Chapitre 6 : Les entiers

1. Représentation des nombres entiers

1.1.Notion de base d'écriture

D'une manière générale, nous écrivons les nombres en utilisant la notation positionnelle en base dix. Ceci signifie que nous utilisons dix chiffres $0,...,9$ et que les positions relatives de ces chiffres (tout particulièrement celle du zéro) sont capitales. Ainsi, tout nombre entier N peut s'écrire :

$$N = c_n c_{n-1} ... c_0 = c_n \times 10^n + c_{n-1} \times 10^{n-1} + ... + c_0$$

Exemple : $357 = 3 \times 100 + 5 \times 10 + 7$
Le rôle du 0 est primordiale dans l'écriture positionnelle sinon comment distinguer 270 et 27 ou 7803 et 783.

Nous pouvons ainsi envisager l'écriture positionnelle d'un nombre entier N dans une base b où b est un entier quelconque, $b \geq 2$, en utilisant b chiffres, de 0 à $b-1$. On aura alors :

$$N = \left(a_n a_{n-1} ... a_0\right)_b = a_n \times b^n + a_{n-1} \times b^{n-1} + ... + a_1 \times b + a_0$$

Si $b = 2$, on parle alors d'écriture binaire. Dans ce cas, les chiffres utilisés se limitent à 0 et 1. Ceux-ci sont appelés bit qui vient de l'anglais « binary digit ».

On peut noter que si $b > 10$, se pose le problème du nombre de chiffres puisque nous n'en disposons que de dix dans notre écriture. On viendra ainsi compléter avec des lettres $A, B, C, ...$ C'est notamment le cas pour le système hexadécimal ($b = 16$) pour lequel les chiffres de 0 à 15 sont notés $1,2,...,9,A,B,C,D,E,F$.

1.2.Choix du système binaire

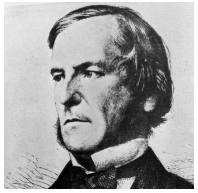
Gottfried Wilhelm Leibniz - 1646 - 1716

Le mathématicien Leibniz est considéré comme à l'origine du développement du système binaire. En effet, c'est lui qui a notamment précisé comment calculer en binaire, en particulier pour les quatre opérations de base.

John Von Neumann - 1903 - 1957

Alors que certains des premiers ordinateurs calculaient en utilisant le système décimal, les travaux de Von Neumann dans les années 1940 ont fait du système binaire la norme.

Les raisons de l'utilisation du binaire dans le domaine de l'informatique sont multiples :
- Avec une machine utilisant des circuits électroniques, il est plus simple de ne considérer que deux états, un état haut et un état bas, chacun correspondant à un chiffre 0 ou 1.
- Les deux valeurs 0 et 1 peuvent être associées aux valeurs Vrai ou Faux d'une propriété. Il est ainsi possible d'effectuer un lien entre l'arithmétique et le calcul logique.



Georges Boole - 1815 - 1864

Les valeurs 0 et 1 sont ainsi appelées variables booléennes en hommage au mathématicien Georges Boole qui a développé au $19^{ème}$ siècle une algèbre liée à la théorie des ensembles. Les trois opérations logiques d'addition, de multiplication et négation correspondent à la réunion, l'intersection et le complémentaire sur l'ensemble des parties d'un ensemble de référence.

- En base deux, une multiplication est guère plus compliquée qu'une addition.
- Le passage de la base 2 à la base 2^p est aisé. Il suffit pour cela de regrouper les bits par groupe de p.
- Dans une addition en base b, les retenues éventuelles valent toujours 1. Mais, dans une multiplication, il peut y avoir $b-1$ retenues possibles. En binaire, la seule retenue possible vaut 1 comme pour l'addition.

1.3. Vocabulaire du système binaire

- On appelle mot une suite composée de 0 et de 1. En binaire, les chiffres 0 et 1 sont appelés lettres et l'ensemble $\{0;1\}$ l'alphabet. Le mot est l'unité de base pour la machine ou le processeur.
- La taille d'un mot s'exprime en bits. Un mot de huit bits est appelé octet. Pour un mot de seize bits (soit deux octets), on parle de word, pour 32 bits (quatre octets), dword (double word), pour 64 bits (quatre octets), qword (quadruple word). Suivant les architectures des ordinateurs, la taille du mot ou du word change. Actuellement, les machines les plus fréquentes travaillent avec des mots de 64 bits, les plus anciennes avec des mots de 32 bits.
- ASCII : ce code introduit dans au début des années 70 permet d'associer à un caractère de certains langages un mot codés sur 8 bits (octet). Il est ainsi possible de représenter 256 caractères ($2^8 = 256$) en particulier ce que l'on trouve sur les touches d'un clavier.

