Leçon d'informatique : Représentation des entiers sur des mots de tailles fixées

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

Notions	Commentaires
Représentation des entiers positifs sur des mots de taille fixe.	La conversion d'une base à une autre n'est pas un objectif de formation.
Représentation des entiers signés sur des mots de taille fixe.	Complément à deux.
Entiers multi-précision de Python.	On les distingue des entiers de taille fixe sans détailler leur implémentation. On signale la difficulté à évaluer la complexité des opérations arithmétiques sur ces entiers.

À retenir : En informatique, un entier non signé correspond à un entier naturel appartenant à $\mathbb{N}$.

I Différentes représentations

I.A *Bit*, octet, *bytes*, mots

On rappelle que l'unité élémentaire sous laquelle est codée l'information est le *bit* (pour *binary digit*) que l'on symbolise par un état 0 ou un état 1.
On peut regrouper les bits par paquet de 8 et former un octet (ou *bytes*). Par exemple, $a = 1001\ 1111$ représente un octet.

À retenir : Si un bit peut prendre $2 = 2^1$ valeurs, un octet pourra prendre $2^8 = 256$ valeurs.

Enfin, on peut prendre un ensemble d'octet et former un **mot**. Par exemple, une adresse IPv4 locale peut être 192.168.1.77 et un masque de sous-réseau 255.255.255.0 sont dans la pratique, deux mots composés de quatre octets en représentation décimale. On peut aussi donner cette adresse IP

- en représentation binaire : 11000000 10101000 00000001 01001101 ;
- en représentation hexadécimale : $c0 : a8 : 01 : 4d$.

I.B Représentation décimale

La représentation décimale est la représentation habituelle en mathématiques. On rappelle que :

$$n = 2367 = 2 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 7 \times 10^0$$

On pourra alors associer à l'entier n la représentation décimale $(2,3,6,7)$ ou $\underline{2367}_{10}$.
Pour obtenir les différents chiffres, on peut :

- diviser successivement par 10 et récupérer les restes :

2367	10		236	10		23	10		2	10	
236		puis	23		puis	2		puis	0		
7			**6**			**3**			**2**		

On récupère donc la liste 7,6,3 et 2 que l'on peut ensuite inverser.

- Si on connaît le nombre p de chiffres de la représentation, on divise successivement par 10^{p-1} puis 10^{p-2} …pour finir par 10^0 en récupérant les quotients successifs.

2367	$\underline{10^3}$		367	$\underline{10^2}$		67	$\underline{10^1}$		7	$\underline{10^0}$	
	2	puis		**3**	puis		**6**	puis		**1**	
367			67			7			0		

Finalement, la représentation décimale d'un entier naturel n à p chiffres peut s'écrire :

$$n = \underline{a_{p-1}a_{p-2}\dots a_1a_0}_{10} = \sum_{k=0}^{p-1} a_k \times 10^k \text{ avec } \forall k, a_k \in [\![0,9]\!]$$

Remarque : La représentation décimale d'un entier naturel est unique.

Remarque : Pour un entier naturel n à p chiffres, on a bien évidemment :

$$0 \leqslant n \leqslant 10^p - 1$$

En effet, pour $n = \underline{2367}_{10}$, $p = 4$ et $0 \leqslant n \leqslant 10^4 - 1 = 9999$.

I.C Représentation binaire

Les chiffres successifs de la représentation binaire représentent les puissances de 2. Ainsi :

$$n = \underline{13}_{10} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = \underline{1101}_2$$

L'écriture binaire de 13 est ici écrite sur $p = 4$ bits. On constate d'ailleurs que $0 \leqslant 13 \leqslant 2^4 - 1 = 15$. Ainsi, la représentation binaire à p bits peut s'écrire :

$$n = \underline{b_{p-1}b_{p-2}\dots b_1b_0}_2 = \sum_{k=0}^{p-1} b_k \times 2^k \text{ avec } \forall k, b_k \in [\![0,1]\!]$$

Pour obtenir les différents chiffres, on peut :

- Si on connaît le nombre p de chiffres de la représentation, on divise successivement par 10^{p-1} puis 10^{p-2} …pour finir par 10^0 en récupérant les quotients successifs.

