Codage et typage

1. À l'intérieur d'un microprocesseur…

Dans un ordinateur, il n'y a que des 0 et des 1… si l'on peut dire, puisqu'en fait il n'y a que du courant électrique, qui passe plus ou moins bien dans les circuits ! Quand il passe bien, on dit que c'est 1, quand il passe mal ou pas, on dit que c'est 0. On aurait pu dire blanc et noir tout aussi bien, c'est un choix, pas tout à fait arbitraire cependant comme on va le voir ci-dessous.
On doit donc trouver une correspondance, un code (pas secret !), pour faire correspondre ces chiffres aux données que l'on manipule.
Puisque l'ordinateur travaille avec des 0 et des 1, on utilise naturellement la base 2, qui n'utilise que les chiffres 0 et 1 pour représenter les nombres :

Base 10	0	1	2	3	4	5	6	7	8
Base 2	0	1	10	11	100	101	110	111	1000

La valeur 0 ou 1 s'appelle un *bit* (binary digit). Ces valeurs sont groupées suivant des puissances de 2, en effet là aussi c'est plus simple pour l'ordinateur de gérer les circuits suivants des puissances de 2. 8 bits forment un *octet*. Un byte est souvent un octet, mais pas toujours… je vous déconseille d'utiliser ce terme tant que vous n'en aurez pas la définition exacte (qui dépasse le cadre de l'ISN). Un *mot* est l'unité de base manipulée par un microprocesseur. Les ordinateurs actuels utilisent des mots de 8, 16, 32 ou 64 bits (respectivement appelés octet, mot, double mot, quadruple mot). Vous pouvez constater que « mot » est mal défini…

2. Les types et leur représentation.

Les données sont d'un certain type : nombre, caractère,… Les types de données rencontrés dans les différents langages de programmation se recoupent en grande partie. Les données sont toutes traduites en 0 et 1 dans la machine, mais le codage associé à un type permet de « comprendre » la donnée.

a. Les booléens.

Un booléen est en théorie un bit : Vrai ou Faux. On utilise généralement le codage 0 pour faux et 1 pour vrai, comme par exemple en logique ou en calcul booléen (que l'on verra dans quelques séances).
Typage : En Python, le type booléen « bool » est en fait un entier, qui vaut False pour 0 et True dans tous les autres cas. Ce qui peut donner des résultats surprenants.

b. Représentation des entiers (positifs).

Ils sont traduits en base 2, sur 32 bits au moins .La valeur maximale n'est pas $2^{32}-1=4294967295$, comme on pourrait le croire. En effet il faut aussi pouvoir représenter les entiers négatifs (cf paragraphe suivant). Le codage interne à l'ordinateur est classiquement sur un mot de 4 octets, pour un entier de 32 bits. En Python, les entiers sont aussi grands que l'on veut : si nécessaire, Python prend le nombre de mots nécessaires pour coder l'entier.
Typage : type plain integer, « int ». On trouve parfois dans des vieux programmes des entiers « longs » de type long integer, « long ».

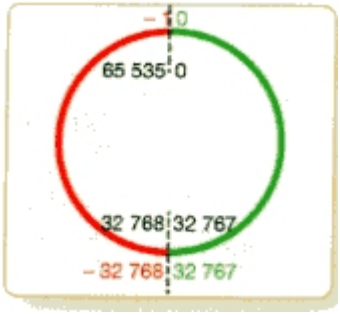
c. Les entiers (avec les négatifs).

On représente un entier relatif par un entier naturel… avec la convention que l'on expose ci-après, dite du complément à deux.
Supposons que l'on travaille avec des mots de 16 bits. Si l'on ne représente que des entiers positifs, leur valeur ira de 0 à $\overline{1111111111111111}^{2}=2^{16}-1=65535$. Si l'on souhaite représenter des entiers relatifs, on gardera le premier bit pour donner le signe. Il restera 15 bits pour la valeur absolue de l'entier. On pourra donc représenter les entiers de -32768 à 32767.

Remarque préliminaire : représentons le premier bit par « + » ou « - », plutôt que par 0 ou 1, puisque d'une part nous voulons des entiers signés, et d'autre part la représentation des bits par des symboles est arbitraire. L'idée « naturelle » est alors de poser $\overline{+111111111111111}^2 = 2^{15}-1 = 32767$ et $\overline{-111111111111111}^2 = -\left(2^{15}-1\right) = -32767$. Cette méthode a plusieurs inconvénients, le plus visible étant les deux écritures de 0 : $\overline{0000000000000000}^2$ et $\overline{1000000000000000}^2$, où le premier 1 (respectivement 0) représente le + ou le -. Par ailleurs, essayez d'ajouter un nombre positif et un nombre négatif avec ce codage : le résultat est surprenant…

