II – Représentation des nombres

1/ Représentation des entiers naturels

1.1) Notion de base

De nos jours, on utilise la *représentation décimale* pour écrire les entiers positifs. Cela revient à regrouper les quantités que l'on dénombre par paquets de dix, puis paquets de dix paquets de dix, etc ... Ainsi, de droite à gauche, le premier chiffre représente le nombre d'éléments isolés, le suivant le nombre de paquets de dix, puis le nombre de paquets de cent, etc ...

Ce **système décimal** voit le jour en Inde au Vème siècle ap. J-C, à partir d'un système chinois du IIème siècle av. J-C. Il est ensuite propagé au Proche-Orient et en Europe durant le moyen-âge. Sa simplicité réside dans le fait que c'est un **système positionnel** (comme les systèmes de numération babyloniens ou mayas) : il n'y a pas besoin de répéter plusieurs fois le même symbole, on joue sur les positions pour représenter différentes puissances de 10. De plus, on peut effectuer des opérations à l'aide de tables.

Exemple : en chiffres romains, 1987 s'écrit MCMLXXXVII ... c'est peu lisible !

On peut aussi regrouper par paquets non plus de 10, mais de 2 (en Océanie), de 20 (chez les Mayas) ou de 60 (en Mésopotamie). Tout dépend en fait du choix de la **base**.

Définition

Fixons un entier naturel $b \geqslant 2$ et considérons un entier $n \in \mathbb{N}^*$.
*La **représentation de** n **en base** b est l'écriture*

$$\overline{a_p a_{p-1} \ldots a_1 a_0}^b$$

où $a_i \in \{0, \ldots, b-1\}$ pour tout $i \in \{0, \ldots, p\}$ et $\boxed{\sum_{i=0}^{p} a_i b^i = n.}$

*Les entiers (a_i) sont appelés les **chiffres** de la représentation.*

Remarque : quand $b > 10$, on représente les chiffres suivant 9 par A, B, C, etc ...

Exemples : $1987 = 1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$; $\overline{1011}^2 = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 11$.

On voit ci-dessus que le passage d'une base quelconque à la base 10 est un simple calcul. Par contre, pour trouver la représentation d'un entier dans la base b, il faut effectuer des divisions euclidiennes successives par b. Reprenons en effet les notations précédentes :

$$n = \sum_{i=0}^{p} a_i b^i = \left(\sum_{i=1}^{p} a_i b^{i-1} \right) \times b + a_0$$

donc a_0 est le reste de la division de n par b tandis que $n_1 = \sum_{i=1}^{p} a_i b^{i-1}$ en est le quotient.

Il suffit de réitérer le procédé avec n_1 pour trouver le chiffre suivant a_1, et ainsi de suite.

Exemples : avec cette méthode, retrouver $57 = \overline{111001}^2$ et $13 = \overline{1101}^2$.

Les systèmes positionnels permettent également d'effectuer des additions et des multiplications directement à partir des représentations : il suffit de connaître les tables d'addition, les tables de multiplication et d'utiliser des retenues, comme en primaire ! Tout ceci découle en fait de la distributivité de la multiplication par rapport à l'addition.

Exemples : disséquer $23 + 18$ et 23×18, en écrivant $23 = 2 \times 10 + 3$ et $18 = 10 + 8$.

Si la base est petite, les tables sont simples mais les nombres sont longs à écrire ; *a contrario*, si la base est grande, l'écriture des nombres est concise mais les tables sont complexes.

Exemples : calculer $\overline{10011}^2 \times \overline{1110}^2$ et $\overline{734}^8 + \overline{3312}^8$.

1.2) La représentation binaire, source du langage informatique

La mémoire des ordinateurs est constitués de circuits électroniques élémentaires qui ne peuvent se trouver que dans deux états : hors tension ou sous tension. On les note 0 et 1, ce qui permet de coder des entiers en base 2, avec autant de chiffres ou **bits** que de circuits.

Dans la mémoire d'un ordinateur, ces circuits sont groupés par huit, formant un **octet**. L'ordinateur représente alors les nombres sur 1, 2, 4 ou 8 octets, soit 8, 16, 32 ou 64 bits. Par exemple, avec 64 bits, l'ordinateur peut représenter tous les entiers de 0 à $2^{64} - 1$, c'est-à-dire $2^{64} \simeq 18 \times 10^{18}$ nombres.

Remarque : on utilise en informatique les bases 8 et surtout 16, qui sont plus concises et se prêtent bien aux conversions depuis et vers la base 2. En effet, on regroupe les bits binaires 3 par 3 pour passer en base 8, et 4 par 4 pour la base 16.

Par exemple, l'encodage des couleurs au format RGB consiste à utiliser 6 chiffres en base 16, chaque paire servant à coder la nuance (de 00 à FF) de rouge, de vert et de bleu.

Exemples : $\overline{10\,011}^2$ s'écrit $\overline{23}^8$ en base 8 et $\overline{64}^8$ s'écrit $\overline{110\,100}^2$ en base 2.

