Représentation de l'information sur un ordinateur

Par K1wy, le 11 novembre 2010

Ce document a pour objectif d'expliquer les bases de la représentation d'informations en informatique. Ce papier traitera de la représentation des nombres avec pour finir une courte explication sur la représentation du texte, des images, etc.
Il est essentiel à mon sens de comprendre le fonctionnement d'un ordinateur (comment il traite les informations) et toute la logique et les méthodes qu'il utilise pour prétendre s'y connaitre en sécurité. Une bonne maison se fait toujours sur de bonnes fondations.

Les bases :

Cette section va vous présenter les bases. Tout d'abord, répondons à une question que l'on peut se poser :

Qu'est-ce qu'une base ?

Voici la définition (un peu ardue) donnée par wikipedia :

*En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres dans la numération N-adique, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre. Par commodité, on utilise usuellement, pour les bases entières à partir de deux, un nombre de chiffres égal à la base. En effet, l'écriture d'un nombre en base N à l'aide de N chiffres allant de 0 à N-1 correspond à son développement en base N.*

Pour bien comprendre, il faut savoir que classiquement nous utilisons la base la plus commune : la base décimale (ou base 10).

Comment interprétons-nous réellement un nombre ? Prenons par exemple le nombre 12345.

Ce nombre est égal à 10000 + 2000 + 300 + 40 + 5 (j'espère que personne ne me contredira sur ce point). Écris autrement, ce nombre est égal à $1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0 = 12345$ (Pour rappel, $10^0 = 1$).

Et bien voilà comment fonctionne une base : on lit le nombre de droite à gauche, et on multiplie le chiffre par la base (ici 10) puissance (la position). On commence à la position 0 et on augmente de 1 au fur et à mesure qu'on lit les chiffres.

Bien sûr il n'existe pas que la base 10 ! On peut écrire un nombre en n'importe quelle base à partir de 2. Exemple de nombre en base 8 : 3123 en base 8 qui est égal à $3*8^3 + 1*8^2 + 2*8^1 + 3*8^0$.

Dans ce-cas là on peut voir une règle s'imposer : chaque chiffre composant le nombre en base N doit être compris entre 0 et N -1. Concrètement, cela veut tout simplement dire que 888 n'existe pas en base 8. En effet 8 > 8-1(=7). Vous allez automatiquement passer à la puissance d'au-dessus. 8 est égal à 10 en base 8. Regardez : $8 = 1*8^1 + 0*8^0$.

En conséquence, au-dessus de la base 10, il vous faudra utiliser d'autre symbole pour représenter nos nouveaux chiffres. Exemple : en base 11, nous pouvons aller de 0 à « 10 ». Or 10 n'est plus un chiffre mais un nombre, ça risque de bloquer. On utilise pour cela les lettres. Donc en base 11, 10 serra traduit par A. On aura donc comme symbole à notre disposition {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A}. En base 16, on a donc {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Vous pouvez après utiliser les lettres grecs ou des symboles pour continuer arrivé à la lettre Z. Mais sachez que l'on va rarement au-dessus de la base 16 (mis à part la base 64, mais c'est une autre histoire).

Pourquoi les bases en informatique ?

On utilise principalement 4 bases en informatiques : les bases 2 (binaire), 8 (octale), 10 (décimale), et 16 (hexadécimal).

Tout d'abord la base 10, parce que c'est la plus commune et elle est universelle. Pour les autres bases c'est une autre histoire.

La base 2 :

Il faut revenir au fonctionnement de base d'un ordinateur. Un ordinateur est composé de différentes parties, dont des entrées (input) : votre clavier, un port USB, un lecteur CD... Et de sortie : carte graphique (CGU en anglais), un graveur, un port USB, des cartes PCI, … Il contient aussi d'autres composants tout aussi importants (si ce n'est plus) tels que la mémoire et le processeur (CPU en anglais). Le tout est relié par la carte mère. Nous nous intéresserons à la communication entre ces différents composants.

Cette dernière s'effectue de la manière suivante (simplifié à l'extrême, la prétention de cet article n'est pas de donner un cours d'électronique) : soit on envoie un courant avec une tension nulle, ou un courant avec une tension. Pour une explication plus complète de ce principe, voir en annexe (1).

On a donc 2 états : courant ou pas courant. Comment représenter cela numériquement (avec des nombres) ? Avec des 0 et des 1 : la base 2 bien sûr !

On peut utiliser 2 symboles en base 2 : {0, 1}. Par convention on identifie 0 comme une absence de tension, et 1 comme une présence de tension. L'ordinateur comprend donc uniquement des nombres en base 2. Exemple : 101101 , 01100110, 11111111, 10000001.