2. Codage des entiers naturels

2.1.Mot de taille fixe

Habituellement, le nombre de chiffres utilisés pour écrire un nombre, quelle que soit la base, dépend de la valeur de ce nombre et de la base utilisée. En base dix, 12 s'écrit avec deux chiffres tout comme 68 et 102 avec trois chiffres. En revanche, en binaire, ces mêmes nombres s'écrivent respectivement 1100, 1000100 et 11000100.
La mémoire d'un ordinateur peut être comparée à une feuille quadrillée pour laquelle on vient écrire dans chaque carreau 1 ou 0. Se pose alors le problème du nombre de bits composant le mot associé au nombre codé. En effet, si la taille du mot n'est précisée, une même série de bits peut conduire à plusieurs interprétations.
Exemple : pour le code 11000100, on pourrait entre autres lire
- 1100 et 0100 soit les deux entiers 12 et 4
- 110 et 00100 soit les deux entiers 6 et 4
- 11000 et 100 soit les deux entiers 24 et 4

Pour lever toute ambiguïté, il parait incontournable de fixer le nombre de chiffres qui composent un mot. Le nombre d'entiers pouvant être codés dépend ainsi du nombre de bits par mot. Par exemple, pour des mots de quatre bits, on pourra codés les nombres entiers allant de 0 (0000) à 15 (1111) soit au total $2^4 = 16$ entiers. Les calculs étant effectués par un processeur, la taille des mots dépend des capacités de celui-ci.

2.2.Changement de base

Partant d'un nombre donné en base dix, on souhaite changer de base pour, par exemple, l'écrire en base deux (binaire). Afin d'établir la méthode qui permettra d'obtenir cette nouvelle écriture, on peut déjà détailler ce qu'est l'écriture en base dix et comment elle est obtenue.

Pour le nombre 8050, on peut écrire sous forme de tableau :

$\ldots$	$10000 = 10^4$	$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$
$\ldots$		8	0	5	0
$\ldots$	0	8	0	5	0

Ceci permet de constater une fois de plus que l'utilisation du chiffre 0 est déterminante.

Si on considère maintenant le nombre 5326, on peut remarquer que les chiffres qui composent ce nombre sont les restes obtenus dans les divisions euclidiennes successives du nombre par 10 jusqu'à obtenir un quotient nul.

$$\begin{array}{r|l} 5326 & 10 \\ \cline{2-2} \mathbf{6} & 532 \end{array} \quad \begin{array}{r|l} & 10 \\ \cline{2-2} \mathbf{2} & 53 \end{array} \quad \begin{array}{r|l} & 10 \\ \cline{2-2} \mathbf{3} & 5 \end{array} \quad \begin{array}{r|l} & 10 \\ \cline{2-2} \mathbf{5} & 0 \end{array}$$

On en déduit qu'il en sera de même pour déduire l'écriture en base deux. Par exemple, pour le nombre 11, on aura :

$$\begin{array}{r|l} 11 & 2 \\ \cline{2-2} \mathbf{1} & 5 \end{array} \quad \begin{array}{r|l} & 2 \\ \cline{2-2} \mathbf{1} & 2 \end{array} \quad \begin{array}{r|l} & 2 \\ \cline{2-2} \mathbf{0} & 1 \end{array} \quad \begin{array}{r|l} & 2 \\ \cline{2-2} \mathbf{1} & 0 \end{array}$$

Tout comme pour la base dix, on peut écrire le résultat à l'aide d'un tableau.

$\ldots$	$16 = 2^4$	$8 = 2^3$	$4 = 2^2$	$2 = 2^1$	$1 = 2^0$
$\ldots$		1	0	1	1

Le bit le plus à gauche est appelé bit de poids fort tandis que celui le plus à droite est appelé bit de poids faible.
En informatique, tout est écrit en bas deux mais, souvent, les données sont regroupées en base huit ou seize pour plus de lisibilité.

Base deux	$\longleftrightarrow$	Base seize
1101	$\longleftrightarrow$	D
0011	$\longleftrightarrow$	3
$1101\,0011$	$\longleftrightarrow$	D3

Écrire le nombre 147 en base deux puis en base seize.

2.3. Représentation machine et opérations élémentaires

Un entier naturel s'écrit sans signe ; on parle d'entier non signé. Ainsi, l'ensemble des bits utilisés pour le codage de l'entier servent au codage de sa valeur.