2/ Représentation des entiers négatifs

Un ordinateur équipé d'un processeur à 64 bits peut manipuler les entiers de 0 à $2^{64} - 1$. Pour tenir compte des entiers négatifs, on va chercher à coder les entiers de -2^{63} à $2^{63} - 1$.

2.1) Codage avec bit de signe

On code le signe sur le premier bit ou *bit de signe* (0 pour + et 1 pour −), et la valeur absolue sur les 63 bits restants. C'est simple, mais cela pose plusieurs problèmes : il existe deux zéros, la comparaison à 0 est problématique et les opérations « naïves » sur des nombres de signes différents posent des problèmes, comme le montre le calcul $2 + (-2)$.

2.2) Codage en complément à 2

Définition

Soit n un entier compris entre entre -2^{63} et $2^{63}-1$.

- *Si $n \geqslant 0$, son codage est sa représentation binaire sur 64 bits.*
- *Si $n < 0$, c'est celle de l'entier $n + 2^{64}$, qui est compris entre 2^{63} et $2^{64}-1$.*

Notons que les entiers positifs commencent par 0 et les négatifs commencent par 1.

Remarque : les codages n et $\widetilde{n}$ d'un entier et de son opposé vérifient $n+\widetilde{n} = 2^{64}$. On obtient ainsi $\widetilde{n}$ en intervertissant les chiffres (binaires) de n et en lui ajoutant 1.

Ce codage en complément à 2 est la méthode la plus utilisée pour représenter les entiers relatifs, car elle a un bon comportement vis-à-vis des opérations (on oublie les retenues au-delà du 64e bit). Ceci n'exclut pas des problèmes de **dépassement arithmétique**.

Exemple : si l'on code sur 4 bits l'opération $5+6$, on obtient -5 ... avec 5×6 c'est pire !

Pour pallier ces problèmes, les langages de programmation ont pour la plupart un type d'entiers particuliers, les **entiers longs**, qui sont codés sur plusieurs nombres de 64 bits.

3/ Représentation des réels

3.1) Les nombres à virgule

La notation binaire permet de représenter des nombres non entiers en utilisant une virgule, comme la notation décimale : les chiffres à gauche de la virgule représentent les puissances positives de deux et ceux situés à droite représentent les puissances négatives. Plus précisément, intéressons-nous aux réels situés entre 0 et 1.

Définition

Soit x un réel positif strictement inférieur à 1.
Sa représentation binaire est l'écriture

$$\overline{0,a_1a_2a_3a_4a_5\ldots}^{2}$$

avec $a_i \in \{0,1\}$ pour tout i et $\boxed{\sum_{i\geqslant 1} a_i\, 2^{-i} = x.}$

Pour trouver la représentation binaire d'un tel réel x, il faut effectuer des multiplications successives par 2, en retenant et retirant la partie entière à chaque étape. En effet :

$$2x = \sum_{i\geqslant 1} a_i\, 2^{1-i} = \sum_{i\geqslant 1} a_i\, 2^{-(i-1)} = a_1 + \sum_{j\geqslant 1} a_{j+1}\, 2^{-j}$$

en posant $j = i-1$. On réitère le procédé avec $x_1 = 2x - a_1$ pour obtenir a_2, et ainsi de suite.

Remarque : la représentation binaire de 0.2 explique la valeur de $1-0.2-0.2$ en Python.

3.2) Codage des nombres à virgule flottante (norme IEEE 754)

Pour représenter les nombres réels très grands ou très petits, la méthode précédente est inadaptée (trop de chiffres). On va ainsi écrire les réels sous la forme scientifique binaire

$$x = (-1)^s \times m \times 2^n$$

où par convention :

- le **bit de signe** s vaut 0 si x est positif et 1 si x est négatif ;
- l'**exposant** n est un entier compris entre -1022 et 1023 ; pour gérer les exposants négatifs, on code sur 11 bits l'entier naturel $n + 1023$, qui est compris entre 1 et 2046.
- la **mantisse** m est un réel de l'intervalle $[1;2[$ représenté en binaire avec 52 chiffres après la virgule ; comme il y aura fatalement le chiffre 1 avant la virgule, on ne conserve que les autres chiffres, qui occupent 52 bits (soit 16 chiffres significatifs en base 10).

On représente ainsi les réels sur $1 + 11 + 52 = 64$ bits. Dans le cas de processeurs à 32 bits, on consacre 8 bits à l'exposant et 23 à la mantisse (plus que 7 chiffres significatifs).

Exemple : le réel -108.25 va se coder $1\,10000101\,10110001000000000000000$ sur 32 bits.

On convient qu'un nombre vaut 0 si, et seulement si, les bits de son exposant et de sa mantisse sont tous nuls (c'est en réalité $\pm 2^{-1023}$). De même, un nombre vaut $\pm\infty$ si les bits de son exposant valent tous 1 et ceux de sa mantisse 0. Enfin, quand les bits de l'exposant valent tous 1 mais que la mantisse n'est pas nulle, on obtient *Nan* (pour *not a number*).

