Représentation des nombres

Les données dans la mémoire d'un ordinateur sont stockées sous forme de bits (un **bit** est une unité d'information qui ne prend que les valeurs 0 ou 1). On explique ici comment les entiers et les réels sont représentés dans un ordinateur.

Représentation des entiers

Soit b un entier supérieur ou égal à 2. Pour tout $n \in \mathbb{N}$, il existe un unique $p \in \mathbb{N}$ et une unique famille $(a_0, \ldots, a_p)$ d'éléments de $\{0, \ldots, b-1\}$, avec $a_p \neq 0$, tels que :

$$n = a_0 + a_1 b + a_2 b^2 + \ldots + a_p b^p.$$

On note alors $n = \overline{a_p a_{p-1} \ldots a_1 a_0}_b$: c'est **l'écriture de n en base b.**

Exemples :

– En base 10 : $\overline{2587}_{10} = 2 \times 10^3 + 5 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$.

– En base 2 : $\overline{1101011}_2 = 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 64 + 32 + 8 + 2 + 1 = 107$.

– En base 16 : on utilise les chiffres $0123456789ABCDEF$. Ainsi $\overline{1A2E}_{16} = 1 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 14 \times 16^0 = 7602$.

Pour passer de la base 10 à la base b, on procède par divisions euclidiennes successives par b et on récupère les restes dans l'ordre inverse. Par exemple, pour convertir 113 en base 2, on a successivement :

$$\begin{aligned} 113 &= 2 \times 56 + 1 \\ 56 &= 2 \times 28 + 0 \\ 28 &= 2 \times 14 + 0 \\ 14 &= 2 \times 7 + 0 \\ 7 &= 2 \times 3 + 1 \\ 3 &= 2 \times 1 + 1 \\ 1 &= 2 \times 0 + 1 \end{aligned}$$

Ainsi $\overline{113}_{10} = \overline{1110001}_2$. De même, pour convertir 123 en base 16 :

$$\begin{aligned} 123 &= 7 \times 16 + 11 \\ 7 &= 0 \times 16 + 7 \end{aligned}$$

donc $\overline{123}_{10} = \overline{7B}_{16}$.

En Python, la fonction `bin` permet de passer de la base 10 à la base 2 et `int(., 2)` permet de faire l'opération inverse.

```
>>> bin(113)
'0b1110001'
>>> int('1110001', 2)
113
```

Dans un ordinateur, les bits sont regroupés par groupes de 8 (les **octets**) et les entiers en base 2 sont représentés par des mots de 1, 2, 4 ou 8 octets (soit 8, 16, 32 ou 64 bits). Par exemple, un processeur 8 bits représente 2 ($\overline{10}_2$ en binaire) sous la forme 00000010.

Pour représenter les entiers relatifs sur N bits, on utilise le bit de gauche pour désigner le signe : 0 pour $+$, 1 pour $-$. Sur 8 bits, on pourrait ainsi représenter -2 sous la forme 10000010, mais cela présenterait des inconvénients (en particulier 0 aurait deux représentations 00000000 et 10000000).

En pratique, la méthode employée est le **complément à 2** : la représentation d'un entier strictement négatif n est celle de $n + 2^N$. Ainsi la représentation de -2 sur 8 bits est celle de $-2 + 2^8 = 254$, i.e. 11111110. Sur N bits on peut ainsi représenter les entiers compris entre -2^{N-1} et $2^{N-1} - 1$.

L'intérêt de cette méthode est que 0 n'a qu'une représentation et on peut montrer que les algorithmes d'addition, de soustraction et de multiplication usuels restent valables.

On peut démontrer que pour obtenir rapidement le complément à 2 d'un entier strictement négatif on change en 1 (resp. en 0) tous les 0 (resp. les 1) de l'entier positif correspondant (c'est le complément à 1), puis on ajoute 1 au résultat. Ainsi, pour représenter -2, on prend le complément à 1 de 00000010, soit 11111101, et on ajoute 1, ce qui donne 11111110.

Noter enfin qu'en Python, la taille des entiers n'est pas limitée (si ce n'est par la mémoire de l'ordinateur). Pour représenter les entiers non compris entre -2^{N-1} et $2^{N-1} - 1$ on utilise plusieurs mots de N bits : ce sont des **entiers multi-précision**. Cependant, les calculs sur ces entiers longs sont plus coûteux (en temps et en mémoire) que sur les entiers courts.

Exercice 1

1) Convertir 73 en base 2 puis en base 16.

2) Convertir $\overline{1101001}_2$ et $\overline{21D}_{16}$ en base 10.

3) Représenter -73 sur 8 bits en utilisant la méthode du complément à 2.

4) Quel est l'entier dont le complément à 2 sur 8 bits est 10100110 ?

Représentation des réels

Soit b un entier supérieur ou égal à 2. Tout réel $x \in [0, 1[$ peut s'écrire sous la forme :

$$x = \sum_{k=1}^{+\infty} a_k b^{-k} = a_1 b^{-1} + a_2 b^{-2} + a_3 b^{-3} + \ldots,$$

où les $a_k \in \{0, \ldots, b-1\}$. On note alors $x = \overline{0{,}a_1 a_2 a_3 \ldots}_b$.

Exemples :

– On a $0{,}253 = \overline{0{,}253}_{10} = 2 \times 10^{-1} + 5 \times 10^{-2} + 3 \times 10^{-3}$.