Avec n bits, on peut représenter les entiers de 0 à 2^n-1, soit un total de 2^n valeurs. Pour tout nombre entier $k \in [\![0\,;2^n-1]\!]$, il existe n entiers $\{b_0\,;\ldots;b_{n-1}\} \in [\![0\,;1]\!]^n$ tels que :

$$k=\sum_{i=0}^{n-1} b_i \times 2^i$$

En général, un entier est codé sur 4 octets (32 bits) ou sur 8 octets (64 bits).

Il est à souligner que le fait qu'un entier soit codé sur un mot de taille fixe peut dans certains cas poser problème. Ceci peut être illustré en examinant le processus d'addition de nombres binaires. Pour cela, on utilise les résultats suivants :

- $0+0=0$
- $0+1=1$
- $1+1=10$

Lorsqu'on doit effectuer l'addition de deux nombres binaires, on peut poser l'addition comme dans l'exemple ci-dessous :

	1	1	0	1	treize
+	1	0	0	1	neuf
1			1		retenues
1	0	1	1	0	vingt-deux

Toutefois, on peut remarquer que, dans le cas d'un codage sur 4 bits, l'opération précédente retourne un résultat erroné. En effet, le bit de poids fort du résultat est perdu et le résultat annoncé et 0110 soit 6 et non 22.

Calculer la somme $100110 + 001101$ en posant l'addition (les nombres sont écrits en binaire). Traduire le calcul en décimal.

Concernant la multiplication, on peut effectuer les remarques suivantes :

- $011 \times 1 = 011$ $(3 \times 1 = 3)$
- $011 \times 10 = 0110$ $(3 \times 2 = 6)$
- $011 \times 100 = 01100$ $(3 \times 4 = 12)$
- ...

On déduit de ces résultats que multiplier en binaire un nombre par une puissance de deux 2^n revient à décaler l'ensemble des bits de ce nombre de n bits vers la gauche en ajoutant n zéros.

Calculer la produit 1001×111 (les nombres sont écrits en binaire). Traduire le calcul en décimal.

3. Codage des entiers relatifs

3.1.Préambule

Pour mieux cerner le problème, considérons que l'ordinateur utilise l'arithmétique décimal et des nombres écrits, avec quatre chiffres. Il est donc ainsi possible d'écrire 10000 entiers naturels compris entre 0 et 9999. Supposons que l'on souhaite désormais représenter également des nombres entiers négatifs.

On pourrait choisir de représenter les nombres compris entre -0 et -9999 d'une part et $+0$ et $+9999$ d'autre part. Toutefois, on remarque qu'il faudra un symbole supplémentaire et il y a deux zéros. Si on souhaite rester sur des mots de quatre chiffres, un chiffre est monopolisé pour le signe et il n'en reste plus que trois pour la valeur absolue du nombre.

On peut imaginer translater tous les nombres x compris entre -5000 et 4999 en effectuant l'opération $x+5000$. En effet, si on tient pas compte du bit supplémentaire lié à la retenue, on a que $9999+0001=0000$ ce qui entraîne que $0000-0001=9999$, $0000-0002=9998$ ... On pourrait donc dire que les nombres positifs sont représentés par les nombres de 0000 à 4999 et les négatifs par les nombres de 5000 à 9999. Les nombres négatifs x seraient ainsi représenter par les nombres $x+10^4$.

Remarque : cette technique s'inspire de la lecture des angles sur le cercle trigonométrique. Ce dernier peut en effet être gradué de 0 à $359°$ ou de -180 à $179°$. On peut remarquer que les angles négatifs α du second système de graduation sont équivalents à $\alpha+360°$ dans le premier.

3.2.Complément à deux

Si pour la lecture d'angles, la périodicité est de $360°$, elle est de 10^4 pour des mots de quatre chiffres en décimal et de 2^n pour des mots de n bits en binaire.
Exemple : Sur quatre bits, on aura $1111+0001=0000$

Soit r un entier négatif. La représentation machine de r sera ainsi l'écriture binaire de la différence $2^n-(-r)=2^n+r$. Cette représentation est appelée complément à 2^n souvent abrégé en complément à deux.
En considérant des mots de six bits, écrire en complément à deux le nombre -12.

Dans la pratique, on peut remarquer que, pour écrire le nombre r en complément à deux, il suffit d'écrire en binaire le nombre $-r$, d'inverser tous les bits c'est-à-dire que 0 devient 1 et vice versa (complément à un) puis d'ajouter 1.

