mantisse **par la droite !**

$(10.25)_{10} = (0|1000\ 0010|\mathbf{01001}\ 00000\ 00000\ 00000\ 000)_2$
et $(-0.625)_{10} = (1|0111\ 1110|\mathbf{01}000\ 00000\ 00000\ 00000\ 000)_2$

E.3 / Limites de l'écriture *float*

➢ **Infiniment grand et NaN :**

En float simple précision le plus grand exposant que l'on peut coder est (sachant que 255 est réservé pour ***NaN*** **:** *Not a Number* qui peut être utilisé pour coder l'∞) : 254 – 127 = $(127)_{10}$
Sachant que la caractéristique est toujours *c* = 1, pour une mantisse = 23 bits à 1 et l'exposant précédent, on obtient le plus grand nombre codable : $(1{,}111\ldots)_2 \cdot 2^{127} \approx 2^{128} \approx (3{,}4 \cdot 10^{38})_{10}$

En float double précision (utilisé nativement par *Python*), cette limite dite d'overflow est d'environ $1{,}8 \cdot 10^{308}$
Au-delà de cette limite, on fixe l'exposant à 1 1 1 1 … (11 bits = 1, en double précision).

Exemples :

```
In [1]: from sys import *   # bibliothèque système

In [2]: float_info   # infos sur le codage float, double par défaut en Python
Out[2]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)

In [3]: x = 1.8*10**309    # supérieur à max
Traceback (most recent call last):
OverflowError: int too large to convert to float

In [4]: x = 1.7*10**308    # peut être codé en float

In [5]: print(x)
1.7e+308

In [6]: x = x*2   # dépasse le max... va devenir un NaN

In [7]: print(x)
inf
```

➢ **Infiniment petit et erreurs d'arrondi :**

On comprend (c'est l'objet de la remarque en bas de page 8) que l'écriture en virgule flottante impose de « discrétiser » l'ensemble (infini) des réels en un ensemble discret (fini) des flottants, espacés entre eux d'un ε de l'ordre de $2 \cdot 10^{-16}$ en double précision.

Tout nombre réel (par exemple π) est donc **approché** par un *float* à $\pm\varepsilon$ près. On parle de **précision machine**.

➔ **Conséquence 1** : il ne faut **jamais** faire de tests d'égalité sur des *float* ! **Exemples :**

```
In [1]: X1 = 1 + 5e-16

In [2]: X2 = 1 + 1e-16 + 1e-16 + 1e-16 + 1e-16 + 1e-16

In [3]: X1 == X2
Out[3]: False

In [4]: print(X1)
1.0000000000000004

In [5]: print(X2)
1.0

In [6]: X1 == 1 + 4e-16
Out[6]: True
```

```
In [1]: 0.1 + 0.2
Out[1]: 0.30000000000000004

In [2]: 0.1 + 0.2 == 0.3
Out[2]: False
```

→ On ne peut faire des tests du type `a == b` que sur des entiers (et des str)

→ *Comment faire, alors ?*

On pose (à l'avance) un ε petit, c'est-à-dire : $\varepsilon \ll \min(|a|, |b|)$ … mais pas trop petit : $\varepsilon > 5 \cdot 10^{-16}$

Pour vérifier que $a \approx b$ avec des *float*, on vérifie alors si `abs(b-a) < ε`

➔ **Conséquence 2** : les problèmes d'arrondi successifs peuvent mener à des grosses erreurs sur des procédures récursives ou récurrentes avec un grand nombre d'itération. **Exemple :** problème du pivot de Gauss.

➢ **Codage de la valeur « zéro »**

Puisque la caractéristique (1er chiffre significatif) vaut forcément $c = 1$ → on ne peut pas coder 0

Le plus petit nombre (en valeur absolue) pouvant être codé est + 0 = $(1{,}0000\ldots)_2 \cdot 2^{e_{\min}}$ avec une mantisse (000…00).
En *float* simple, $e_{\min} = -127$ donc $|x|_{\min}^{float} = 2^{-127} \approx 6 \cdot 10^{-39}$.

Remarque 1 : de la sorte, $+0$ et -0 ne se codent pas de la même façon (on change le bit de signe), et obtient $\approx -6 \cdot 10^{-39}$

Remarque 2 : il ne faut donc jamais tester si $x == 0$ pour un x de type *float* (car 0 n'est jamais atteint), ni $x == 0.0$ car si x vaut «-0», ce test peut renvoyer `False`.

F – Codage des caractères et chaînes de caractères

Dans la plupart des langages de programmation, on distingue le type **caractère** (*char*) du type **chaîne de caractères** (*string*). Il existe plusieurs types d'encodage, qui **codent tous un *char* en un entier naturel *int***.

Citons notamment le type *ASCII* (*American Standard Code for Information Interchange*) qui code un caractère comme un entier naturel sur 1 octet (8 bits). Les 127 premières valeurs sont standardisées, les 128 autres dépendent du pays et de la langue (selon les accents ou caractères spéciaux usuels).

0 NUL	32 espace	64 @	96 `
1 SOH	33 !	65 A	97 a
2 STX	34 "	66 B	98 b
3 ETX	35 #	67 C	99 c
4 EOT	36 $	68 D	100 d
5 ENQ	37 %	69 E	101 e
6 ACK	38 &	70 F	102 f
7 BEL	39 '	71 G	103 g
8 BS	40 (	72 H	104 h
9 HT	41)	73 I	105 i
10 LF	42 *	74 J	106 j
11 VT	43 +	75 K	107 k
12 FF	44 ,	76 L	108 l
13 CR	45 -	77 M	109 m
14 SO	46 .	78 N	110 n
15 SI	47 /	79 O	111 o
16 SLE	48 0	80 P	112 p
17 CS1	49 1	81 Q	113 q
18 DC2	50 2	82 R	114 r
19 DC3	51 3	83 S	115 s
20 DC4	52 4	84 T	116 t
21 NAK	53 5	85 U	117 u
22 SYN	54 6	86 V	118 v
23 ETB	55 7	87 W	119 w
24 CAN	56 8	88 X	120 x
25 EM	57 9	89 Y	121 y
26 SIB	58 :	90 Z	122 z
27 ESC	59 ;	91 [	123 {
28 FS	60 <	92 \	124 |
29 GS	61 =	93]	125 }
30 RS	62 >	94 ^	126 ~
31 US	63 ?	95 _	127 ■

(ceci n'est évidemment pas à savoir !)

Une alternative courante est l'encodage *UNICODE*, qui code un caractère sur 2 octets (65 535 valeurs contre 256 pour l'*ASCII*). L'avantage principal est une standardisation car il y a plus « d'adresses » disponibles que de caractères spéciaux nécessaires, l'inconvénient est qu'il n'optimise pas la gestion de mémoire.

Python utilise par défaut l'encodage *UNICODE* pour chaque caractère. Une **chaîne de caractères** (*str*) est stockée dans la RAM de façon contigüe (cases mémoires jointives). Par exemple, le *str* :

```
Texte = "Vive la PT"
```

compte `len(Texte) = 10` caractères (penser aux espaces !) et est donc stocké sur 20 octets (= 160 bits)