Chapitre 8

Principes de la représentation des nombres en mémoire

Nous décrivons les principes de la représentation des nombres en mémoire.

8.1 Principe Mathématique de la représentation d'un entier dans une base

8.1.1 Ecriture d'un entier naturel dans une base

Nous avons coutume de représenter les nombres entiers par leur écriture décimale, ou écriture en base 10, qui utilise les 10 chiffres de 0 à 9. Ce principe de représentation d'un nombre entier dans une base se généralise à 2 chiffres, 3 chiffres, etc..., ou à tout nombre arbitraire > 1 de chiffres.

• Rappel (Division euclidienne) :

Théorème 1 (Division euclidienne.) *Pour tout entiers $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$, il existe une unique couple d'entiers $(q,r) \in \mathbb{N}^2$ tels que : $\begin{cases} a = q \times b + r \\ 0 \leqslant r < b \end{cases}$. q s'appelle le quotient et r le reste, dans la division euclidienne de a par b.*

Remarque : $q = \left\lfloor \frac{a}{b} \right\rfloor$ et $r = a - qb$ convient. En effet :

$$\left\lfloor \frac{a}{b} \right\rfloor \leqslant \frac{a}{b} < \left\lfloor \frac{a}{b} \right\rfloor + 1 \implies qb \leqslant a < qb + b \implies \begin{cases} a = q \times b + r \\ 0 \leqslant r < b \end{cases}$$

On déduit de ce résultat, le principe de la représentation d'un entier naturel à l'aide d'une base. La base est un entier > 1.

Théorème 2 *Pour tout entier $m \in \mathbb{N} \setminus \{0,1\}$, pour tout entier $n \in \mathbb{N}$, il existe une unique suite finie $(u_i)_{i \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que :*

$$n = \sum_{i=0}^{N} u_i \times m^i$$

avec si $N > 0$, $u_N \neq 0$.

On peut alors définir ce qu'est l'écriture d'une entier naturel dans la base m :

Définition. L'écriture en base m de l'entier n est :

$$u_N \, u_{N-1} \cdots u_1 \, u_0$$

pour la suite finie $(u_i)_{i \in [[0,N]]}$ donnée par le théorème.

Preuve. Existence : par récurrence forte. Fixons $m \in \mathbb{N} \setminus \{0,1\}$, et considérons pour proposition de récurrence :

$\mathscr{P}(n)$: "Il existe une suite finie $(u_i)_{i \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que $n = \sum_{i=0}^{N} u_i \times m^i$ avec si $N > 0$, $u_N \neq 0$".

Initialisation. La proposition $\mathscr{P}(0)$ est vraie. Soit $n = 0$, la suite $u_0 = 0$ convient.

Hérédité. Supposons la proposition $\mathscr{P}(k)$ vraie pour tout $k < n$, et montrons que $\mathscr{P}(n)$ est vraie.

Par division euclidienne de n par m, il existe un unique couple $(q,r) \in \mathbb{N}^2$ tel que $n = q \times m + r$ avec $0 \leqslant r < m$. Nécessairement $0 \leqslant q < n$ car autrement on aurait $n = q \times m + r > n$ (puisque $m > 1$ et $r \geqslant 0$).
Donc par hypothèse de récurrence il existe une suite finie $(v_n)_{n \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que $q = \sum_{i=0}^{N} v_i \times m^i$ avec si $N > 0$, $v_N \neq 0$".

Ainsi :

- Si $q > 0$:

$$n = q \times m + r = \left(\sum_{i=0}^{N} v_i \times m^i \right) \times m + r = \sum_{i=0}^{N} v_i \times m^{i+1} + r = \sum_{i=0}^{N'} u_i \times m^i$$

  en posant $N' = N + 1$ et pour $1 \leqslant i \leqslant N+1$, $u_i = v_{i-1}$ et $u_0 = r$.
- Si $q = 0$: $n = r = u_0$.