Reprendre l'exemple de l'écriture de -12 sur six bits avec cette nouvelle méthode.

Par conséquent, en complément à deux, sur n bits, on peut coder les nombres compris entre -2^{n-1} et $2^{n-1}-1$. Ainsi, avec quatre octets (32 bits), on peut écrire les entiers naturels de 0 à $2^{32}-1$ et donc représenter les entiers relatifs compris entre -2^{31} et $2^{31}-1$.
Le tableau ci-dessous donne un résumé de la représentation d'entiers signés sur trois bits en complément à deux.

codage binaire	000	001	010	011	100	101	110	111
entiers non signés	0	1	2	3	4	5	6	7
entiers signés	0	1	2	3	-4	-3	-2	-1

On peut alors effectuer les remarques suivantes :
- Les entiers négatifs ont alors tous leur bit de poids fort égal à 1.
- Les écritures des entiers naturels de 0 à $2^{n-1}-1$ représentent les entiers naturels positifs ou nuls correspondant.
- Les écritures des entiers naturels de 2^{n-1} à 2^n-1 représentent les entiers relatifs négatifs compris entre -2^{n-1} et -1.
- Quel que soit le nombre n de bits utilisés, le nombre -1 s'écrira toujours comme une suite de 1.

4. Entiers multi-précision

4.1.Gestion des entiers longs

La plupart des processeurs calculent avec des nombres binaires de taille limitée à 32 ou 64 bits. Avec certains langages, le nombre d'octets avec lequel on travaille sur les entiers peut être précisé. Les calculs sont alors gérés directement par le processeur.

Lorsqu'on programme en langage python, nous n'avons pas, a priori, à nous soucier de la représentation des entiers. C'est le langage qui gère la taille, illimitée dit-on, mais limitée quand même par les capacités de la mémoire. Lorsque la taille d'un entier dépasse la taille maximale utilisable par le processeur, ce nombre est découpé en deux ou plusieurs parties et Python s'occupe alors des différentes opérations à effectuer. Nonobstant le ralentissement du programme, il faut aussi comprendre que cela va solliciter encore plus de mémoire vive.

Afin de mettre en évidence le ralentissement des opérations, on considère deux boucles effectuant le même travail mais avec une condition initiale différente. Pour la première, le résultat n reste nul alors que la seconde génère des entiers longs. On constate que le temps d'exécution est lourdement impacté par la gestion des entiers longs dans la deuxième boucle.

```
from time import time

top=time()
n=0
for i in range(1800):
    n=n*2**i
print(time()-top)

top=time()
n=2**180
for i in range(1800):
    n=n*2**i
print(time()-top)
```

4.2.Écriture des entiers en Python

Sous python, la mise en mémoire d'un entier n ne se limite pas à sa valeur ; diverses autres informations seront également concernées. On parle alors d'entiers multi-précision. Sur une machine fonctionnant en 64 bits, toutes ces informations sont stockées sous forme de suites (blocs) d'octets. L'ordre d'enregistrement des octets dépend de la machine ; on parle d'ordre little endian ou big endian.

Les entiers multi-précision sont codés de la manière suivante :
- Une première série de 32 bits permet d'indiquer le nombre de mots (blocs d'octets) contenus dans le code de l'entier
- Une série de mots correspondant au code l'entier

À l'aide du module `sys`, nous pouvons obtenir certaines :
```
>>> Import sys
```
- `maxsize` : indique la valeur maximale des entiers non-signés gérés par la machine donnée (c'est-à-dire $2^{32}-1$ en 32 bits et $2^{64}-1$ en 64 bits)

```
>>> sys.maxsize
9223372036854775807
```