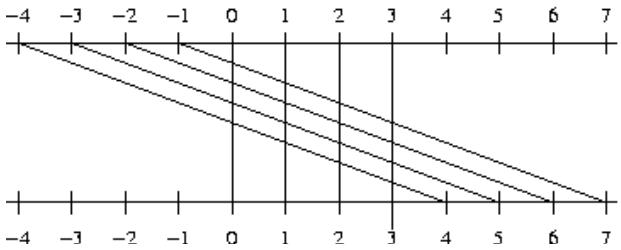
Principe de la notation en complément à deux.
On représente les entiers positifs de manières standard

- 0 est représenté par $\overline{0000000000000000}^2$;
- 1 par $\overline{0000000000000001}^2$;
- etc jusqu'à $\overline{0111111111111111}^2 = 2^{15}-1 = 32767$
- Quand on ajoute 1, on passe alors à $\overline{1000000000000000}^2 = 2^{15} = 32768$ si l'on considère des entiers positifs. Dans le cas d'un entier négatif, puisque le premier bit représente le signe « - », on considérera en fait que $\overline{1000000000000000}^2 = -2^{15} = -32768$.
- Puis le suivant : $\overline{1000000000000001}^2 = -\left(2^{15}-1\right) = -32767$
- Ainsi de suite jusqu'à $\overline{1111111111111111}^2 = -1$



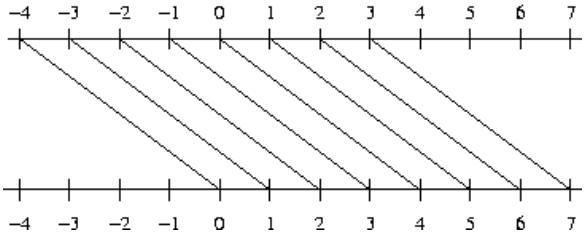
On peut considérer que l'on tourne sur un cercle comme ci-contre.
En résumé, un entier positif n est représenté par lui-même. Un entier négatif n est représenté par $n+2^{16}$ (avec des mots de 16 bits).

Autre représentation : le schéma ci-dessous donne la représentation en complément à deux pour des mots de 3 bits. Sur la première ligne, on a les nombres à représenter, compris entre -4 et 3. Sur la deuxième ligne figure leur codage, entre 0 et 7.



d. <u>Les nombres réels.</u>
On ne peut représenter que des valeurs décimales, vu qu'on ne dispose que de mots de longueur finie. $\sqrt{2}$ et π ne peuvent pas être représentés exactement.
Les réels sont codés sous la forme de la représentation scientifique : $x = (-1)^s \times m \times b^e$ dite mantisse-exposant.
Par exemple $-100\pi \simeq (-1)^1 \times 3{,}14149 \times 10^2$ en décimal.
En informatique, cette écriture est d'abord convertie en binaire, et on l'écrira sous la forme $x = (-1)^s \times m \times 2^e$

Pour un mot de 64 bits, en norme IEEE 754, on aura 1 bit pour le signe (0 pour + et 1 pour -), 11 bits pour l'exposant et 52 bits pour la mantisse. Comme la mantisse, en base 2, commence forcément par un 1, on l'omet et on garde les 52 bits pour les chiffres après la virgule.
L'exposant est compris entre -1022 et 1023, on le représente comme l'entier naturel $n+1023$, compris entre 1 et 2046. On réserve 0 et 2047 pour les situations exceptionnelles (+∞ , -∞, NaN : Not a Number pour les divisions par 0 entre autres). Cette représentation, différente de celle des entiers relatifs, est appelée représentation biaisée.



Représentation biaisée pour des exposants sur 3 bits.

Remarque : une autre norme (IEEE 754) en simple précision.
Pour un mot de 32 bits, on aura 1 bit pour le signe (0 pour + et 1 pour -), 8 bits pour l'exposants et 23 bits pour la mantisse.
Comme la mantisse, en base 2, commence forcément par un 1, on l'omet et on garde les 23 bits pour les chiffres après la virgule.
Les exposants 00000000 et 11111111 sont réservés pour les situations exceptionnelles (+∞ , -∞, NaN : Not a Number).
Les exposants vont donc de 00000001 à 11111110. Il sont représentés avec un décalage de $127 = 2^7 - 1$.
L'exposant est compris entre -126 et 127 ; 00000001 représente $1 - 127 = -126$ et 11111110 représente $254 - 127 = 127$

Exemple : Codage de π sous la forme d'un flottant en simple précision.

Typage : type « float » en Python. On peut même représenter les complexes en Python, c'est un des rares langages le permettant. On utilise j et non i, et on fait systématiquement précéder j d'un nombre, sinon il est considéré comme une variable. Par exemple, pour écrire $3+i$, on tapera `3 + 1j`.

e. Codage des caractères

Pour permettre les échanges d'informations textuelles entre systèmes informatiques, le codage ASCII a été proposé en 1963. Chaque caractère est associé à un mot de 7 bits. Par exemple R est associé à 1010010 et le symbole 5 à 0110101. Par commodité, chaque caractère est en fait codé dans un octet. Sont codés en plus des caractères de contrôle, comme le « line feed », saut de ligne et « carriage return », retour chariot. Ces deux caractères font passer à la ligne, et donnent des comportements différents sous Windows ou sous Unix (système à la base de Linux et d'OsX).