Puisque $0 \leqslant r < m$, la suite $u_0, u_1, \dots u_{N+1}$ est une suite d'entiers compris entre 0 et $m-1$. De plus si $N' > 0$, $u_{N'} = v_N \neq 0$.
Ainsi $\mathscr{P}(n)$ est vraie.

Unicité. Supposons que :

$$n = \sum_{i=0}^{N_1} u_i \times m^i = \sum_{i=0}^{N_2} v_i \times m^i$$

avec $u_{N_1} \neq 0$ si $N_1 > 0$ et $v_{N_2} \neq 0$ si $N_2 > 0$.
Premier cas : si $n = 0$ alors $N_1 = N_2 = 0$ et $u_0 = v_0 = 0$.

Deuxième cas : si $n > 0$. Alors u_{N_1} et v_{N_2} sont non-nuls. En effectuant la division euclidienne de n par m, on obtient :

$$n = q \times m + r \qquad \text{avec} \qquad r = u_0 = v_0 \quad \text{et} \quad q = \sum_{i=0}^{N_1-1} u_{i+1} \times m^i = \sum_{i=0}^{N_2-1} v_{i+1} \times m^i$$

En recommençant avec q à la place de n, on obtient $u_1 = v_1$, et en poursuivant ainsi de suite avec chaque quotient obtenu, $N_1 = N_2$ et $u_2 = v_2,\ u_3 = v_3, \dots,\ u_{N_1} = v_{N_2}$. Ceci prouve l'unicité. □

- Exemple : le nombre 10, s'écrit :

• $10 = 1 \times 10^1 + 0 \times 10^0$	10 en décimal.
• $10 = 1 \times 9^1 + 1 \times 9^0$	11 en base 9.
• $10 = 1 \times 8^1 + 2 \times 8^0$	12 en base 8.
• $10 = 1 \times 7^1 + 3 \times 7^0$	13 en base 7.
• $10 = 1 \times 6^1 + 4 \times 6^0$	14 en base 6.
• $10 = 2 \times 5^1 + 0 \times 5^0$	20 en base 5.
• $10 = 2 \times 4^1 + 2 \times 4^0$	22 en base 4.
• $10 = 1 \times 3^2 + 0 \times 3^1 + 1 \times 3^0$	101 en base 3.
• $10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	1010 en binaire.

Il apparaît implicitement dans la preuve (démonstration de l'unicité) un algorithme permettant d'obtenir la suite de chiffres de l'écriture d'un entier naturel arbitraire dans une base.

Théorème 3 (Algorithme d'écriture d'un entier dans une base)
Donné $m \in \mathbb{N} \setminus \{0,1\}$, donné une entier $n \in \mathbb{N}^*$, l'écriture $u_N\ u_{N_1} \cdots u_1\ u_0$ de l'entier n dans la base m s'obtient grâce à l'algorithme suivant :

- *Soit $k = 0$*
- *TANT QUE $n \neq 0$ FAIRE :*
  - *u_k est le reste dans la division euclidienne de n par m*
  - *Changer n par le quotient de n par m*
  - *Changer k par $k+1$*
- *FIN TANT QUE*
- *$u_{k-1}\ u_{k-2} \cdots u_2\ u_1\ u_0$ est l'écriture de n dans la base m.*

lorsque $n = 0$, son écriture dans la base m est simplement 0.

- Exemple : Ecriture de 123 en base 10 :

$$\begin{aligned} 123 &= 12 \times 10 + 3 & u_0 &= 3 \\ 12 &= 1 \times 10 + 2 & u_1 &= 2 \\ 1 &= 0 \times 10 + 1 & u_2 &= 1 \end{aligned}$$

Sans surprise on obtient comme écriture de 123 en base 10 (ou décimal) : 123.

