éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

REPRÉSENTATION DES ENTIERS NATURELS

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. Des nombres

En mathématiques, les *nombres*[1] comme les *entiers naturels*, les *entiers relatifs*, les *rationnels* ou encore les *réels* sont de natures différentes. Bien que beaucoup d'entre eux puissent être usuellement *représentés* à l'aide de *signes* particuliers appelés *chiffres*, ils n'en diffèrent pas moins par leur nature. Le nombre entier 123 n'est pas le réel $123{,}0$ pour lequel les décimales nulles ne sont pas écrites. Comment écrire le nombre $1/3$ sous forme décimale ? Ou encore, comment écrire le nombre π ? Toutes ces différences peuvent être oubliées par l'utilisateur des nombres tant que celui-ci applique des règles de calculs qui facilitent, voire rendent complètement transparents, ces calculs entre nombres de natures différentes.

En informatique, cette distinction est nécessaire puisque des nombres de natures différentes sont définis par des objets de *types* différents. Un type *entier* permet de coder les nombres entiers. Un type *flottant* permet de coder certains nombres réels. Les opérations arithmétiques sont, de fait, définies pour chaque type de nombres. En pratique, des langages comme *Python* autorisent l'utilisation de même signes, comme `+`, `-`, `*`, `/` pour désigner les opérateurs binaires s'appliquant à tous les types de nombres. Mais ce caractère permissif ne doit pas masquer certaines difficultés liées, d'une part au *codage* des nombres dans un ordinateur, d'autre part aux conséquences arithmétiques liées à ces codages. Par exemple, comment expliquer que `0.1 + 0.2 + 0.3` ne renvoie par exactement `0.6` en *Python* ? La réponse à cette question sera donnée dans le document relatif à la représentation des nombres réels. Mais comment expliquer que $255 + 1$ renvoie 0 plutôt que 256 sur certaines machines ? Ce présent document consacré à la représentation des entiers naturels répond à cette dernière question.

2. Entiers naturels

En *représentation décimale*, les chiffres sont les signes `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` et `9`. D'autres *représentations* des nombres sont possibles. La *représentation binaire* adopte un sous-ensemble des chiffres précédents dans lequel les seuls signes utilisés pour représenter des entiers sont `0` et `1`. La *représentation hexadécimale* adopte un sur-ensemble des chiffres précédents en ajout les signes A, B, C, D, E et F. Enfin, d'autres représentations sont encore possibles, avec avec d'autres signes comme par exemple les chiffres romains, les sinogrammes, etc.

1. Dans cet exposé, la question de l'origine des nombres n'est pas abordée. Elle pourrait toutefois constituer un sujet d'ouverture pour amener les élèves à dépasser le point de vue scolaire où les nombres sont introduits et acceptés de manière ad-hoc dès les petites classes. Toutefois, il conviendrait de se limiter à une prise de conscience du sujet.

Représentation	Signes utilisés	Un même entier
Binaire	0, 1	1111011
Ternaire	0, 1, 2	11120
Octale	0, 1, 2, 3, 4, 5, 6, 7	173
Décimale	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	123
Hexadécimale	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	7B

2.1. Représentation décimale

Tout nombre entier naturel décimal peut s'écrire comme combinaison linéaire de puissance de 10. Par exemple :

$$123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

De manière générale, pour tout entier naturel non nul [2] n comportant p chiffres, il existe un ensemble de chiffres $\{d_0, d_1, \ldots, d_{p-1}\}$, où chaque $d_i \in \{0, 1, \ldots, 9\}$ pour $i \in \{0, \ldots, p-1\}$ et $d_{p-1} \neq 0$, tels que :

$$n = \sum_{i=0}^{p-1} d_i \times 10^i$$

Cette écriture constitue la *décomposition décimale* de n. La suite $(d_{p-1}, d_{p-2}, \ldots, d_0)$ est sa *représentation décimale*. Dans le reste de l'exposé, cette représentation est notée plus traditionnellement sous la forme $d_{p-1}d_{p-2}\ldots d_0$. [3]

Pour obtenir les chiffres d_i à partir de l'entier naturel n, deux opérations sont utiles.

- ▷ la division entière, `//` en *Python*, qui à deux entiers a et b associe leur quotient q ;
- ▷ le reste de la division entière, `%` en *Python*, qui à deux entiers a et b associe le reste r de la division euclidienne : $r = a - b \times q$.