13	$\underline{2^3}$		5	$\underline{2^2}$		1	$\underline{2^1}$		1	$\underline{2^0}$	
	1	puis		**1**	puis		**0**	puis		**1**	
5			1			**1**			0		

- diviser successivement par 2 et récupérer les restes successifs :

13	2		6	2		3	2		1	2	
	6	puis		3	puis		1	puis		0	
1			**0**			**1**			**1**		

On récupère donc la liste 1,0,1 et 1 que l'on peut ensuite inverser.

Remarque : La représentation binaire d'un entier naturel est unique.

Remarque : Pour un entier naturel n dont la représentation binaires a p chiffres, on a bien évidemment :

$$0 \leqslant n \leqslant 2^p - 1$$

En effet, pour $n = \underline{13}_{10} = \underline{1101}_2$, $p = 4$ et $0 \leqslant n \leqslant 2^4 - 1 = 15$.

Exercice 1 : En utilisant l'une ou l'autre des méthodes, écrire une fonction `binaire(n, p)` qui prend en argument deux entiers positifs `n` et `p` et qui renvoie une chaîne de caractères correspondant à la représentation binaire de `n` de longueur `p`. Par exemple, `binaire(13, 4)` renvoie `"1101"` et `binaire(0, 3)` renvoie `"000"`. La réponse est en figures 1 et 2.

Exercice 2 : Écrire une fonction `binaire_rec(n)` qui prend en argument un entier positif `n` et qui renvoie une chaîne de caractères correspondant à la représentation binaire de `n`. La fonction sera écrite sous forme récursive. La réponse est en figure 3.

Méthode `bin` : La méthode `bin` est déjà implémentée. `bin(13)` renvoie `"0b1101"`. La lettre `b` indique que la représentation binaire.
On peut récupérer un résultat similiare au fonctions précédentes par `bin(13)[2:]` qui renverra `"1101"`.

I.D Représentation hexadécimale (hors-programme)

Les chiffres successifs de la représentation hexadécimale représentent les puissances de 16. Ainsi :

$$n = \underline{2367}_{10} = 9 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = \underline{93f}_{16}$$

L'écriture hexadécimale de 2367 est ici écrite sur $p = 3$ *digits*. On constate d'ailleurs que $0 \leqslant 2367 \leqslant 16^3 - 1 = 4095$. Ainsi, la représentation hexadécimale à p bits peut s'écrire :

$$n = \underline{h_{p-1}h_{p-2}\dots h_1h_0}_2 = \sum_{k=0}^{p-1} h_k \times 16^k \text{ avec } \forall k, h_k \in [\![0,15]\!]$$

Remarque : On notera ici que l'écriture hexadécimale fait, par commodité, intervenir des lettres.

valeur du *digit*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
symbole	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`	`8`	`9`	`a`	`b`	`c`	`d`	`e`	`f`

Exercice 3 : Écrire une fonction `hexadecimale(n, p)` qui convertit un entier `n` en sa réprésention hexadécimale de longueur `p`. Le résultat sera une chaîne de caractère. La réponse est donnée en figure 4.

II Représentation des entiers naturels sur des mots de 32 ou 64 bits

Remarque : On rappelle que 32 bits correspondent à 4 octets et 64 bits correspondent à 8 octets.

II.A Mots de taille constante

Idée : Par souci de simplicité, on peut coder tous les entiers sur des mots de taille constante (par exemple, des mots de taille 8 pour les machines 64 bits). En effet :

- si tous les mots sont de même taille, on peut optimiser les calculs ;
- l'adresse du premier octet permet aussi de connaître celle des 7 suivants.

Remarque anecdoctique : Convention de bit fort ou bit faible - Lorsqu'on écrit $\underline{8}_{10} = \underline{1000}_2$, le premier bit de la représentation binaire correspond à 2^3 qui est le bit de poids le plus fort. On parle de statégie **big-endian** (la plus utilisée). On aurait pu utiliser la stratégie **little-endian** : $\underline{8}_{10} = \underline{0001}_2$ où le premier bit est celui de poids plus faible.