- Exemple : Ecriture de 72 en base 2 :

$$\begin{aligned} 72 &= 2 \times 36 + 0 & u_0 &= 0 \\ 36 &= 2 \times 18 + 0 & u_1 &= 0 \\ 18 &= 2 \times 9 + 0 & u_2 &= 0 \\ 9 &= 2 \times 4 + 1 & u_3 &= 1 \\ 4 &= 2 \times 2 + 0 & u_4 &= 0 \\ 2 &= 2 \times 1 + 0 & u_5 &= 0 \\ 1 &= 2 \times 0 + 1 & u_6 &= 1 \end{aligned}$$

72 s'écrit en base 2 (ou binaire) : $\overline{1001000}$.

8.2 Représentation normalisée des entiers

8.2.1 Représentation des données en mémoire

L'unité de mémoire en informatique est le bit (physiquement constitué d'une unité électronique pouvant prendre deux états, par exemple un conden-

sateur, déchargé ou chargé).

En informatique toutes les données, quelles qu'elles soient, sont stockées sous la forme abstraite d'une suite de zéros et de uns.

Afin d'optimiser l'accès à la mémoire, un ordinateur manipule des unités de stockage plus importantes. Traditionnellement, l'unité de stockage est l'octet, constitué d'un groupement de 8 bits.

Les premiers ordinateurs personnels (transportables) étant '8 bits', c'est à dire que processeur, bus, et mémoires avaient pour unité de stockage l'octet, soit 8 bits.
– 8 bits (fin 70, premiers ordinateurs personnels, performances limitées)
– 16 bits (mi 80, les premiers PC, Apple Macintosh, Atari st, jeux graphiques,...)
– 32 bits (90', internet, interfaces graphiques, multimédia,...)
– 64 bits (00', tout connecté, multitâche multimédia)

8.2.2 Représentation normalisée des entiers naturels

En informatique les entiers naturels se représentent naturellement à l'aide de leur écriture en base 2.

Pour se donner un entier naturel on se donne habituellement son écriture en base 10. Un nombre dont l'écriture décimale est $a_N a_{N-1} \cdots a_1 a_0$ (a_i valant $0, 1, \ldots, 9$) désigne le nombre :

$$a_N \times 10^N + a_{N-1} \times 10^{N-1} + \cdots + a_1 \times 10 + a_0 = \sum_{k=0}^{n} a_k \times 10^k$$

De la même manière :

Pour tout entier naturel n il existe une unique suite suite a_N, a_{N-1}, a_1, a_0 de nombres valant 0 ou 1, tels que :

$$n = \sum_{k=0}^{N} a_k \times 2^k$$

avec $a_N \neq 0$ si $n \neq 0$, et la suite (0) pour $n = 0$.

• Sur un octet, 8 bits, on peut représenter tous les entiers de :
$\overline{00000000} = 0$ à $\overline{11111111} = 2^8 - 1 = 255$.

• Sur 2 octets, soit 16 bits, on peut représenter tous les entiers de :
$\overline{0000000000000000} = 0$ à $\overline{1111111111111111} = 2^{16} - 1 = 65535$.

• Sur 4 octets, soit 32 bits, on peut représenter tous les entiers de :
0 à $2^{32} - 1 = 4294967295$.

• Sur 8 octets, soit 64 bits, on peut représenter tous les entiers de :
0 à $2^{64} - 1 = 18446744073709551615$.

Remarquer que :

$$\underbrace{\overline{111\cdots1111}}_{n \text{ fois } 1} = \sum_{k=0}^{n-1} 2^k = \frac{1-2^n}{1-2} = 2^n - 1$$

ou encore que :

$$\underbrace{\overline{111\cdots1111}}_{n \text{ fois } 1} + \overline{1} = \overline{1\underbrace{000\cdots0000}_{n \text{ fois } 0}} = 2^n$$

8.2.3 Conversion entier binaire

En `python` il existe une fonction de conversion d'un nombre, vers une chaîne de caractère contenant son écriture en base 2. Il existe aussi les fonctions réciproques :

Conversion	`python`
décimal vers binaire	`bin`
binaire vers décimal	`int(.,2)`

```
>>> bin(10)
'0b1010'
>>> int('1010',2)
10
>>> int('0b1010',2)
10
```

• `python`met le préfixe '0b' en début d'une écriture binaire (optionnel). Le deuxième argument de `int` spécifie la base, ici 2.