Ces 0 et 1 sont appelés bit pour **bi**nary digi**t.**

Les bases 8 et 16 :

Par convention on regroupe les bits par paquet de 8. Ces paquets de 8 bits sont appelés octet. Les informaticiens trouvaient plus utile de regrouper les bits par paquet de 8. C'était beaucoup plus facile. Imaginez vous écrire 125 en binaire. Cela donne 111110, c'est un peu long. Et en octal on a 175. (Nous verrons les conversions plus tard). C'est tout de même rudement plus cours ! La base 8 a été choisie car il fallait qu'elle soit une puissance de la base binaire 2. Elle permet de faire de la conversion par paquet (que nous verrons plus tard dans cet article).

La base 16 est utilisée pour réduire encore plus l'écriture des nombres. Elle est aussi une puissance de 2 ($16 = 2^4$). En reprenant notre exemple, on a 125 = 7D. C'est encore plus court. Elle permet de faire une conversion elle aussi très facile par paquet de 4 bits.

Pour le moment retenez que ces deux bases sont des écritures plus pratiques et plus compactes. Elles permettent aussi de faire des conversions depuis la base binaire très rapidement.

Conversion de la base 10 aux bases 2, 8 et 16 :

On va pour cela utiliser un algorithme très facile d'utilisation : l'algorithme d'Euclide. Il se base sur un principe mathématiques qui va remonter à votre plus tendre enfance : les divisions avec restes.

Exemples : 15 = 2 * 7 + 1 ou autrement dit : 15 / 7 = 2 reste 1.

Je vous passe l'explication de la division, je pense que pouvez vous en sortir.

L'algorithme d'Euclide :

C'est un algorithme très simple qui consiste à faire une suite de division avec reste jusqu'à qu'on arrive avec un quotient égal à 0. On va diviser notre nombre en base 10 par notre base B. On va ensuite prendre nos restes à chaque fois et on aura notre chiffre en base B.

Comme un exemple est beaucoup plus explicite qu'un paragraphe théorique, appliquons l'algorithme d'Euclide pour passer le nombre **241** en base **8**.

On commence par diviser 241 par 8. On trouve comme reste 1 et comme quotient 30.

Autrement dit (plus mathématiquement), on peut écrire : 240 = 8***30** + **1**.

On a donc :

240 = 8*30 + **1**

30 = 8*3 + **6**

3 = 8*0 + **3**

On lit maintenant nos restes dans l'ordre inverse. Cela nous donne donc $\mathbf{361_8}$ (le petit 8 indique dans quelle base le nombre est. Si aucun indice est présent, on suppose que nous sommes en base 10).

On a donc $\mathbf{240 = 361_8}$**.** Un autre pour bien comprendre le principe. Encore pour passer de décimal à octal. Convertissons 3 en octal :

3 = 8*0 + **3**

On a donc $\mathbf{3 = 3_8}$**.** Si vous avez tout compris, cela ne devrait pas vous surprendre.

Un dernier en base 8 : 32768

32768 = 8*4096 + **0**

4096 = 8*512 + **0**

512 = 8*64 + **0**

64 = 8*8 + **0**

8 = 8*1 + **0**

1 = 8*0 + **1**

On a donc $\mathbf{32768 = 10000_8}$**.** On aurait pu le calculer beaucoup plus rapidement en remarquant que $32768 = 8^5$, donc par définition $32768 = 1*8^5 + 0*8^4 + 0*8^3 + 0*8^2 + 0*8^1 + 0*8^0$. Mais ceci est un cas particulier.

Convertir en base 2 :

Pour convertir en base 2, on utilise le même algorithme. Un exemple rapide.

241 = 120 * 2 + **1**

120 = 60 * 2 + **0**

60 = 30 * 2 + **0**

30 = 15 * 2 + **0**

15 = 7 * 2 + **1**

7 = 3 * 2 + **1**

3 = 1 * 2 + **1**

1 = 0 * 2 + **1**

On a donc 241 = 11110001_2.

Un excellent petit jeu pour convertir rapidement de décimal à binaire et vice-versa est disponible en annexe (2).

Et en base 16 :

Toujours le même principe (qui marche pour n'importe quelle base).

241 = 15 * 16 + 1

15 = 0 * 16 + 15

En base 16, 15 est symbolisé par la lettre F. On a donc 241 = $F1_{16}$.

