CHAPITRE

Numération

ARITHMÉTIQUE 1

Sommaire

I Introduction : que signifie 1789 ?

On a besoin, dans de nombreux domaines, de pouvoir exprimer des quantités. Pour dire qu'on a un troupeau de 252 moutons, on pourrait montrer une allumette par tête, ou tracer un bâton par tête, de manière à ne pas avoir à trimballer tout son troupeau, mais cela ne serait guère pratique[1].

Il a donc fallu, au cours du temps, inventer des méthodes plus efficaces pour représenter les quantités. L'arrivée des symboles a permis de représenter les nombres par des écritures plus ou moins faciles à manipuler : systèmes babylonien, égyptien, basés sur la représentation de certaines quantités par des symboles, et par mise bout-à-bout de ces symboles pour les autres nombres, système romain, dans lequel la position d'un symbole peut modifier la signification du symbole suivant...

Notre système de numération moderne est fondé sur plusieurs idées intéressantes : un symbole pour chacun des nombres de 0 à 9, en raison de l'utilisation de la base décimale, et un principe de *numération de position* : un même chiffre a une signification différente selon sa position dans l'écriture du nombre.

De nombreuses civilisations ont utilisé (et utilisent encore) la base 10, sans doute pour des raisons physiologiques ! Le système de notation positionnelle provient de Chine, et a été amélioré et diffusé à partir de l'Inde, au VI$^{\text{ème}}$ siècle. Enfin, les chiffres que nous utilisons aujourd'hui ont été inventé par les indiens, et leur diffusion en Europe s'est faite par l'intermédiaire de la civilisation arabe aux alentours du IX$^{\text{ème}}$ siècle.

Mais que signifie donc une écriture telle que 1789 ? Et bien, à chaque position est associée un "poids", d'autant plus important que le chiffre est plus à gauche. Ce poids est une puissance de la base utilisée, ici la base 10. Ainsi :

$$\begin{aligned} 1789 &= 9 \times 10^0 + 8 \times 10^1 + 7 \times 10^2 + 1 \times 10^3 \\ &= 9 + 80 + 700 + 1000 \end{aligned}$$

Cette écriture est exceptionnellement économique en symboles, puisqu'on évite l'utilisation de symboles représentant 10, 100,... Elle permet surtout de réaliser efficacement les opérations dont nous avons le plus besoin dans la vie courante : *interprétation* d'une quantité, *comparaison* de deux quantités, *addition*, *soustraction*, *multiplication*[2]... Nous mettrons en œuvre ces méthodes en TP d'algorithmique lorsque nous programmerons les opérations usuelles sur des "grands" entiers.

II Les numérations de position

A Numération en base 10

Nous venons donc de voir le principe de la numération en base 10. Si un nombre entier s'écrit

$$a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0$$

où n est un entier supérieur ou égal à 1, les symboles a_i représentant des chiffres pris dans l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, alors la quantité qu'il représente est :

$$a_n \times 10^n + a_{n-1} \times 10^{n-1} + \dots a_2 \times 10^2 + a_1 \times 10^1 + a_0 \times 10^0 = \sum_{i=0}^{n} a_i \times 10^i$$

Le *poids* du chiffre a_k est 10^k, la puissance de 10 par laquelle il faut le multiplier pour connaître son influence dans le nombre. On remarquera que les chiffres dont le poids est le plus important (on parle des chiffres *les plus significatifs*) sont à gauche dans l'écriture du nombre. Ainsi, si l'on veut obtenir une bonne *approximation* d'un grand nombre, il suffit de ne conserver que les chiffres les plus à gauche, et de remplacer les autres par des 0 (pour conserver la signification des positions !).

B Numérations en base b

Si b est un entier supérieur ou égal à 2, on peut utiliser le principe ci-dessus pour représenter les nombres "en base b".

[1] Par contre, ce système de représentation "une allumette pour un mouton" est extrêmement pratique pour additionner les nombres de moutons de deux troupeaux : il suffit de réunir les paquets d'allumettes de chaque troupeau !

[2] On ne va pas mettre dans cette liste la division, qui n'est quand même pas une opération si simple que cela, même si notre système de numération permet de concevoir un algorithme relativement efficace. Mais essayez de diviser deux nombres écrits en chiffres romains, pour voir !