Comme l'ASCII ne permettait pas le codage des lettres accentuées, d'autres normes de codage sont apparues (Windows cp 1252, MacRoman, ISO 8859-15). Par ailleurs, la présence d'un bit « libre » dans les octets du codage ASCII permet de représenter plus de caractères. La majorité de ces normes intègrent l'ASCII, mais ne sont pas compatibles entre elles. Comme il n'y a pas moyen de savoir quel codage est utilisé pour un document, cela donne parfois des surprises à la lecture, avec des caractères étonnants.
Actuellement la norme qui tend à s'imposer, venant de Linux et du HTML, est l'UTF-8. Elle est présente par défaut sur les dernières versions de Windows et d'OsX. La principale différence par rapport aux normes précédentes est qu'un caractère est représenté par un mot de 1 à 4 octets, de longueur variable.

Typage : aussi bien les caractères isolés, que les chaînes de caractères sont de type « str » (string, qui veut aussi dire chaîne en anglais). D'autres langages distinguent les caractères isolés des chaînes.

Exercices (entiers positifs)

1. Trouver la représentation en base 2 des nombres 1, 3, 7, 15, 31 et 63. Expliquer le résultat.
2. Trouver en base 10 la représentation du nombre $\overline{10010110}_2$ (la notation $\overline{xxxx}_2$ signifiant : nombre exprimé en base 2)
3. Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite. Quelle est l'opération équivalente en base 2. Le vérifier sur 3, 6 et 12.
4. Calculer en binaire la somme $\overline{1101101} + \overline{1001011}$. On fera comme en primaire, avec des retenues.
5. Comment peut-on multiplier un nombre en binaire par 2 ? Le diviser par 2 ? Que peut-il se passer dans ce dernier cas ?
6. Écrire en langage naturel un algorithme permettant d'ajouter deux entiers exprimés en binaire. L'implémenter en Python, en utilisant des tableaux de 0 et de 1 pour représenter les nombres en binaire.

Exercices (entiers relatifs) (codage sur 8 bits sauf mention contraire)

7. Quels nombres peut-on représenter avec des entiers de 8 bits ? De 32 ou 64 bits ?
8. Trouver la représentation décimale des entiers relatifs dont la représentation en binaire est $\overline{00110010}$, $\overline{10000000}$ et $\overline{10110011}$.
9. Coder -23 et -78.
10. Calculer la représentation binaire de 4, puis de -4, sur des mots de 8 octets. Vérifier que l'on obtient la représentation de -4 à partir de celle de 4 en inversant les 1 et les 0, puis en ajoutant 1.
11. Appliquer la méthode de l'exercice précédent pour trouver la représentation de -16. Vérifier que « ça marche ».
12. Représenter les entiers relatifs 96 et 48 sur 8 bits, et ajouter les résultats en base 2. Quel est l'entier relatif obtenu ? Que remarquez-vous ? Expliquer.
13. Comment faire une soustraction à partir d'un part, de la méthode d'addition en binaire, et d'autre part, de la représentation des entiers relatifs ? Appliquer à 15 - 7 (sur des mots de 8 bits)

Exercices (représentation des décimaux). On considèrera que les entiers sont codés sur 8 bits et les flottants sur 32 bits.

14. Trouver le décimal représenté par le mot : 0100 0011 0101 0100 1110 0000 0000 0000
15. Coder 141,42 sur 32 bits.
16. Comment est représenté le nombre décimal 2^{-122} ?
17. Comment est représenté le nombre entier 7 ? Et le nombre décimal 7,0 ?
18. Comment est représenté le nombre 0,1 ? que remarquez-vous ?
19. Quelle précision perd-on si on divise à nombre à virgule par 2, puis qu'on le multiplie à nouveau par 2 ? (cet exercice a plusieurs réponses !)

Exercices (représentation des caractères et chaines)
En Python, les chaînes de caractères sont traitées en partie comme des tableaux. Les fonctions ord() et chr() permettent de travailler en Python sur les caractères.
Exemple :

```
a = « bonjour »
premierCar = a[0]          # premierCar contient « b »
ord(premierCar)            # renvoie 98, le code ASCII de « b »
a + «  les petits »        # renvoie « bonjour les petits » ; c'est une opération de concaténation de chaîne
a + chr(65)                # renvoie bonjourA, c'est à dire « bonjour » concaténée avec le caractère dont le code unicode vaut 65 : A
```

20. Ecrire un programme en Python qui, étant donné une chaîne de caractères, renvoie un tableau contenant les codes unicode correspondants. Et un autre qui fait le contraire (pour tester ce dernier, n'utiliser que les codes de 65 à 90 –majuscules-, 97 à 122-minuscules-, 32 –espace-)