Conversion de base 2, 8, 16 à la base 10 :

Pour convertir un nombre d'une base B à la base 10, rien de plus simple. On retourne à notre définition d'une base. Vous vous rappelez : $1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0 = 12345$

Et bien pour toutes les autres bases, c'est pareil. On multiplie chaque chiffre par la base exposant sa position. Et encore une fois, des exemples :

$241_8 = 2*8^2 + 4*8^1 + 1 = 161$.

$111011011001_2 = 1*2^{11} + 1*2^{10} + 1*2^9 + 0*2^8 + 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 3801$

$1BF_{16} = 1*16^2 + B*16 + F = 1*16^2 + 11*16 + 15 = 447$

Jusqu'ici pas trop dur non ? Nous allons voir la conversion d'une base B_1 à une base B_2. (Ces 2 bases étant différentes de 10).

Conversion des bases 2, 8, 16 aux bases 2, 8, 16 :

Pour passer d'une base B_1 à une base B_2, il suffit de repasser par la base 10 au milieu.

Un exemple très rapide car ce point n'est vraiment pas compliqué :

Convertissons 241_8 en base 16 :

Passage en base 10 :

$241_8 = 2*8^2 + 4*8 + 1 = 197$

Passage en base 16 :

$197 = 16 * 12 + 5$

$12 = 0 * 16 + C$

Donc $241_8 = C5_{16}$.

Conversions par paquets :

On va ici utiliser une subtilité des bases utilisées. Vous vous rappelez lorsque l'on parlait de l'intérêt de l'usage de la base 8 et de la base 16, je vous avais dit d'attendre. Et bien ici vient l'explication.

Comme vous le savez, $8 = 2^3$ et $16 = 2^4$. En utilisant cette propriété, on peut faire une conversion d'un nombre binaire en des nombres en bases 8 et 16 beaucoup plus facilement qu'en repassant par la base 10.

Pour convertir en base 8, on va prendre les bits par paquet de 3, et trouver la valeur correspondante.

Exemple : convertissons $010111000100110010_2 = 10111000100110010_2$ (les zeros à gauche du 1 final sont superflus, comme en base 10 on écrit 4 et non 04) en base 8 (à lire de droite à gauche).

$010\ 111\ 000\ 100\ 110\ 010_2$

$2\quad 7\quad 0\quad 4\quad 6\quad 2_8$

On a donc découpé le nombre en paquet de 3 bits, puis on a trouvé le chiffre correspondant à chaque paquet en base 8. $10111000100110010_2 = 270462_8$

Pour la base 16, c'est presque pareil, sauf que ce coup-ci on fait des paquets de 4 bits (car $16 = 2^4$).

11111000100110010_2 en base 16 ;

$1\ 1111\ 0001\ 0011\ 0010_2$

$1\quad F\quad 1\quad 3\quad 2_{16}$

Donc $11111000100110010_2 = 1F132_{16}$

Vous comprenez maintenant pourquoi les informaticiens ont choisi ces deux bases à l'époque ? Tout simplement pour leur permettre de faire des conversions beaucoup plus facilement.

Exercices :

Convertir en base 2, 8 et 16 :

- 1
- 153
- 8
- 654
- 255
- 256
- 9999
- 145632

Convertir en base 10 :

- 1_2
- 153_{16}
- 153_8
- 255_8
- $ABCD_{16}$
- 1001110010_2

Convertir (en passant par la base 10 et par paquet) :

- 1_2 en base 16
- 123_8 en base 2
- FF_{16} en base 8
- $DCBA_{16}$ en base 8
- 111111_2 en base 8 et 16
- 101010101010_2 en base 8 et 16

Voilà la fin de cette première partie. Elle peut paraître un peu « lente », mais il était nécessaire de bien poser les bases. Nous allons voir maintenant comment sont codés les nombres signés (avec un signe, + ou -) par l'ordinateur.

Une histoire de signe :

Dans cette partie nous allons aller un peu plus vite. Comme vous le savez, les mathématiques ne se résument pas à des entiers naturels (de 0 à +infini) mais ils comportent aussi les entiers relatifs (de -infini à + infini), c'est-à-dire (entre autre) les nombres négatifs.

Il a fallu alors trouver un moyen de coder des nombres négatifs en mode binaire. Une méthode a alors été inventée : le complément à 2.

Il fallait pouvoir effectuer des opérations très simplement tel que l'addition de deux nombres négatifs. Soit deux nombres entiers relatifs 4 et -4 (base 10).