- `sizeof_digit` : indique la taille en octets d'un mot en Python (dans notre cas, chaque mot contient 4 octets)
- `bits-per-digit` : indique la base utilisée pour le stockage des entiers longs c'est-à-dire la base associée à chaque mot (ici 2^{30}).

```
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

Remarque : afin de simplifier l'écriture, on indiquera par la suite le contenu de chaque octet en hexadécimal et non en binaire. Chaque octet contiendra ainsi deux valeurs et non huit bits.
Ex : $(53)_{16} = 5\times16+3 = 83$ pour $(0101\,0011)_2 = 1\times2^6+1\times2^4+1\times2+1 = 64+16+2+1 = 83$

Considérons un premier exemple d'entier multi-précision n'utilisant qu'un seul mot de quatre octets. Supposons que l'on veuille mettre en mémoire le nombre 537 sous forme d'entier multi-précision. On aura alors :

01	00	00	00	00	00	00	00	19	02	00	00

Le premier bloc de huit octets (huit cases) représente le nombre 1 c'est-à-dire le nombre de bloc utilisé pour écrire le nombre 537. Le bloc de quatre octets suivant représente le nombre 537 en décimal. Dans ce bloc, la valeur est codé en base $2^8 = 256$ (un octet). En effet, on a :

$$1\times 16+9=25\,;\ 0\times 16+2=2 \text{ soit } 0\times 256^3+0\times 256^2+2\times 256+25=537$$

Remarque : ici, on utilise l'ordre big endian, c'est-à-dire qu'on lit les octets de gauche à droite, ceux de gauche correspondant au puissance de 256 les plus faibles. Avec un ordre Little endian, on aurait :

$$0\times 256^0+0\times 256^1+2\times 256^2+25\times 256^3=131072+419430400=419561472$$

Considérons maintenant le nombre suivant codés à l'aide de trois mots.

03	00	00	00	00	00	00	00	19	02	00	00	05	00	00	00	07	00	00	00

On a alors : $n=537+5\times 2^{30}+7\times 2^{60}$

Remarque : on sait que chaque bloc correspond à $\left(2^{30}\right)^n$ suite à la commande (`bits-per-digit`)
Les huit premiers octets donnent le nombre de blocs (3). Ensuite, on a quatre octets pour le nombre 537, quatre octets pour le nombre 5 et quatre pour le nombre 7.

D'une manière générale, nous retiendrons que pour une entier positif, on trouve en mémoire ,codé sur huit octets, le nombre de blocs de 4 octets utilisés. L'entier est ensuite codé suivant sa taille (le nombre de blocs de quatre octets) sur 4 octets, 8 octets, 12 octets... Chaque bloc de 4 octets est un nombre compris entre 0 et $2^{30}-1$ bornes comprises. Le dernier bloc ne peut être 0. La valeur du nombre sera obtenue par :

$$[bloc\,1]+[bloc\,2]\times 2^{30}+[bloc\,3]\times 2^{60}...$$

4.3.Calculs

Les calculs sur des entiers doivent donner des résultats exacts. Pour des entiers très grands, stockés en Python sur plusieurs blocs, il faut gérer les opérations qui sont données à exécuter au processeur.

Considérons l'exemple du produit de deux entiers positifs $p=a+b\times 2^{30}$ et $q=c+d\times 2^{30}$. En utilisant la distributivité, il vient que le résultat a pour expression :
$r=ac+(ad+bc)\times 2^{30}+bd\times 2^{60}$
Il y a donc quatre produits à calculer avec des nombres compris entre 0 et 2^{30}. La valeur de chaque produit est donc compris entre 0 et 2^{60}. Le résultat est quant à lui compris entre 0 et 2^{120}.

L'algorithme de Karatsuba présenté ci-après, utilise une stratégie plus performante. On peut commencer par simplifier la présentation en passant en base dix. Pour simplifier, on prend :

$$p=a\times 10^m+b \text{ et } q=c\times 10^m+d$$

Dans ce cas, on a alors :

$$p\times q=ac\times 10^{2m}+(ad+bc)\times 10^m+bd$$

Toutefois, le nombre de produits à effectuer reste le même qu'avant. On peut alors remarquer que :

$$\begin{aligned} p\times q &= ac\times 10^{2m} + \big(ac+bd-(a-b)(c-d)\big)\times 10^{m} + bd \\ &= \big(10^{2m}+10^{m}\big)\times ac + (a-b)(c-d)\times 10^{m} + \big(10^{m}+1\big)bd \end{aligned}$$

Nous n'avons plus que trois produits à calculer.

Cette formule peut être utilisée dans le cadre d'un processus récursif. La valeur par défaut du paramètre m est 9. Cela correspond à une taille de mot de 10^9 soit un peu moins que 2^{30}.

```
def karatsuba(p,q):
    m=9
    if p<q:
        p,q=q,p
    if q==0:
        return 0
    if q<=10**m:
        return p*q
    a,b=p//10**m,p%10**m
    c,d=q//10**m,q%10**m
    ac=karatsuba(a,c)
    bd=karatsuba(b,d)
    b_a_c_d=karatsuba(b-a,c-d)
    r=(10**(2*m)+10**m)*ac+10**m*b_a_c_d+(10**m+1)*bd
    return r
```