À l'aide de ces opérations, il est possible d'obtenir les chiffres d'un entier naturel à l'aide de l'algorithme suivant.

```
Choisir un entier naturel n.
Initialiser r à n.
Tant que r n'est pas nul, répéter les instructions suivantes.
  Calculer r % 10 et stocker le résultat.
  Remplacer r par r // 10
Renvoyer les chiffres stockés
```

Les chiffres sont les entiers stockés. En *Python*, cet algorithme peut être traduit par le code suivant.

```
# entier à décomposer
n = 123
# initialisation
r = n
chiffres = []
# détermination des chiffres
while r != 0:
  chiffres.append(r % 10)
    r = r // 10
chiffres.reverse()
```

Dans ce code, les résultats sont stockés dans un tableau initialement vide. À chaque nouveau calcul du reste, le résultat est ajouté en queue de tableau à l'aide de la méthode `append`. Pour avoir les chiffres dans le bon ordre, il faut alors retourner le tableau à l'aide de `reverse`.
Partant d'une liste de chiffres représentée sous forme d'un tableau, un entier est peut être re-construit. Noter l'absence d'utilisation de l'opération d'exponentiation dans le code suivant.

2. Le nombre 0 a une écriture qui diffère un peu, puisqu'elle comporte un unique `0`.
3. La représentation décimale de 0 devrait être une suite de longueur 0. À la place, on a comme convention d'écrire `0`.

```
# tableau de chiffres
chiffres = [3, 1, 4, 1, 5, 9, 2]
# initialisation
n = 0
# calcul de l'entier
for c in chiffres:
  n = 10 * n + c
```

2.2. Représentation binaire

Tout nombre entier naturel peut aussi être représenté en base 2 à l'aide des seuls signes 0 et 1. Par exemple, $(101)_2$, $(111001)_2$, $(1111111)_2$. Pour distinguer ces représentations binaires de représentations décimales, un indice 2 est ajouté. Il convient également de ne pas lire ces nombres comme on lirait des nombres décimaux. Ainsi, $(101)_2$ n'est pas cent un mais un zéro un.

De manière générale, la *représentation binaire* de tout entier naturel non nul n à m chiffres 0 ou 1 peut se mettre sous la forme $b_{m-1}b_{m-2}\dots b_0$ où $b_i \in \{0,1\}$ pour $i \in \{0,\dots,m-1\}$ et $b_{m-1} \neq 0$ (en binaire, cette dernière condition impose nécessairement $b_{m-1} = 1$). Ainsi, on peut écrire :

$$n = (1b_{m-2}\dots b_0)_2$$

Convertir un entier du système binaire au système décimal et réciproquement peut aisément se faire.

$$(1111011)_2 \quad \leftrightarrow \quad 1\times 2^6 + 1\times 2^5 + 1\times 2^4 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 \quad = \quad 123$$

Un paragraphe présente plus loin un code réalisant ces opérations.

2.3. Représentation hexadécimale

Comme les représentations décimale et binaire, la *représentation hexadécimale* est un système de numération positionnel en base 16. Tout entier naturel non nul n à p chiffres dans ce système peut s'écrire sous la forme :

$$n = (h_{p-1}\dots h_0)_{16}$$

avec, pour tout entier $i \in \{0,\dots,p-1\}$, $h_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ et $h_{p-1} \neq 0$.
Le tableau suivant présente les conversions des premiers entiers naturels entre les trois représentations précédentes.

Décimal	Binaire	Hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Décimal	Binaire	Hexadécimal
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Là encore, convertir un entier du système hexadécimal au système décimal et réciproquement peut aisément se faire.

$$(7B)_{16} \quad \leftrightarrow \quad 7\times 16^1 + 11\times 16^0 \quad = \quad 123$$

La conversion hexadécimal ↔ binaire peut également se faire rapidement : chaque chiffre hexadécimal équivaut à exactement quatre chiffres binaires.

- ▷ Reprenant l'exemple du nombre 7B, le tableau précédent indique que 7 peut être converti en 0111 et B en 1011. Ce qui donne la représentation binaire 01111011 ou 1111011 si on omet le premier zéro.
- ▷ À la représentation binaire 1101110 peut être associée deux blocs de quatre chiffres : 0110 et 1110. Le premier bloc se convertit en 6 et le second en E de sorte que la représentation hexadécimale de l'entier est 6E.

Particulièrement pratique pour représenter de grands entiers sous forme compacte, la représentation hexadécimale est fréquemment utilisée en électronique numérique et dans le monde des réseaux pour décrire des adresses.