II.B Addition de deux entiers non signés

Si on dispose de deux entiers non signés écrits chacun sur un octet, l'addition se fait chiffre par chiffre en tenant compte d'une éventuelle retenue. Par exemple, si on veut faire la somme de $\underline{157}_{10} = \underline{10011101}_2$ et de $\underline{9}_{10} = \underline{00001001}_2$, cela donne :

$$\begin{array}{cccccccccc} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline = & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Exercice 4 : Faire la somme de $\underline{123}_{10} = \underline{1111011}_2$ et de $\underline{29}_{10} = \underline{00011101}_2$. La réponse est donnée en figure **??**.

II.C Produit de deux entiers non signés

En base 10, multiplier par 10 revient à décaler vers la gauche. C'est la même chose en base 2 :

- pour multiplier par 2, on décale une fois à gauche : $2 \times 25 = 2 \times \underline{00011001}_2 = \underline{00110010}_2 = 50$;
- pour multiplier par 4, on décale deux fois à gauche : $4 \times 25 = 2 \times \underline{00011001}_2 = \underline{01100100}_2 = 100$;
- pour multiplier par 6, on additionne le produit par 2 et le produit par 4.

Exercice 5 : Écrire la multiplication en représentation binaire de 12×7. La réponse est donnée en figure **??**.

II.D *Overflow*

Prenons deux entiers codés sur 8 bits, si on fait la somme de 217 et 50, cela donne :

	1	1	0	1	1	0	0	1
+	0	0	1	1	0	0	1	0
=	0	0	0	0	1	0	1	1

Ce calcul donne un résultat faux car on n'a pas pu mettre la dernière retenue. En effet, $217 + 50 = 267 > 255 = 2^8 - 1$. On observe un phénomène de **dépassement** ou d'**overflow**.

Remarque : Python résout ce problème par un codage des entiers en **multi-précision**.

III Représentation des entiers signés sur des mots de taille fixée

À retenir : En informatique, un entier signé correspond à un entier relatif appartenant à $\mathbb{Z}$.

III.A Une première représentation

Pour représenter à la fois, les entiers positifs et négatifs, on peut d'abord imaginer attribuer le premier bit à la représentation du signes (0 si positif et 1 si négatif). Si on code un entier n sur un octet, il resterait alors 7 bits pour coder la valeur absolue. Cela donne :

- des entiers positifs entre 0 et $2^7 - 1 = 127$ (respectivement $\underline{00000000}_2$ et $\underline{01111111}_2$) ;
- des entiers négatifs entre -127 et 0 (respectivement $\underline{11111111}_2$ et $\underline{10000000}_2$) ;

On constate alors :

- il y a deux zéros !
- si on fait la somme de 43 et -43, cela donne :

	0	0	1	0	1	0	1	1
+	1	0	1	0	1	0	1	1
=	1	1	0	1	0	1	1	0

qui correspond à $\underline{11010110}_2 = -86$ au lieu de 0.

III.B Représentation en complément à deux

Pour palier à ce problème, on utilise la représentation en complément à deux.

À retenir : Pour des entiers n signés codés sur des mots de taille 1 octet (8 bits), on propose :

- de coder les entiers positifs entre 0 et $2^7 - 1 = 127$ avec un bit de poids fort nul. Ainsi, on aura $\underline{127}_{10} = \underline{01111111}_2$.
- un entier négatif $-n$ entre -2^7 et -1 sera codés par la représentation binaire de $2^8 - 1$. Ainsi, on aura $-\underline{128}_{10}$ qui sera représenté par $\underline{256}_{10} - \underline{128}_{10} = \underline{128}_{10} = \underline{10000000}_2$ et $-\underline{1}_{10}$ qui sera représenté par $\underline{256}_{10} - \underline{1}_{10} = \underline{255}_{10} = \underline{11111111}_2$

Remarque : Il n'y qu'un seul zéro. L'addition se fait sans difficulté :

Somme de $110 - 43 = \underline{1101110}_2 + \underline{11010101}_2$.

