CH2 LE CODAGE DES NOMBRES

1 Codage d'un entier naturel en binaire

Sous ses grands airs, un ordinateur ne comprend finalement que deux choses : « le courant passe », qu'on notera 1 et « le courant ne passe pas », qu'on notera 0. C'est ce qu'on appelle le binaire.

Exemple :

Comment pourrait-on, pour communiquer avec un ordinateur, « noter » les entiers suivants : 0 ; 1 ; 2 ; 5 ; 45 ?

1.1 Représentation binaire

Habituellement, nous représentons les valeurs entières dans le système décimal, on dit aussi en base 10. Nous utilisons les dix chiffres de 0 à 9. La position des chiffres définit la valeur associée à ce chiffre. Par exemple, 5402 est compris comme 5 milliers, 4 centaines, 0 dizaine et 2 unités :

$$5402 = 5 \times 1000 + 4 \times 100 + 0 \times 10 + 2 \times 1$$

Les différents chiffres correspondent aux puissances successives de 10 :

$$5402 = 5 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 2 \times 10^0$$

A retenir :

L'information numérique, qu'il s'agisse de valeurs entières, de textes, d'images, ou de sons est en fin de compte représentée uniquement par des suites de 0 et de 1. On parle de bit : un bit peut prendre deux valeurs, 0 ou 1.

*BIT = **BI**nary digi**T***

Le système binaire permet d'écrire les valeurs entières en n'utilisant que les deux chiffres 0 et 1. On utilise alors la base 2.

De même que pour la base 10, les positions des chiffres sont associées aux puissances successives de 2

$2^0 = 1$; $2^1 = 2$; $2^2 = 4$; $2^3 = 8$; $2^4 = 16$; $2^5 = 32$; $2^6 = 64$; *etc*.

Ainsi la valeur entière qui correspond à la représentation binaire 101010 est

$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 42$

Il nous faut pouvoir indiquer que 101010 est une représentation binaire et non une représentation décimale, qui serait comprise *cent un mille dix* (ou encore une représentation dans une autre base...).

On notera par exemple 0b101010 ou 101010_2, ou encore 101010.

On distingue donc les **valeurs entières** (les entiers) et leur représentation.

A retenir :

À une valeur entière donnée est associée une représentation décimale, mais aussi une représentation binaire.

Dans une représentation binaire d'un nombre, le bit le plus à gauche est appelé **bit de poids fort**, celui le plus à droite **bit de poids faible**.

Exemples :

- *Expliquez ce que peut signifier le signe '=' dans l'équation suivante : 10 = 2, et pourquoi on préférera écrire 0b10 = 2*
- *Donnez les valeurs entières représentées par 0b0100, 0b10101, 0b101, 0b0101 et 0b00101.*
- *Comparez les valeurs entières représentées par 0b11 et 0b100, 0b111 et 0b1000.*
- *Combien d'entiers peut-on coder avec un octet ? (un octet est une série de 8 bits, c'est-à-dire une série de 8 « 0 ou 1 ») ?*
- *Quelle est la représentation binaire de 14 ?*

Passer d'un nombre en base dix à sa représentation binaire

Méthode : On divise le nombre donné par 2, on note le quotient et le reste (qui est 0 ou 1). On recommence en divisant le quotient par 2 … On s'arrête lorsque le quotient vaut 0.

La représentation binaire du nombre est donnée par la suite des restes, de bas en haut.

Exemple : on souhaite écrire 135 en base 2 :

Nombre	*Quotient de la par 2*	*Reste*
135	*67*	*1*
67	*33*	*1*
33	*16*	*1*
16	*8*	*0*
8	*4*	*0*
4	*2*	*0*
2	*1*	*0*
1	*0*	*1*

L'écriture binaire de 135 est donc 0b10000111. Vous pouvez vérifier en transformant ce nombre binaire en décimal.

Exemples :

- *Donner la représentation binaire de 78 et de 561.*

- *Sur n bits, combien d'entiers naturels peut-on coder ?*

ACT1 Conversion

1.2 Opérations en binaire

Les opérations utilisées en décimal sont valables aussi en binaire : addition, multiplication, etc...

Voir CH1 Booléens pour un exemple d'additionneur.