3. Conversions

3.1. Fonctions Python

En *Python*, quelques fonctions permettent le passage d'une représentation à une autre. Tout d'abord, pour passer une représentation décimale à une représentation binaire ou hexadécimale, *Python* dispose des deux fonctions :

- **bin** qui reçoit un entier naturel et renvoie une chaîne de caractères associée à la représentation binaire de cet entier ; cette chaîne débute systématiquement par le préfixe `0b` ;

```
>>> bin(123)
'0b1111011'
```

- **hex** qui procède de la même façon et renvoie une chaîne de caractères associée à la représentaton hexadécimale de l'entier, avec le préfixe `0x`.

```
>>> hex(123)
'0x7b'
```

Les opérations inverses peuvent se faire à l'aide de la fonction **int** qui reçoit deux arguments : une chaîne de caractères représentant un entier dans une base donnée et un entier représentant cette base.

```
>>> int('0b1111011', 2)
123
```

```
>>> int('0x7b', 16)
123
```

3.2. Programmation

L'exercice consistant à programmer ces mêmes fonctions est très formateur. Nous proposons donc de construire plusieurs fonctions qui réalisent ces opérations de conversions. Partant d'un type entier, en représentation décimale, ces fonctions doivent renvoyer une chaîne de caractères associée à la représentation de l'entier dans n'importe quelle base comprise entre 2 et 16.

Avant d'envisager le cas général, commençons par écrire deux fonctions qui réalisent la conversion en binaire et en hexadécimal. Les algorithmes mis en œuvre pour réaliser ces conversions sont très proches de celui présenté en début de document, permettant la détermination des chiffres composant un entier. Dans notre cas, il suffit de remplacer la base 10 par 2 ou 16. Toutefois, dans ce dernier cas, la construction de la chaîne de caractères requiert une opération supplémentaire consistant à associer à chaque entier compris entre 0 et 15 un chiffre hexadécimal. Nous proposons de définir une première fonction qui renvoie un *dictionnaire* réalisant cette conversion décimal → hexadécimal.

```
def dic_hex_to_dec():
    s = '0123456789ABCDEF'
    dic = {}
    for i in range(16):
        dic[s[i]] = i
    return dic
```

Les code *Python* des deux fonctions de conversion d'un entier décimal sont alors très proches. La fonction **int** réalise la conversion d'une chaîne de caractères en un entier.

```
def bin_to_dec(chaine_bin):
    n = 0
    for b in chaine_bin:
        n = 2 * n + int(b)
    return n
```

```
def hex_to_dec(chaine_hex):
    dic = dic_hex_to_dec()
    n = 0
    for h in chaine_hex:
        n = 16 * n + int(dic[h])
    return n
```

Partant d'une chaîne de caractères représentant un entier binaire ou hexadécimal, les fonctions suivantes déterminent l'entier associé en représentation décimale.

```
def dec_to_bin(n):
    chaine_bin = ''
    nb = n
    while nb > 0:
        r = nb % 2
        nb = nb // 2
        chaine_bin = str(r) + chaine_bin
    return chaine_bin
```

```
def dec_to_hex(n):
    s = '0123456789ABCDEF'
    chaine_hex = ''
    nb = n
    while nb > 0:
        r = nb % 16
        nb = nb // 16
        chaine_hex = s[r] + chaine_hex
    return chaine_hex
```

Enfin, ces fonctions peuvent être utilisées pour réaliser les conversions binaire ↔ hexadécimal.

```
def hex_to_bin(lst_hex):
    return dec_to_bin(hex_to_dec(lst_hex))

def bin_to_hex(lst_bin):
    return dec_to_hex(bin_to_dec(lst_bin))
```

Les fonctions suivantes synthétisent les propositions précédentes.

```
def dic_to_dec(base):
    '''
    Renvoie un dictionnaire associant un chiffre en base base
        à sa représentation décimale.
    Entrée : base <= 16
    Sortie : dictionnaire
    '''
    s = "0123456789ABCDEF"
    dic = {}
    for i in range(base):
        dic[s[i]] = i
    return dic
```

```
def to_dec(entier,base):
```

```
    '''
    Conversion entier décimal -> entier en base base
    Entrée : chaîne de caractères d'un entier en base base, base
    Sortie : entier décimal
    '''
    dic = dic_to_dec(base)
    n = 0
    for c in entier:
        n = base * n + dic[c]
    return n
```

```
def dec_to(entier,base):
    '''
    Conversion entier en base base -> entier décimal
    Entrée : entier décimal, base (base <= 16)
    Sortie : chaîne de cararactères représentant l'entier en base base
    '''
    s = "0123456789ABCDEF"
    nb = entier
    chaine = ""
    while nb > 0:
        r = nb % base
        nb = nb // base
        chaine = s[r] + chaine
    return chaine
```

4. Codage binaire

4.1. Des bits

D'un point de vue matériel, un ordinateur est un ensemble de composants électroniques parcourus par des courants électriques. Par convention, le passage d'un courant dans un composant est codé par le chiffre 1, l'absence de courant étant codé par un 0. Ainsi, toute information stockée dans un ordinateur peut être codée par une suite finie de 0 et de 1 appelée suite de *bits* (*binary digits*). Une convention fixe la *taille* de ces suites finies de bits. Par exemple, un codage sur 4 bits signifie qu'une information est représentée en machine à l'aide de quatre 0 ou 1. L'ordre de ces bits importe de sorte que 0111 ne code pas la même information que 1110. En outre, les zéros présents en début de codage sont indispensables. Ainsi, sur 4 bits, 2^4 entiers naturels peuvent être codés ; par exemple, les entiers de 0 à 15.