Comme vous le savez désormais, la représentation binaire du nombre 4 est 00000100_2. On doit avoir un nombre xxxxxxxx (on travaille sur un octet, soit 8 bits) tel que :

00000100_2

\+ $xxxxxxxx_2$

00000000_2

On va pour cela utiliser une astuce et utiliser un $9^{ème}$ bit qui ne serra par la suite pas comptabilisé par l'ordinateur. On a donc maintenant :

00000100_2

\+ $xxxxxxxx_2$

100000000_2

Je vous laisse trouver $xxxxxxxx_2$ par vous-mêmes. Vous devriez trouver 11111100_2. En effet, $00000100_2 + 11111100_2 = 100000000_2$

Le neuvième bit étant ignoré, on a donc bien $00000100_2 + 11111100_2 = 00000000_2$

Cette méthode permet de réaliser des calculs très rapide avec les nombres négatifs.

Note : nous travaillons ici avec des nombres codés sur 1 octet. Si le nombre prend plus de place, il faut savoir que la même méthode et que le 1 qui arrive sur l'octet non prévu au départ serra lui aussi ignoré.

Processus :

Voilà une méthode automatique pour générer un nombre négatif à partir de son nombre positif :

- Inverser tous les bits (les 1 deviennent 0 et les 0 deviennent 1) : cette méthode est appelée complément à 1
- Ajouter 1 au nombre obtenu par l'étape précédente

Voici deux exemples pour mieux comprendre.

Exemples :

Prenons le nombre 16. On a 16 = 00010000_2

On fait le complément à 1 : 00010000_2 devient 11101111_2

On ajoute 1 :11101111_2 devient 11110000_2

On a donc -16 codé par 11110000_2 en binaire.

Prenons le nombre 125. On a 125 = 01111111_2

On fait le complément à 1 : 01111111_2 devient 10000000_2

On ajoute 1 : 10000000_2 devient 10000001_2

On a donc -125 codé par 11110000_2 en binaire.

Exercices :

Ecrire en binaire les nombres négatifs suivant (attention aux bases de départ) :

- -111111
- -9999
- -145632
- -123_8
- $-FF_{16}$
- $-DCBA_{16}$

Et les virgules ?

Nous allons nous interesser ici à des nombres codés sur 4 octets (type float sur une architecture 32 bits). Il faut savoir que pour les nombres à virgule flottante (d'où le float), il n'y a pas deux conventions différentes pour les nombres positifs et les nombres négatifs, mais bien une seule.

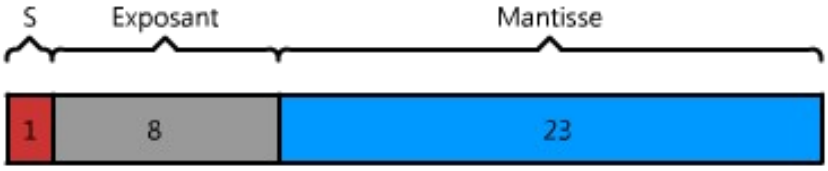
On va utiliser la convention dites IEEE. Tout d'abord, posons-nous la question : qu'est-ce qu'un nombre décimal (ou nombre à virgule) ?

Dans les décimaux, il y a deux parties : une entière et une décimale. Les deux étant séparés par une virgule. Exemple : 53,42. Partie entière : 53, partie décimale : 42.

Ces nombres peuvent aussi être écrit sous forme scientifique. C'est-à-dire sous la forme x, yyyyyyyy * 10 ^ e. Ici x est la partie entière (toujours un chiffre), yyyyyyyyy est la partie décimale et e est l'exposant. Par exemple pour 53,42 on obtient 5,342 * 10 ^ 1.

On va donc utiliser l'écriture scientifique pour représenter nos nombres couplée à la norme IEEE.

Voici un schéma qui représente comme un nombre à virgule est représenté :



Made with **lovelycharts.com**

On a donc (**en base 2**):

- 1 bit destiné au signe (S) : 0 pour positif et 1 pour négatif
- 8 bits destinés à l'exposant : il faut savoir qu'on doit ajouter 127 à l'exposant, ce qui permet d'avoir des nombres allant de de -126 à 127
- 23 bits pour la mantisse.

Qu'est-ce que la mantisse :

La mantisse est la partie décimale d'un nombre binaire. On peut donc avoir une partie décimale allant de 0 à 2^23 = 8 388 608, soit 7 chiffres après la virgule.
En binaire la partie entière importe peu puisqu'elle sera forcément égale à 1 (je vous rappelle qu'on ne prend pas en compte les 0 à gauche du premier 1).

Une petite règle sur les exposants :

Voici une petite règle sur l'exposant qu'il est nécessaire de respecter :

- l'exposant 00000000 est interdit
- l'exposant 11111111 est interdit. Il sert seulement à signaler les erreurs, et nous sommes dans le cas dit NaN (Not a Number).