On peut aussi coder : (conversion entier positif – > binaire)

```
def binaire(n):
    if n==0:
        return '0b0'
    res = ''
    while n>0:
        res = str(n%2) + res
        n = n//2
    return '0b' + res
```

(conversion str binaire – > nombre (décimal))

```
def decimal(s):
    s = s[2:]    # supprimer le préfixe '0b'
    N = len(s)
    n = 0
    for k in range(N):
        n += int(s[k]) * 2**(N-1-k)
    return n
```

```
def decimal(s):
    s = s[2:]    # supprimer le préfixe '0b'
    n = 0
    for char in s:
        n *= 2
        n += int(char)
    return n
```

8.2.4 Limitation de la représentation des nombres

• Lorsque les nombres sont représentés dans un espace alloué de taille fixé, on peut être confronté à des problèmes de dépassement dans les calculs.

Exemple : Dans la représentation sur 8 bits des entiers naturels :

$$\overline{1111\ 1111} + \overline{0000\ 0001} = \overline{1\ 0000\ 0000}$$

le résultat de l'addition de deux nombres d'un octet peut donner un résultat qui ne peut pas s'écrire sur 8 bits. Si le résultat de l'opération est représenté sur 8 bits, le résultat retourné sera erroné, ce sera 0 !

Il faut faire très attention à la façon dont sont codés les nombres dans le langage utilisé lorsque l'on effectue des calculs complexes. En fait on calcule dans $\mathbb{Z}/2^N\mathbb{Z}$ ($N = 8, 16, 32, 64$).

En `python` on n'a pas vraiment ce problème : si dépassement l'interpréteur augmente le nombre de bits alloués au codage : 64, 128, etc... autant que nécessaire (c'est le *typage dynamique*).

Ce n'est pas le cas dans les autres langages !!

8.2.5 Opérations binaires sur les entiers

Puisque les entiers sont constitués d'une suite 0 et de 1, on peut effectuer des opérations logiques bit à bit *et, ou, non, ou exclusif*.

Opération	`python`
"et" bit à bit	`a & b`
"ou" bit à bit	`a | b`
"ou exclusif" bit à bit	`a^b`
décalage à droite de `a` sur `b` bits	`a >> b`
décalage à gauche de `a` sur `b` bits	`a << b`

Exemple :

```
>>> bin(10), bin(12)
('0b1010', '0b1100')
>>> 10 & 12, 10 | 12, 10^12
(8, 14, 6)
>>> bin(10 & 12), bin(10 | 12), bin(10^12)
('0b1000', '0b1110', '0b110')
>>> 255 & 13, 255 & 146, 255 & 256
(13, 146, 0)
>>> 0 | 13, 0 | 256
(13, 256)
```

8.2.6 Somme d'entiers

Un microprocesseur ne manipule que des données, suites de 0 et de 1, et des opérations logiques sur ces données. Comment réaliser une addition d'entiers en n'utilisant que des opérations binaires ?

L'algorithme d'addition d'entiers binaires et similaire à l'addition d'entiers en base 10 :

Exemple : Addition : $23 + 7 = 30$.
$23 = \overline{10111}$ et $7 = \overline{111}$.

retenue :		1	1	1	
23 :	1	0	1	1	1
7 :	0	0	1	1	1
	1	1	1	1	0

Résultat : $\overline{11110} = 16 + 8 + 4 + 2 = 30$.

Remarquer que :

Dans l'addition de 2 bits `a` et `b` le résultat à reporter est `a^b`, (a xor b) et la retenue `a & b`, (a et b).

Ainsi dans l'addition d'entiers `a`,b écrits en binaire :

`a+b= (a^b) + ((a & b) << 1)`

On en déduit l'algorithme suivant pour l'addition d'entier, ne faisant appel qu'à des opérations binaires.

```
def binadd(a,b):
    while b != 0:
        a, b = a^b, (a&b) << 1
    return a
```

C'est ce procédé qu'effectue un microprocesseur pour additionner deux nombres.