Il faut pour cela une collection de symboles pour représenter tous les *chiffres* de 0 jusqu'à $b-1$. C'est facile lorsque b est inférieur ou égal à 10, puisqu'il suffit de prendre les chiffres usuels en ne gardant que ceux strictement inférieurs à b. Par contre, pour des bases supérieures à 10, il faut "inventer" de nouveaux "chiffres".

Ainsi, en base 16, les chiffres sont : $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$, le A étant le chiffre "10", B le chiffre 11, etc.

Une fois cette collection de symboles choisie, un nombre dont l'écriture en base b est

$$a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0$$

où n est un entier supérieur ou égal à 1, les symboles a_i représentant des chiffres de la base b, alors la quantité qu'il représente est :

$$a_n \times b^n + a_{n-1} \times b^{n-1} + \dots a_2 \times b^2 + a_1 \times b^1 + a_0 \times b^0 = \sum_{i=0}^{n} a_i \times b^i \quad (*)$$

Lorsqu'il peut y avoir une confusion entre plusieurs bases, on ajoute en indice à droite du nombre la base utilisée :

- 754_8 est un nombre écrit en base 8,
- 11101110010_2 est un nombre écrit en base 2...
- qui ne doit pas être confondu avec 11101110010_{10}, qui est une écriture en base 10.

En l'absence d'indice et de contexte, la base employée est la base décimale.

Lorsqu'on écrit un source en langage informatique, on utilise un préfixe ou un suffixe pour préciser la base employée :

- en Pascal, l'absence de notation indique la base 10, un préfixe \$ indique un nombre hexadécimal, un % un nombre binaire, et un & un nombre octal (base 8) ; ainsi, \$1AE représente le nombre hexadécimal $1AE_{16}$;
- en C, les préfixes $0x$ et $0b$ désignent respectivement des nombres écrits en hexadécimal ou en binaire.

Notons que la formule $(*)$ fournit une méthode pour convertir un nombre de la base b vers la base 10.

C Deux bases particulièrement utiles en informatique

1 La base 2, ou système binaire

C'est la plus petite base envisageable. Elle n'utilise que deux symboles, 0 et 1[3]. Un chiffre binaire est appelé "bit" en informatique, ce qui est une contraction de "binary digit", autrement dit "chiffre binaire" en anglais. Le poids du bit en position k est 2^k.

Voici la représentation des premiers entiers en binaire :

En base 10	En binaire
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

En base 10	En binaire
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101

[3] ce qui tombe bien puisque l'électronique numérique sait représenter ces deux valeurs par deux plages de tensions différentes, de façon efficace. On pourrait imaginer un plus grand nombre de plages, mais le système deviendrait alors beaucoup plus sensible au *bruit*, sans gain réel d'efficacité.

Exemples :

- Le nombre 1110111_2 a pour valeur

$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 64 + 32 + 16 + 4 + 2 + 1 = 119$$

- Pour convertir le nombre 221 en base 2, on va chercher les puissances de 2 "entrant" dans ce nombre :
  - la plus grande puissance de 2 inférieure à 221 est $2^7 = 128$; le reste est $221 - 128 = 93$;
  - la plus grande puissance de 2 inférieure à 93 est $2^6 = 64$; le reste est $93 - 64 = 29$;
  - la plus grande puissance de 2 inférieure à 29 est $2^4 = 16$; le reste est $29 - 16 = 13$;
  - la plus grande puissance de 2 inférieure à 13 est $2^3 = 8$; le reste est $13 - 8 = 5$;
  - la plus grande puissance de 2 inférieure à 5 est $2^2 = 4$; le reste est $5 - 4 = 1 = 2^0$.

  Ainsi, $221_{10} = 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^0 = 11011101_2$.

Exercices :

a) Écrire les nombres 27, 31, 84 et 128 en binaire.

b) Donner la valeur des nombres dont l'écriture binaire est 110110_2, 111111_2 et 10101010_2.

2 La base 16, ou système hexadécimal

En base 16, on a vu que les "chiffres" sont $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. Nous verrons par la suite l'intérêt de cette base, qui est un substitut plus "humain" du binaire pour "communiquer" avec le microprocesseur d'un ordinateur.

Voici la représentation des premiers entiers en hexadécimal :

En base 10	En hexadécimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A

En base 10	En hexadécimal
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14
21	15

Exercices :

a) Écrire les nombres 27, 31, 84 et 128 en hexadécimal.

b) Donner la valeur des nombres dont l'écriture hexadécimale est 83_{16}, $A1_{16}$, FF_{16} et $A10E_{16}$.