A retenir

L'**addition** binaire fonctionne comme l'addition décimale, en utilisant le fait que $1+1=10$ revient à « poser zéro et retenir 1 ».
On constate que l'addition de deux bits A et B donne A XOR B avec une retenue valant A ET B.

Multiplier par deux se fait en décalant chaque chiffre d'un cran à gauche et en insérant un zéro à la fin. C'est le même principe que pour multiplier par 10 un nombre décimal.
La **division** entière par deux se fait en décalant chaque chiffre d'un cran à droite, le chiffre de droite étant le reste supprimé.

Exemples :

- *Additionner les nombres binaires 0110 et 1010 (vérifier en les traduisant en nombres décimaux).*
- *Multiplier : deux fois onze*
- *Diviser : onze divisé par deux*

ACT2 Opérations bit-à-bit

1.3 Et en python ?

Binaire

Par défaut, en Python, les nombres entiers sont en base 10. Pour manipuler des séquences de bits, on utilise la notation 0b….

```
In [1]: 0b01001101
Out[1]: 77
```

Inversement, on peut convertir une valeur entière en base 10 vers la base 2 à l'aide de la fonction bin() :

```
In [2]: bin(43)
Out[2]: '0b101011'
```

Exemple :

- *Vérifiez les résultats précédents et d'autres valeurs dans une console python.*

Hexadécimal

La base 16, dite hexadécimale, est fréquemment utilisée. Pour pouvoir écrire 16 « chiffres » hexadécimaux, on utilise les chiffres de 0 à 9, puis les lettres A à F (A correspondant à 10 et F à 15).

Le codage fonctionne sur le même principe que pour une base 2 ou 10 :

Le nombre **3A5** vaut $3\times16^2+10\times16+5\times16^0=933$.

La base 16 est souvent utilisée pour simplifier l'écriture de nombres binaires. En effet, on peut aisément passer d'un nombre en base 2 à un nombre en base 16 en regroupant les chiffres binaires par 4 comme ci-dessous :

$$\overbrace{1\,0\,0\,1}^{A}\overbrace{0\,1\,0\,1}^{5}\overbrace{1\,1\,1\,1}^{F}\overbrace{0\,0\,1\,1}^{3}$$

Le nombre binaire 1001 0101 1111 0011 se code donc A5F3 en hexadécimal.

La transformation inverse se fait en codant chaque chiffre hexadécimal en binaire sur 4 bits.

Remarques :

- *On utilise parfois la notation* $\dots_b$ *pour indiquer que le nombre (représenté ici par …) est codé en base* b. *Ceci permet d'éviter la confusion entre* 101_2, 101_{10} *et* 101_{16} *!*
- *Python permet de manipuler les nombres hexadécimaux à l'aide de la notation 0x et de transformer un nombre en hexadécimal à l'aide de la fonction hex().*

```
In [1]: 0xDF40E
Out[1]: 914446
```

```
In [3]: bin(0xA4F2)
Out[3]: '0b1010010011110010'
```

```
In [4]: hex(1759)
Out[4]: '0x6df'
```

```
In [5]: hex(0b10101)
Out[5]: '0x15'
```

Exemple :

Vérifier « à la main » les affichages de la première colonne ci-dessus.

2 Les entiers relatifs en binaire (ou entiers *signés*)

Nous savons maintenant coder les entiers naturels en binaire. On pourrait imaginer, pour coder les relatifs, réserver le premier bit du nombre au signe : 0 pour positif, 1 pour négatif par exemple.

→ Avec cette méthode, coder en binaire, sur un octet, les nombres 15 et -15. Additionner ces deux nombres en binaire et transformer le résultat en nombre décimal. Que constatez-vous ?

Cette méthode n'est donc pas pertinente. Pour coder un entier relatif, on utilisera la méthode « du complément à 2 ».

→ Additionner, après codage binaire sur 4 bits, les nombres 15 et 1.

Sur 4 bits, le résultat est donc 0. Pour la machine, 1111 est l'opposé de 0001, donc 1111 représente -1.

On souhaite déterminer l'opposé de 5 (codé en binaire sur 4 bits 0101).

→ Compléter l'addition suivante :

	0	1	0	1
+	…	…	…	…
1	0	0	0	0

Pour la machine, sur 4 bits, l'opposé de 5 est donc codé par 1011.