8.2.7 Multiplication d'entiers

La multiplication d'entiers est analogue à la multiplication décimale, mais en plus simple (plus nécessaire de connaître ses tables de multiplications).

Exemple :

			1	0	0	1	0
		×	0	0	1	0	1
			1	0	0	1	0
1	0	0	1	0	.	.	
1	0	1	1	0	1	0	

$18 \times 5 = 90 = 64 + 16 + 8 + 2 = \overline{1011010}$.

On en déduirait l'algorithme de multiplication binaire : (Pour récupérer le bit de poids `i` d'un entier `b` il suffit de saisir `b` & (1 << i)).

```
def binmult(a,b):
    res=0
    i=0
    while (b>>i) != 0:
        if ((b & (1<<i)) != 0):
            res = binadd(res, a<<i)
        i= binadd(i,1)
    return res
```

8.2.8 Représentation normalisée des entiers signés

Pour représenter un entier signé (positif ou négatif) sur N bits ($N = 8, 16, 32, 64$), le bit de poids fort (le plus à gauche) est utilisé pour coder le signe : 0 pour positif, 1 pour négatif.

On pourrait alors utiliser les $N-1$ bits les plus faibles pour coder la valeur absolue du nombre.

Cela présenterait 2 désavantages :

1. Le zéro admettrait les deux représentations $\overline{0000\,0000}$ et $\overline{1000\,0000}$ (exemple sur 8 bits). Or la comparaison à zéro est une opération très souvent réalisée ; elle consisterait alors en 2 tests au niveau du processeur !
2. L'algorithme d'addition demeure correct entre entier positifs, mais est incorrect en général :

   $2 + 1 = \overline{0000\,0010} + \overline{0000\,0001} = \overline{0000\,0011} = 3$

   $(-2) + (-1) = \overline{1000\,0010} + \overline{1000\,0001} = \overline{1|0000\,0011} = \overline{0000\,0011} = 3$

   $(-2) + 1 = \overline{1000\,0010} + \overline{0000\,0001} = \overline{1000\,0011} = -3$

Pour y pallier on utilise la notation '*en complément à deux*' des entiers signés.

En complément à deux :

1. Un nombre positif est codé normalement sur N bits.
2. Pour un nombre négatif. On part de l'écriture binaire de sa valeur absolue sur N bits. On l'inverse (changer 0 en 1 et 1 en 0, c'est prendre le *complément à 1*), puis on ajoute 1.
3. Exemple : Pour coder sur 8 bits -9 en complément à deux :

   On part de $9 = \overline{0000\,1001}$. On l'inverse $\overline{1111\,0110}$. On ajoute 1 : $-9 = \overline{1111\,0111}$.
4. L'inverse de 0 est : $\overline{0000\,0000}$ inversé devient $\overline{1111\,1111}$, $+1$: $\overline{1|0000\,0000} = \boxed{\overline{0000\,0000}}$. Zéro a une seule représentation sur N bits.

Pour écrire -1 on prend le complément à deux de $1 = \overline{0000\,0001}$: complément à $1 = \overline{1111\,1110}$ auquel on ajoute 1 : -1 s'écrit $\overline{1111\,1111}$ tout comme $2^N - 1$ (ici $N = 8$).

Avec la notation en complément à deux on peut représenter tous les entiers signés entre -2^{N-1} et $2^{N-1} - 1$ (on peut représenter $2^{N-1} + 1 + 2^{N-1} - 1 = 2^{N-1} + 2^{N-1} = 2^N$ nombre différents sur N bits).

• La complémentation à 2 est une opération involutive : prendre le complément à deux du complément à deux d'un nombre renvoie le nombre initial :

Exemple : $-9 = \overline{1111\,0111}$ a pour complément à deux : $\overline{0000\,1000} + 1 = \overline{0000\,1001} = 9$.