	0	1	1	0	1	1	1	0
+	1	1	0	1	0	1	0	1
=	0	1	0	0	0	0	1	1

Ce qui correspond bien à $110 - 43 = 67 = \underline{01000011}_2$.

Somme de $43 - 43 = \underline{00101011}_2 + \underline{11010101}_2$

	0	0	1	0	1	0	1	1
+	1	1	0	1	0	1	0	1
=	0	0	0	0	0	0	0	0

ce qui donne bien 0.

Remarque : Sur la plupart des machines, les entiers sont codés, par défaut, en 64 bits, cela veut dire que :

- les entiers positifs n compris entre 0 et $2^{63}-1$ [a] sont codés avec un 1 bit de poids fort 0 et il reste 63 bits pour la valeur de n.
- les entiers négatifs n sont codés en compléments à deux par $2^{64}-|n|$.

a. cela correspond à un entier de l'ordre de 10^{19}.

III.C Entiers multi-précision

Même si les entiers sont généralement codés en 64 bits, il est possible de dépasser cette taille. Les entiers sont virtuellement sans limite de taille. Dans la pratique, la limite est posée par le stockage.

Module `sys` (hors-programme) : Le module **sys** renvoie des informations sur l'implémentation de python. Comme on voit ci-dessous, l'entier le plus grand en 64 bits vaut bien $2^{63}-1 = 9223372036854775807$.
Les entiers sont stockés par des mots de taille 30 bits (ce qui est compris dans 4 octets). Les entiers peuvent contenir plusieurs mots : c'est la multi-précision. On peut donc obtenir des entiers de plus en plus grands en augmentant le nombre de mots.

```
import sys                          # module sys
print(sys.int_info)                 # sys.int_info(bits_per_digit=30, sizeof_digit=4)
print(sys.maxsize)                  #renvoie 9223372036854775807
```

Exercice 6 : Quel est le plus grand entier que l'on peut coder sur 1 Go ?

Conclusion

On résume : On retiendra que l'écriture binaire des entiers se fait sur des mots de taille généralement 64 bits mais cette taille est extensible.
Le caractère signé est obtenu par le complément à deux, cela donne un entier maximal $2^{63}-1$ et un entier minimal -2^{63}.

```
def binaire1(n : int, p:int) ->  str:
    """convertit un integer n en
    représentation binaire
    On suppose que la valeur de p convient
    """
    res = ""
    for k in range(p):
        bk = n%2
        res += str(bk)
        n=n//2
    return res[::-1] # on inverse
print(binaire1(13, 4)) # renvoie "1100"
```

FIGURE 1 – On divise par 2 à chaque itération.

```
def binaire2(n : int, p:int) ->  str:
    """convertit un integer n en
    représentation binaire
    On suppose que la valeur de p convient
    """
    res = ""
    k = p-1
    while k>=0:
        bk = n//(2**k)
        n=n%(2**k)
        res += str(bk)
        k -= 1
    return res # pas besoin d'inverser
print(binaire2(13, 4)) # renvoie "1100"
```

FIGURE 2 – On divise par des puissances de 2 décroissantes.

```python
def binaire_rec(n):
    if n < 0:
        return "n doit etre entier positif"
    if n <= 1:
        return str(n)
    else:
        return binaire_rec(n//2)+str(n%2)

print(binaire_rec(13)) ## renvoie 1101
print(binaire_rec(-10)) ## renvoie message
```

FIGURE 3 – Conversion de la représentation décimale en représentation hexadécimale.

```python
def hexadecimale(n : int, p:int) ->  str:
    """convertit un integer n en
    représentation hexadécimale
    On suppose que la valeur de p convient
    """
    L = "0123456789abcdef"
    res = ""
    k = p-1
    while k>=0:
        bk = n//(16**k)
        n=n%(16**k)
        res += str(L[bk])
        k -= 1
    return res # pas besoin d'inverser
print(hexadecimale(168, 2)) # renvoie "a8"
```

FIGURE 4 – Conversion de la représentation décimale en représentation hexadécimale.