<u>Méthode générale</u>

De manière générale, avec *n* bits, la représentation machine d'un entier relatif r est l'écriture binaire de la différence $2^n - r$. Cette représentation s'appelle le complément à 2^n, souvent abrégée en complément à 2.

Pratiquement, pour trouver la représentation d'un entier négatif r, on prend l'écriture binaire de -r (qui est un entier positif), on inverse les bits de cette écriture et on ajoute 1. Inverser les bits et ajouter 1 sont des opérations simples pour la machine. Dans toutes ces écritures, le nombre de bits est fixé.

On effectue l'opération inverse pour trouver l'entier relatif correspondant à un nombre binaire commençant par un.

Exemple :

- *On souhaite coder -12, sur 6 bits.*

Codage de 12 en binaire sur 6 bits : *001100*

On inverse chaque bit : *110011*

On ajoute 1 : *110100*

La représentation de -12 en complément à 2 sur 6 bits est donc 110100.

- *Déterminer la représentation de -28 en complément à 2 sur 8 bits. Vérifier qu'après codage en binaire, on a bien -28+28=0*
- Quel est l'entier relatif codé en complément à 2 sur 8 bits par 10011001 ?
- Sur 1 octet (8 bits), quels sont les plus petit et plus grand entiers relatifs que l'on peut coder ? Quelle est la caractéristique des entiers positifs ? Celle des entiers négatifs ?

A retenir :

Avec n bits, on peut représenter les entiers compris entre -2^{n-1} et $2^{n-1} - 1$ .Les nombres négatifs ont tous le bit de poids fort égal à 1.

La représentation en machine est fixée par le nombre de bits utilisés. Ainsi avec un octet, on peut représenter les entiers naturels de 0 à 255. On peut également représenter les entiers relatifs de à $-2^7 = -128$ à $2^7 - 1 = 127$.

Remarque :

*Avec certains langages, le nombre d'octets avec lequel on travaille sur les entiers peut être précisé. En Python, c'est Python qui gère la taille, a priori illimitée. Les entiers sont représentés par le type **int** (integer). Si un nombre dépasse la taille maximale que peut gérer le processeur, ce nombre est découpé en deux ou plusieurs parties, et Python s'occupe des différentes opérations à effectuer. Cela prend alors plus de temps et d'espace en mémoire …*

Exécuter le script ci-contre Que remarque-t-on ? Expliquer.

```python
from time import time
start=time()
for i in range(50000):
    a=2**i
print(time()-start)

start=time()
for i in range(50000):
    a=3**i
print(time()-start)
```

3 Les nombres réels

ACT3 Codage des flottants

On peut, par analogie avec les nombres décimaux, écrire un nombre à virgule en notation binaire en utilisant les puissances négatives de 2 :

Par exemple :

$(11{,}0101)_2 = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$(11{,}0101)_2$ = 2 + 1 + 0 + 0,25 + 0,0625 = 3,3125

Il devient vite compliqué d'utiliser cette méthode pour les nombres très petits ou très grands.

En notation décimale, on utilise la notation scientifique : pour écrire un nombre x , on précise son signe s, puis un nombre décimal m (appelé **mantisse**) et un entier relatif n (appelé **exposant**) :

$x = s\,m \times 10^n$.

Exemple : $173{,}95 = +1{,}7395.10^2$. *On peut aussi écrire* $173{,}95 = +17{,}395.10^1$.

A Retenir

On appelle cette écrire **virgule flottante**. La virgule « flotte » de droite à gauche, on peut la placer où on le souhaite.

La norme IEEE-754 définit l'adaptation de cette méthode à la base 2.

Tout nombre réel peut être représenté sous la forme $x = (-1)^s m.2^E = (-1)^s m.2^{n-(N-1)}$, où s correspond au signe de x ($s=0$ pour un nombre positif, $s=1$ pour un nombre négatif), m (la mantisse) est un réel de l'intervalle `[1;2[`, $n$est un entier relatif tel que $E = n - (N-1)$, avec N le niveau de précision de la machine (32 ou 64 bits sur les machines modernes).

Cette norme définit les points suivants :

Encodage	Signe s	n	Mantisse m	Valeur x
32 bits	1 bit	8 bits	23 bits	$(-1)^s m 2^{(n-127)}$
64 bits	1 bit	11 bits	52 bits	$(-1)^s m 2^{(n-1023)}$

Soit la représentation suivante sur 32 bits :