III Conversions, changements de bases

A Conversion de la base b à la base décimale

On a déjà vu la méthode permettant de convertir un nombre écrit en base b en décimal : c'est la relation $(*)$ ci-dessus.

B Conversion de la base décimale à la base b

Pour convertir un nombre α écrit en base 10 en son écriture en base b, on effectue des divisions euclidiennes successives. la première donne :

$$(1)\ \ \alpha = bq_0 + a_0$$

avec $0 \leqslant a_0 < b$. Recommençons en divisant le quotient q_0 par b :

$$(2)\ \ q_0 = bq_1 + a_1$$

En reportant (2) dans (1), on obtient :

$$(3)\ \ \alpha = b\,(bq_1 + a_1) + a_0 = q_1 b^2 + a_1 b + a_0$$

Continuons en divisant q_1 par b : $q_1 = bq_2 + a_2$, ce qui donne, en reportant dans (3) :

$$\alpha = b^2\,(bq_2 + a_2) + a_1 b + a_0 = q_2 b^3 + q_1 b^2 + a_1 b + a_0$$

En continuant les divisions jusqu'à obtenir un quotient nul, on arrive à une égalité du type :

$$\alpha = a_n b^n + a_{n-1} b^{n-1} + \ldots a_2 b^2 + a_1 b + a_0$$

Ainsi, l'écriture en base b de α est :

$$\alpha = (a_n a_{n-1} \ldots a_2 a_1 a_0)_b$$

Le principe est donc d'écrire les restes successifs obtenus, de la droite vers la gauche.

Exemple : À titre d'exemple, convertissons 259_{10} en base 3 :

- $259 = 86 \times 3 + 1$,
- $86 = 28 \times 3 + 2$,
- $28 = 9 \times 3 + 1$,
- $9 = 3 \times 3 + 0$,
- $3 = 1 \times 3 + 0$,
- $1 = 0 \times 3 + 1$

Après cette succession de divisions, on relit les restes dans l'ordre inverse : $259_{10} = 100121_3$.

Exercice : Reprendre les conversions de la partie précédente en utilisant cette méthode, et comparer les deux méthodes.

C Conversion directe entre binaire et hexadécimal

On a signalé l'intérêt principal de l'hexadécimal pour manipuler des nombres binaires. On peut bien sûr passer par la base 10, mais il y a un moyen beaucoup plus rapide. Expliquons cela.

Une division (entière) par 16 en binaire revient à effectuer un décalage de quatre bits vers la droite. Ainsi, chaque paquet de quatre bits correspond à un chiffre hexadécimal. Il suffit donc de connaître l'équivalence entre les nombres de quatre bits en binaire et les "chiffres" hexadécimaux pour obtenir une conversion immédiate.

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

Binaire	Hexadécimal
1000	8
1001	9
1010	*A*
1011	*B*
1100	*C*
1101	*D*
1110	*E*
1111	*F*

Exemples :

- Convertissons le nombre binaire 101111110110110 en hexadécimal. Pour cela, on commence par découper le nombre en "paquets" de 4 bits à partir de la droite, en complétant éventuellement le dernier paquet

pour obtenir un bloc complet de 4 bits, puis on écrit en dessous le chiffre hexadécimal correspondant :

0101	1111	1011	0110
5	F	B	6

Ainsi, $101111110110110_2 = 5FB6_{16}$.

- Convertissons maintenant le nombre hexadécimal $FEC5$ en binaire. Il suffit pour cela d'écrire en dessous de chaque chiffre hexadécimal sa correspondance en binaire :

F	E	C	5
1111	1110	1100	0101

Ainsi, $FEC5_{16} = 1111111011000101_2$.

IV Annexe : représentation informatique des nombres

Remarque : cette section ne fait pas partie du programme, et doit être considérée comme uniquement culturelle !

L'unité élémentaire de l'ordinateur est le bit. Mais par soucis d'efficacité et de rapidité de traitement, les microprocesseurs modernes manipulent des mots constitués de plusieurs bits. Le premier microprocesseur commercialisé, l'Intel 4004, utilisait des mots de 4 bits. Puis vinrent les microprocesseurs 8 bits : le Z80, l'Intel 8080, le MOS 6502...