Le complément à deux de $-1 = \overline{1111\,1111}$ est $\overline{0000\,0000} + 1 = \overline{0000\,0001}$.

8.2.9 Méthode simple de complémentation à deux

- De tête pour calculer un complément à 2 :

1. A partir de la valeur absolue du nombre. Par exemple 12 = $\overline{0000\,1100}$.
2. Conserver tous les nombres à partir de la droite jusqu'au premier 1 inclus : $\overline{0000\,1\mathbf{100}}$.
3. Inverser tous les autres : $\overline{1111\,0\mathbf{100}}$
4. Complément à deux de 12 : $\overline{1111\,0011} + 1 = \overline{1111\,0100}$.

8.2.10 Exemple sur 8 bits

Exemple : sur 8 bits :

$\overline{00000000}$	0
$\overline{00000001}$	1
$\overline{00000010}$	2
⋮	⋮
$\overline{01111111}$	127
$\overline{10000000}$	-128
$\overline{10000001}$	-127
⋮	⋮
$\overline{11111101}$	-3
$\overline{11111110}$	-2
$\overline{11111111}$	-1

-128	-127	···	-2	-1	0	1	···	127
$\overline{10000000}$	$\overline{10000001}$	···	$\overline{11111110}$	$\overline{11111111}$	$\overline{00000000}$	$\overline{00000001}$	···	$\overline{01111111}$

En complément à 2, sur 8 bits, on représente l'entier relatif $n \in [[-128; +127]]$ par les 8 bits de poids faibles de $2^8 + n$:

Exemple, pour représenter :
-128 : $2^8 + (-128) = 256 - 128 = 128 = \overline{10000000}$
+127 : $2^8 + 127 = 256 + 127 = \overline{1|01111111}$ -> $\overline{01111111}$

8.2.11 Addition et multiplication

• **L'addition fonctionne avec des nombres en complément en deux.** • Exemple :

$(-2) + (-1) = \overline{1111\,1110} + \overline{1111\,1111} = \overline{1|1111\,1101} = \overline{1111\,1101} = -3$

$(-2) + 1 = \overline{1111\,1110} + \overline{0000\,0001} = \overline{1111\,1111} = -1$. Corrects

• **L'algorithme de multiplication fonctionne** ; exemple :

	0	0	0	0	0	1	1	0	(6)
×	1	1	1	1	1	0	1	1	(−5)
						1	1	0	
					1	1	0	.	
				0	0	0	.	.	
			1	1	0	.	.	.	
		1	1	0	.	.	.	.	
	1	1	0	.	.	.	.	.	
*	1	0	.	.	.	.	.	.	
*	0	.	.	.	.	.	.	.	
*	1	1	1	0	0	0	1	0	(-30)

8.2.12 Code de conversion en Compléments à deux

• En fait représenter $n \in [[-2^{N-1}; 2^{N-1}[[$ consiste à représenter le nombre n *modulo* 2^N en binaire (non signé).

On en déduit le code d'une fonction convertissant un nombre relatif n en notation binaire signée :

```
def binaire(n):
    res = ''
    while n > 0:
        res = str(n%2) + res
        n = n//2
    return res

def binaireSigne(n,N):
    """Représente l'entier relatif n en binaire signé sur N bits"""
    if -2**(N-1) <= n < 2**(N-1):
        n = n % 2**N    # n modulo 2**N
        res = binaire(n)
        k = N-len(res)
        res = k*'0' + res    # Rajouter des 0 devant pour arriver à N
bits
        return res
```

```
In[1] : binaireSigne(-128,8)
Out[1] : '10000000'
```

• Conversion inverse (binaire signé sur N bits vers nombre relatif en décimal) :

```
def decimal(s): # Converion binaire (str) -> decimal (int)
    n = 0
    for char in s:
        n *= 2
        n += int(char)
    return n

def binS2dec(s,N):
    """Donne le nombre relatif codé par s en binaire signé sur N
bits"""
    n = decimal(s)
    if n < 2**(N-1):     # Cas positif
        return n
    elif 2**(N-1) <= n < 2**N:    # Cas négatif
        return n-2**N
```

```
In[2] : binS2dec('10000000',8)
Out[2] : -128
```

8.2.13 Représentation des nombres en `python`

Dans les langages de programmation les plus courants les nombres entiers sont représentés sous cette forme, habituellement signés (en complément à deux), ou lorsque le type existe en entiers non signés.