3.3) Problèmes liés à cette représentation

Limitations due au nombre de bits utilisés

Si l'exposant d'un réel est supérieur à 1023, ou bien s'il est égal à 1023 et que la mantisse vaut $\overline{1,1111\ldots1111}^2$, on obtient un **overflow** ou dépassement arithmétique. Dans ce cas, la machine renvoie un résultat infini.

De même, si l'exposant est inférieur à -1022, on obtient un **underflow**. Le résultat va être arrondi à 0, ce qui peut produire des erreurs comme une division par 0.

Exemples : calculs de 18.5^{500} et de $1/9^{-1000}$.

Précisions et arrondis

En général, la représentation d'un réel sur 64 bits ne permet pas d'obtenir sa valeur exacte (penser à 0.2), et les nombres à virgule flottante ne sont que des valeurs approchées des réels. Par défaut, on prendra pour mantisse la valeur la plus proche de la « vraie » mantisse.

Remarque : la précision de la mantisse est de l'ordre de 2^{-23} sur 32 bits et 2^{-52} sur 64 bits.

Concrètement, les tests du style « $a = b$ » sont à proscrire quand les variables sont réelles, du fait des erreurs d'arrondis qu'elles ont pu subir et qui vont tout fausser.
On leur préfèrera des tests comme « $|a - b| < 10^{-9}$ ».

Exemple : sous Python, si l'on pose $a = 1 + \sqrt{13}$, alors $a^2 - 2a - 12$ ne renvoie pas 0.

Exercices

Exercice 1.
Donner l'écriture décimale des nombres suivants :

$$\overline{111}^{2} \qquad \overline{210}^{3} \qquad \overline{421}^{5} \qquad \overline{473}^{8} \qquad \overline{\mathrm{A5}}^{11} \qquad \overline{\mathrm{F01}}^{16}$$

Exercice 2.
Même exercice avec les nombres suivants :

$$\overline{10101010}^{2} \qquad \overline{202413}^{5} \qquad \overline{666}^{8} \qquad \overline{\mathrm{5AD9}}^{16}$$

Exercice 3.

1) Donner la représentation de 14 en base 2.
2) Donner la représentation de 58 en base 5.
3) Donner les représentations de 1207 en base 16.

Exercice 4.

1) Donner la représentation de 54 en base 2.
2) Donner la représentation de 872 en base 8.
3) Donner les représentations de 1987 en bases 2, 5 et 16.

Exercice 5.

1) Représenter en base 2 les nombres 1, 3, 7, 15, 31 et 63.
2) Qu'observe-t-on ? Comment l'expliquer ?

Exercice 6.
Donner l'écriture décimale des nombres $\overline{11011011}^{2}$ et $\overline{11110111001}^{2}$.

Exercice 7.

1) Dresser les tables d'addition et de multiplication en bases 2 et 5.
2) Calculer $\overline{100101}^{2} + \overline{1110}^{2}$, $\overline{100101}^{2} \times \overline{1110}^{2}$, $\overline{421}^{5} + \overline{143}^{5}$ et $\overline{421}^{5} \times \overline{143}^{5}$.

Exercice 8.
Pour multiplier par 10 un entier, on ajoute un zéro à la fin de sa représentation décimale.
Quel est l'équivalent en binaire ?

Exercice 9.
Convertir $\overline{1011011}^{2}$ en base 16, puis $\overline{F4}^{16}$ en base 2.

Exercice 10. Sur 4 bits :

1) Donner les représentations de 4 et de −6, puis calculer 4 − 6.
2) Donner les représentations de −4 et −3, puis calculer −4 − 3.

Ne pas oublier ... d'oublier les retenues au-delà du 4e bit.

Exercice 11. Sur 8 bits :

1) Donner les représentations des entiers relatifs 11, −11, 127 et −127.
2) Trouver les entiers qui se cachent derrière 01010101 et 10101010.

Exercice 12.
Donner les représentations binaires des nombre suivants :

(a) 0.625 (b) 18.75 (c) 1/3 (d) 0.2

Exercice 13. Sur 8 bits :

1) Représenter les entiers relatifs 101, −94, 99 et 57.
2) Ajouter les résultats. Pourquoi obtient-on des nombres négatifs ?

Exercice 14. Sur 32 bits :

1) Donner la représentation de 234.625.
2) Trouver qui se cache derrière 0 11001001 11101101000100000000000.

Les exercices suivants sont à faire dans une console `Ipython`

Exercice 15.
Calculer $0.3 - 0.2 - 0.1$ et 1000×0.1.

Exercice 16.
Calculer $2^{100} + 1 - 2^{100}$ et $2^{100} + 1.0 - 2^{100}$.

Exercice 17.
Calculer $(a+1)^2 - a^2 - 2a - 1$ pour les valeurs suivantes de a :

10^9 $10^9 \times 1.0$ 10^{10} $10^{10} \times 1.0$ 10^{11} $10^{11} \times 1.0$