De nos jours, les microprocesseurs modernes manipulent des mots de 32 bits, voire de 64 bits pour les plus récents. On remarque que ce sont toujours des puissances de 2, et surtout des multiples de 8. Un mot de 8 bits est appelé un *octet*.

Bien entendu, on ne peut pas se contenter des nombres manipulables par le microprocesseur. On a parfois besoin de plus ou moins de précision. Les langages informatiques fournissent des types plus ou moins standards.

A Les entiers non signés

Les entiers non signés sont simplement codés sous forme de blocs de bits (ou plutôt d'octets), en binaire :

- si l'on manipule des entiers codés sur un octet (soit sur 8 bits), on peut coder les entiers de 0 jusqu'à 255 ;
- si l'on manipule des entiers codés sur deux octets (soit sur 16 bits), on peut coder les entiers de 0 jusqu'à 65535 ;
- si l'on manipule des entiers codés sur quatre octets (soit sur 32 bits), on peut coder les entiers de 0 jusqu'à 4 294 967 296.

Si l'on demande à un microprocesseurs d'ajouter 1 au plus grand entier codable, il renvoie[4]... 0 ! Attention donc aux *dépassements de capacité*.

Une règle utile pour obtenir rapidement une bonne approximation de la valeur d'une puissance de 2 : comme $2^{10} = 1024$ est très proche de $1000 = 10^3$, on a par exemple :

$$2^{32} = 2^{3\times 10+2} = \left(2^{10}\right)^3 \times 2^2 \approx \left(10^3\right)^3 \times 4 = 4 \times 10^9$$

soit approximativement 4 milliards[5].

B Les entiers signés

On a beaucoup plus souvent besoin de coder des entiers *signés*, pour représenter les entiers relatifs (ce qui n'empêche d'ailleurs pas de ne manipuler que des entiers positifs !). Pour cela, on consacre un bit (en général, le bit de poids le plus fort) au signe. Mais pour des raisons pratiques, on utilise un codage un peu spécial pour les entiers négatifs.

[4] et positionne un *bit de dépassement*, ou *"overflow"*, à 1, encore faut-il bénéficier de cette information et l'utiliser !

[5] D'autres approximations utiles, au passage : une année représente environ 31 millions de secondes, et un milliard de secondes représente environ 32 ans. Ce genre d'approximations permet de déterminer rapidement si un programme va terminer son calcul rapidement, ou bien tourner jusqu'à la fin de l'univers, qui devrait se produire dans environ 15 milliards d'années, soit à peu près un demi-milliard de milliards de secondes !

- Les entiers positifs sont simplement codés comme les entiers non signés, le bit de poids le plus fort étant positionné à 0. Si l'on dispose de n bits, le plus grand entier positif représentable est donc $2^{n-1}-1$, codé $0111\ldots111$.
- On pourrait coder les entiers négatifs de la même manière, mais cela rendrait l'algorithme d'addition de deux entiers signés plus complexe à implémenter dans le microprocesseur. On utilise donc un codage moins lisible pour un humain, mais plus efficace pour les calculs : le *complément à* 2.

  Pour cela, on prend le codage binaire de l'opposé du nombre, on inverse tous les bits (les 1 deviennent des 0 et vice-versa, on dit qu'on effectue un *complément à* 1), et on ajoute 1 au résultat.

Exemples :
- Le nombre 9924 est codé sur deux octets sous la forme 0010011011000100.
- Pour coder le nombre -9924 en complément à 2 sur deux octets, on inverse tous les bits, et on ajoute 1 :

$9924 =$	0010	0110	1100	0100
complément à 1	1101	1001	0011	1011
on ajoute 1+	0000	0000	0000	0001
$-9924 =$	1101	1001	0011	1100

Remarquons que ce système est cohérent avec ce qu'on a signalé tout à l'heure : si l'on ajoute 1 au plus "grand" nombre représentable, on obtient 0 :

$$0000\,0000\,0000\,0001 + 1111\,1111\,1111\,1111 = 0000\,0000\,0000\,0000$$

Ainsi, 1111 1111 1111 1111 doit être la représentation binaire de -1, ce qu'on peut facilement vérifier en reprenant la méthode de complément à 2.

Une soustraction de deux nombres a et b en binaire consiste donc à remplacer b par son complément à 2 b', et à calculer la somme $a + b'$.

C Les nombres en virgule flottante

Ces nombres seront vus en détail l'an prochain.