La plupart des langages de programmation proposent 2 formats d'entiers signés :

1. Le type entier court, '`int`' ou encore '`short int`' : les entiers sont codés signés sur N bits, habituellement $N = 32$ ou 64 sur des équipements plus récents. On peut représenter efficacement tous les entiers signés sur la plage de valeurs de $-2^{31} = -2\,147\,483\,648$ à $2^{31} - 1 = 2\,147\,483\,647$ ($N = 32$, 4 octets).
2. Le type entier long, '`long`' ou encore '`long int`' : les entiers sont codés signés sur $2N$ bits. Lorsque $N = 32$ on peut représenter tous les entiers signés entre $-2^{63} = -9\,223\,372\,036\,854\,775\,808$ à $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,808$.
3. Certains admettent aussi un type 'unsigned' ou 'uint, unint'pour coder des entiers non signés ; la limite de dépassement est doublée quand le nombre de bits augmente de 1 : $2^{31} \mapsto 2^{32} = 2^{1+31} = 2 \times 2^{31}$.

En `python` on dispose de deux formats d'entiers, courts et long. Mais :

1. `python` pratiquant le typage dynamique, c'est l'interpréteur qui bascule automatiquement en cas de dépassement d'un format à l'autre. Pour l'utilisateur tout est transparent, et corresponds au seul type '`int`'...
2. en format court, le nombre est codé signé (en complément à deux) sur 32 ou 64 bits (sur un ordinateur ancien 32 bits/ récent 64 bits).
3. en format long, le nombre d'octet alloué est fixé par l'interpréteur selon le besoin et n'a pas de limité prédéfinie : dans la pratique aucun calcul raisonnable ne devrait aboutir à une erreur de dépassement de capacité. Cependant il existe toujours un capacité limitée par les capacités de stockage de l'ordinateur.

Mon interpréteur (64 bits) stocke en format court tous les entiers signés de $-9\,223\,372\,036\,854\,775\,808$ à $9\,223\,372\,036\,854\,775\,807$.

8.3 Représentation normalisée des nombres à virgule flottante

8.3.1 Représentation des réels

Dans la plupart des langages informatiques les nombres réels sont représentés par des nombres à virgule flottante. La représentation des nombres à virgule flottante est normalisée par la norme IEEE 754.

La plupart des langages proposent deux formats de nombres à virgule flottante : les nombres en précisions simple, habituellement codés sur $N = 32$ bits ou 64 bits, et les nombres en double précisions codés sur $2N = 64$ ou 128 bits.

Un nombre x en virgule flottante est constitué de trois éléments :

1. son signe (ϵ),
2. son exposant (e),
3. sa mantisse (m) avec $1 \leqslant m < 2$, sauf pour 0 auquel cas $m = 0$ et pour l'exception des écritures dénormalisées.

tels que $\boxed{x = \epsilon.m.2^e}$.

1. un bit est réservé pour le signe, 0 : positif, 1 :négatif.
2. 8 bits (11 bits en 64b) sont réservés pour l'exposant : code un exposant entre -127 et 128 (-1023 à 1024 en 64b). On ne code pas e en complément à deux, mais $e + 127$ en binaire ($e + 1023$ en 64b), c'est à dire décalé de $2^{N-1} - 1 = 127$ (en 32 bits, 1023 en 64b).
3. 23 bits (52 bits en 64b) sont réservés pour la mantisse.

8.3.2 Nombres à virgule flottante

Exemple : Avec $x = 125,75$, on a :

$x = 125,75 = 64 + 32 + 16 + 8 + 4 + 1 + \frac{1}{2} + \frac{1}{4} = \overline{1111101,11} = \overline{1,11110111}.2^6$.