Exemples :

Pour ceux qui auraient un peu de mal, voici deux exemples qui illustrent les propos ci-dessus :

Prenons le nombre 123,444.

On a 123,44 qui donne +1111011,110111100 (123 = 1111011_2 et 444 = 110111100_2).

On passe tout ceci en écriture scientifique on a donc :

+1111011,110111100 = +1,111011110111100*10^6

A partir de là nous avons déjà deux éléments sur trois : le signe, +, qui est codé par 0. Et la mantisse , 111011110111100_2, auquel nous ajoutons des 0 jusqu'à arriver à 23 bits. Ce qui nous donne : $11101111011110000000000_2$.

Il nous reste à calculer l'exposant : 127 + 6 = 133 = 10000101_2

Nous avons tous nos composants. On assemble le tout :

123,444 = $01000010111101111011110000000000_2$

Prenons le nombre -58749,663221.

On a -58749,663221 qui donne -1110010101111101,10100001111010110101 (58749 = 1110010101111101_2 et 663221 = 10100001111010110101_2).

-1110010101111101,10100001111010110101=-1,11001010111110110100001111010110101*10^15
Pour l'exposant : 127 + 15 = 142 = 10001110_2
On a donc :

- Signe : 1_2 car négatif
- Exposant : 10001110_2
- Mantisse : $11001010111110110100001111010110101_2$

Et là, il y a un problème... La mantisse fais plus de 23 bits. Nous allons donc être obligé de négliger des chiffres après la virgule. C'est la limite de représentation des nombres à virgules par un ordinateur : ils ne peuvent contenir qu'un nombre finis de chiffres. Nous avons donc notre nouvelle mantisse : $11001010111110110100001_2$.

-58749,663221 = $11000111011001010111110110100001111010110101_2$

Note : c'est sans doute la représentation la plus difficile à comprendre. N'hésitez-pas à relire et à faire certains exercices.

Exemples :

Donner la notation binaire des nombres suivants :

- 1,03
- -5,02
- 1542,22
- 0,0005
- -0,000658

Et le texte, les images, les sons ?

Le texte :

Dernière partie de cet article, nous allons parler très brièvement de l'encodage du texte, des images et du son.

Nous avons vu comment sont codés les nombres par notre ordinateur. Nous savons-donc maintenant comment est codé un nombre positif, négatif ainsi que les réels. Mais il a fallu trouver un moyen de représenter du texte pour notre ordinateur. Nous avons pour cela utilisé le codage ascii.

On fait tout simplement correspondre à chaque lettre, chiffre (et autre caractères spéciaux) un nombre de 1 à 255. Par exemple le A (majuscule) vaut 65, le s (minuscule) vaut 115.

Pour écrire le mot « nombre » on aura donc la séquence 110 111 109 98 114 101. La liste de correspondance nombre → lettre est contenue dans une table ascii (3).

Mais le problème est que cette table a été inventée par les Américains : les accents sont donc inexistants. De plus certaines langues ont des alphabets non-latin (l'alphabet cyrillique par exemple). Pour résoudre ce problème, les charsets ont été inventés. Un charset est une table de correspondance entre un nombre et une lettre. Il existe plusieurs dizaines de charsets pour différentes langues et différents alphabets. Pour plus d'information, veuillez consulter le lien (4) en annexe.

Les images, les sons, … :

Et pour chaque type de donnée que nous avons l'habitude d'utiliser tous les jours (images, sons, vidéos, …) il a fallu trouver un codage qui puisse permette à l'ordinateur de comprendre ce que c'est. L'étude de ces différents codages sont en dehors de la visée de cet article, mais vous pouvez en savoir plus en annexe (5) (6) (7).

Conclusion

Voilà j'espère que cet article vous a plus et vous a fait un peu plus comprendre le principe du codage del'information. Il existe des tas de formats pour différents types de données. A vous de trouver celui qui vous intéresse. N'hésitez-pas à me renvoyer un feedback à l'adresse k1wy.mail@gmail.com.

Annexe :

(1) http://fr.wikipedia.org/wiki/Circuit_%C3%A9lectronique

(2) http://forums.cisco.com/CertCom/game/binary_game_page.htm

(3) http://www.asciitable.com/

(4) http://fr.wikipedia.org/wiki/Charset

(5) http://fr.wikipedia.org/wiki/Codec_vid%C3%A9o

(6) http://fr.wikipedia.org/wiki/Format_audio

(7) http://fr.wikipedia.org/wiki/Format_de_fichier#Formats_d.27image