$0 \leftrightarrow$ `0000` $1 \leftrightarrow$ `0001` $2 \leftrightarrow$ `0010` $3 \leftrightarrow$ `0011` $4 \leftrightarrow$ `0100` $5 \leftrightarrow$ `0101` $6 \leftrightarrow$ `0110` $7 \leftrightarrow$ `0111`
$8 \leftrightarrow$ `1000` $9 \leftrightarrow$ `1001` $10 \leftrightarrow$ `1010` $11 \leftrightarrow$ `1011` $12 \leftrightarrow$ `1100` $13 \leftrightarrow$ `1101` $14 \leftrightarrow$ `1110` $15 \leftrightarrow$ `1111`

Si N est un entier naturel non nul, 2^N entiers peuvent être codés en binaire sur N bits : $0, 1, 2, \ldots, 2^N - 1$.
La fonction suivante renvoie, sous la forme d'une chaîne de caractères, le *codage binaire* d'un entier naturel sur N bits.

```
def codage_bin(n, N):
    '''
    Renvoie le codage binaire de l'entier naturel n sur N bits
    Entrée : entiers n et N
    Sortie : chaîne de caractères du codage binaire de n sur N bits
    '''
    nb = n
    chaine = ""
```

```
    for _ in range(N):
        r = nb % 2
        chaine = str(r) + chaine
        nb = nb // 2
    return chaine
```

4.2. Arithmétique élémentaire

La fonction ne renvoie de résultat pertinent que si $n < 2^N$.

```
>>> codage_bin(7, 4)
'0111'
```

```
>>> codage_bin(16, 4)
'0000'
```

Cela signifie que les entiers plus grands que 2^N ne peuvent pas être codés sur N bits. La capacité de codage sur N bits est dépassée. Soit on ajoute des bits pour coder de plus grands entiers, soit on se contente de coder les entiers sur N bits.
Ce résultat prend toute son importance dès qu'on veut réaliser des opérations arithmétiques élémentaires comme l'addition ou la multiplication. La soustraction est abordée dans le document traitant de la *représentation des entiers relatifs*. La division entière n'est pas abordée.
Illustrons notre propos avec $N = 4$ bits et deux entiers $n_1 = 3$ et $n_2 = 4$. L'addition de ces entiers est l'entier $n_3 = 7$. En binaire, les additions sont effectuées suivant les règles suivantes.

- `0 + 0` donne `0` avec une retenue égale à `0`;
- `0 + 1` donne `1` avec une retenue égale à `0`;
- `1 + 0` donne `1` avec une retenue égale à `0`;
- `1 + 1` donne `0` avec une retenue égale à `1`;

Le codage de n_1 et n_2 sur 4 bits donne respectivement `0011` et `0100`. L'addition de ces codages peut être posée et s'écrit :

retenue	0	0	0	0	
		0	0	1	1
	+	0	1	0	0
	0	0	1	1	1

Le résultat obtenu, à savoir `0111` sur 4 bits, correspond au codage binaire de 7.
Choisissons à présent $n_1 = 11$ et $n_2 = 13$ dont les codages binaires sur 4 bits sont respectivement `1011` et `1101`. L'addition de ces codages donne alors :

retenue	1	1	1	1	
		1	0	1	1
	+	1	1	0	1
	1	1	0	0	0

Le résultat obtenu est `11000`. En codant sur 4 bits, seuls les 4 bits de *plus faibles poids* sont conservés, à savoir `1000`. Ce codage est celui de l'entier 8 et non celui de la somme $11 + 23 = 24$. On peut remarquer $8 = 24 - 16$; le résultat est la somme modulo 16. L'addition des codages réalise la somme modulo 2 puissance le nombre de bits de codage. Ceci est à rapprocher de la phrase de l'introduction selon laquelle $255 + 1$ renvoie 0 plutôt que 256 sur certaines machines, en l'occurence les machines codant les entiers naturels sur 8 bits ou moins !
Pour conclure cet exposé, un code réalisant l'addition bit à bit est proposé. Des variations et des prolongements sur ce sujet sont possibles : multiplication binaire, notions d'additionneurs pouvant être abordée à travers la partie consacrée à l'architecture des machines, notamment lors de l'introduction aux circuits logiques.

```
def add(code1, code2, N):
    code3 = ""
    r = 0
    i = N
    while i > 0:
        i = i - 1
        b1 = int(code1[i])
        b2 = int(code2[i])
        s = (b1 + b2 + r) % 2
        r = (b1 + b2 + r) // 2
        code3 = code3 + str(s)
    return code3
```