Le signe est positif : 0,

la mantisse est $\overline{1,11110111}$ mais sera représenté $\overline{11110111}$, sans le préfixe '1,'. Les nombres sont derrière la virgule, et le nombre devra être aligné à gauche.

L'exposant est 6, ce qui donne $6 + 127 = 133 = 128 + 4 + 1 = \overline{1000\,0101}$

Ainsi en simple précision, 125,75 est représenté sur 32 bits :

$$\underbrace{0}_{signe}\ \underbrace{10000101}_{exposant}\ \underbrace{11110111\,000000000000000}_{mantisse}$$

Réciproquement, cherchons le nombre flottant dont l'écriture sur 32 bits est :

$$\overline{1\,10000110\,10110000000000000000000}$$

Le signe est négatif 1, et l'exposant est $\overline{10000110} = 128 + 4 + 2 = 134$ donne $134 - 127 = 7$. L'exposant est 7.

La mantisse est $\overline{10110000000000000000000}$ est égale à :

$$1 + \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} = 1,6875.$$

Le nombre est $-1,6875 \times 2^7 = \boxed{-216,0}$.

8.3.3 Ecriture dénormalisée (hors programme)

Il y a 4 exceptions à l'écriture normalisée (écritures dénormalisées, hors programme) :

1. Si exposant décalé et mantisse sont tous deux nuls : le nombre est 0. Attention il y a deux zéros :

$$\underbrace{0}_{signe}\ \underbrace{00000000}_{exposant}\ \underbrace{00000000000000000000000}_{mantisse} = 0^+$$

$$\underbrace{1}_{signe}\ \underbrace{00000000}_{exposant}\ \underbrace{00000000000000000000000}_{mantisse} = 0^-$$

2. Si l'exposant décalé est 0 et la mantisse est non nulle on n'ajoute non plus 1 mais 0 à la mantisse et on prend l'exposant $e = -126$ ($e = -1022$) plutôt que $e = -127$ ($e = -1023$).
3. Si l'exposant décalé est $2^N - 1 = \overline{1111\,1111}$ et la mantisse est nulle : on obtient les 2 infinis selon le signe :

$$\underbrace{0}_{signe}\ \underbrace{11111111}_{exposant}\ \underbrace{00000000000000000000000}_{mantisse} = +\infty$$

$$\underbrace{1}_{signe}\ \underbrace{11111111}_{exposant}\ \underbrace{00000000000000000000000}_{mantisse} = -\infty$$

4. Si l'exposant décalé est $2^N - 1 = \overline{1111\,1111}$ et la mantisse est non nulle on obtient un `NaN` : Not a Number.

8.3.4 Limitation de la représentation des nombres à virgule flottante

• Limitations de la représentation des nombres en virgules flottantes sur 64 bits :

En virgule flottante, on ne peut par représenter de nombre supérieur ou égal à 2^{1024} :

```
In[1] : 2.0**1023
Out[1] : 8.98846567431158e+307
In[2] : 2.0**1024
OverflowError : 'Result too large'
In[3] : 2**1024
Out[3] : 179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

• Limitations de la représentation des nombres flottants sur 64 bits :

En virgule flottante, on ne peut par représenter de nombres très petits :

```
In[4] : 2**(-1075)
Out[4] : 0.0
In[5] : 2**(-1074)
Out[5] : 5e-324    # La plus petite valeur sur 64 bits !
```

ce dernier nombre s'écrit : `0 0··· 0 0··· 01` (-1074 = -1022-52, écriture dénormalisée (cas 2)).

• Comparer deux nombres flottants (`==`) a en général peu de sens :

```
In[5] : 1+2**-52 == 1
Out[5] : False
In[6] : 1 +2**-53 == 1
Out[6] : True
In[7] : 2**-53 == 0
Out[7] : False
```

• Il faut prendre garde aux problèmes de dépassements de capacités dans la représentation informatique des nombres !