– On a $\dfrac{1}{3} = \overline{0{,}333\ldots}_{10} = 3 \times 10^{-1} + 3 \times 10^{-2} + \ldots = \displaystyle\sum_{k=1}^{+\infty} 3 \times 10^{-k}$.

– On a $\overline{0{,}101}_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,625$.

Pour calculer le développement binaire d'un réel compris entre 0 et 1, on peut utiliser la méthode suivante. Soit $x = \overline{0{,}a_1a_2a_3\ldots}_2 = a_12^{-1} + a_22^{-2} + a_32^{-3} + \ldots$. Alors $2x = a_1 + a_22^{-1} + a_32^{-2} + \ldots$, donc $a_1 = \lfloor 2x \rfloor$. Ensuite, si on pose $x' = a_22^{-1} + a_32^{-2} + \ldots$, on a $2x' = a_2 + a_32^{-1} + \ldots$, donc $a_2 = \lfloor 2x' \rfloor$, et ainsi de suite.

En pratique, on utilise la disposition suivante. Soit par exemple 0,4 à convertir en binaire :

	0,4	0,8	1,6	1,2	0,4	0,8	1,6	1,2	...
$\lfloor . \rfloor$	0,	0	1	1	0	0	1	1	...
reste	0,4	0,8	0,6	0,2	0,4	0,8	0,6	0,2	...

Ainsi $0{,}4 = \overline{0{,}0110011001100\ldots}_2$. On voit que son développement binaire est infini, on ne pourra donc pas le représenter entièrement sous cette forme dans la mémoire finie d'un ordinateur.

Nombres flottants

Un **nombre à virgule flottante** ou simplement un **flottant** à n digits dans la base b est un triplet (s, m, e) représentant le nombre $s \times m \times b^e$.

s est le **signe** du flottant, m sa **mantisse** et e son **exposant**. Par exemple, si on écrit $-123{,}45$ sous la forme $-1{,}2345 \times 10^2$, le signe est -1, la mantisse est 1,2345 et l'exposant est 2.

Remarque : tout ce qui suit concernant la norme IEEE 754 n'est pas à retenir.

Dans un ordinateur, les flottants sont représentés selon la norme **IEEE 754**. En base 2, en double précision (64 bits), s est codé sur 1 bit (0 pour + et 1 pour −), m appartient à l'intervalle $[1, 2[$ (sauf si $m = 0$) et est codé sur 52 bits (le premier 1 n'est pas représenté) et e est un entier compris entre -1022 et 1023 représenté par $e + 1023$ et codé sur 11 bits. L'exposant est placé avant la mantisse. En base 10 cela correspond à environ 16 chiffres significatifs.

Le nombre 0 a deux représentations avec cette norme :

$$0\ 00000000000\ 0000000000000000000000000000000000000000000000000000$$

et

$$1\ 00000000000\ 0000000000000000000000000000000000000000000000000000$$

Exemple : on a vu que $0{,}4 = \overline{0{,}0110011001100\ldots}_2 = \overline{1{,}10011001100\ldots}_2 \times 2^{-2}$. L'exposant est -2 et on arrondit la mantisse à

$$\overline{1{,}1001100110011001100110011001100110011001100110011010}_2$$

pour avoir 52 chiffres après la virgule. L'exposant est représenté par $-2 + 1023 = 1021 = \overline{1111111101}_2$. Ainsi dans la mémoire de l'ordinateur 0,4 est représenté sur 64 bits par

$$0\ 1111111101\ 1001100110011001100110011001100110011001100110011010.$$

Si on reconvertit ce nombre en décimal, on obtient 0,40000000000000002220446049250 3 : il y a une erreur de 2^{-52} due à l'arrondi effectué.

On retiendra que l'arithmétique des nombres flottants peut mener à des résultats surprenants et nécessite beaucoup de précautions.

– Problèmes de retenues lors des opérations en binaire :

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> 0.5 - 0.4
0.09999999999999998
```

– L'addition n'est plus associative :

```
>>> (0.3 + 0.7) + 0.1
1.1
>>> 0.3 + (0.7 + 0.1)
1.0999999999999999
```

– Dépassement de la capacité de la mantisse :

```
>>> 2**53 + 1 - 2**53
1
>>> 2**53 + 1.0 - 2**53
0.0
```

– Dépassement de la capacité de l'exposant :

```
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1024
OverflowError: (34, 'Result too large')
```

– Problèmes de comparaison à 0 : il est absurde de comparer un flottant à 0. Supposons par exemple qu'on veuille résoudre l'équation $0{,}1x^2 + 0{,}6x + 0{,}9 = 0$. Si on calcule son discriminant :

```
>>> 0.6**2 - 4*0.1*0.9
-5.551115123125783e-17
```

Un programme de résolution des équations du second degré qui comparerait le discriminant à 0 en conclurait qu'il n'y a pas de solutions réelles, alors qu'il y a une solution double.

Exercice 2 Convertir 0,1 et 0,3 en binaire avec 7 chiffres après la virgule. Calculer $0{,}1 + 0{,}1 + 0{,}1$ et comparer avec 0,3.

Exercice 3 Que se passe-t-il quand on exécute le programme Python suivant ?

```
>>> x = 0.3
>>> while x != 0:
...     x -= 0.1
...     print(x)
```