RND. Nombres entiers

I/ Un peu d'histoire

A l'ère du numérique, seuls les chiffres 0 et 1 sont utiles et permettent de représenter parfaitement des nombres entiers. Mais le chiffre 0 est apparu bien après les chiffres 1 ; 2 ; 3 etc.

Rappel : un **nombre** est une quantité, un **chiffre** un symbole.

A partir de trois siècles avant notre ère, les Babyloniens utilisaient un signe pour signaler une absence. Pour le reste, le bon sens (ordre de grandeur) permettait de déterminer les quantités écrites. Ils utilisaient un système à la fois décimal et sexagésimal.

Le système de numération romaine associait des quantité avec des lettres. Ainsi, un « I » représentait la quantité 1, un « V » la quantité 5, un « X » représentait la quantité 10, un « L » la quantité 50, un « C » la quantité 100 etc. On comprend aisément que ce système de numération n'est pas favorable aux opérations, même une simple addition.

Les Grecs, comme Thalès et Pythagore n'utilisaient pas le symbole du zéro non plus. Il faudra attendre le **5ème siècle** pour que le **zéro** commence à être introduit, en Inde, en tant que chiffre et nombre. Ce nombre sera utilisé rapidement dans le monde musulman, période au cours de laquelle il y a eu de nombreux scientifiques arabes alors que l'Occident, sous le système de numération romaine jusqu'au 12ème siècle n'en aura pratiquement aucun !

C'est donc à partir du 12ème que le zéro arrive en Europe grâce à Fibonacci, un mathématicien italien qui l'a découvert en Afrique du Nord et eu accès aux ouvrages du mathématicien persan Al Khawarizmi.

Le mot « *zéro* » vient de l'arabe « *sifr* », traduction du mot indien « *sunya* » qui signifie « *vide* ». Bien sûr, l'arabe « *sifr* » est aussi à l'origine du mot « *chiffre* ».

Pour en savoir plus sur les systèmes de numérations de l'Antiquité : https://www.math.univ-angers.fr/~labatte/l3sen/Cours/TD_NUMeRATION.pdf

Pour en savoir plus sur l'histoire du zéro et du un (vidéo) : https://www.youtube.com/watch?v=kjxPNH8CYVA

II/ Notion de base

1/ La base dix

On utilise depuis des siècles la base « dix ». Le nombre qui s'énonce « deux mille cinq cents trente-quatre » s'écrit en base 10 : 2534.

Pour l'écriture dans cette base, 10 chiffres sont nécessaires, de 0 à 9 inclus. Grâce aux puissances, on peut écrire 2534 d'une autre façon :
Deux mille : $2 \times 1000 = 2 \times 10^3$
Cinq cents : $5 \times 100 = 5 \times 10^2$
Trente : $3 \times 10 = 3 \times 10^1$
Quatre : $4 \times 1 = 4 \times 10^0$

Rappel : tout nombre à la puissance zéro vaut 1, y compris zéro.
Ainsi, on a : $2534 = 2 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$, **en base dix, on notera 2534_{10}**.

Sous forme de tableau, on peut alors écrire le nombre 2534 comme cela :

…	10^3	10^2	10^1	10^0
…	2	5	3	4

L'utilisation du chiffre zéro est déterminante et permet d'indiquer une absence de valeur (pas de dizaines par exemple).

2/ La base deux

En base deux, il n'y a que deux chiffres, 0 et 1, et le principe est le même qu'en base 10.
Comment écrire un nombre initialement en base dix en base deux ?

Méthode 1 : on écrit le nombre en question comme une somme pondérée de puissance de deux successives.
Rappel : $2^0 = 1$; $2^1 = 2$; $2^2 = 4$; $2^3 = 8$; $2^4 = 16$; $2^5 = 32$ etc.

Considérons le nombre 14.
On voit que $14 = 8 + 4 + 2$ soit $14 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$.

Sous forme de tableau, on peut alors écrire le nombre 14 comme cela :

…	2^3	2^2	2^1	2^0
…	1	1	1	0

En base deux, 14 s'écrit alors 1110_2.

L'inconvénient de cette méthode est qu'il faut tâtonner un peu pour trouver cette somme composée exclusivement de puissances de deux.

Méthode 2 : on pose une succession de division euclidienne par 2, jusqu'à obtenir un quotient égal à 0.
On reprend le nombre 14.

14 = 7 x 2 + 0, le reste vaut 0
7 = 3 x 2 + 1, le reste vaut 1
3 = 1 x 2 + 1, le reste vaut 1
1 = 0 x 2 + 1, le reste vaut 1.

En « remontant », on voit que $14 = 1110_2$.

A savoir : Dans la notation binaire, le **bit le plus à gauche** est appelé **bit de poids fort** et le **bit le plus à droite** est appelé **bit de poids faible.**

Remarque **:** multiplier un nombre en binaire par deux revient à lui ajouter un zéro à droite du nombre.
Ainsi $1011_2 \times 10_2 = 10110_2$ par exemple.

3/ Une base quelconque

On peut généraliser à une base quelconque les méthodes précédentes.

Pour écrire les entiers naturels en base b, on a besoin de b chiffres. Il existe nombres naturels a_i compris entre 0 inclus et b exclus tel que, pour tout entier naturel n, on a :

$n = a_i \times b^i + a_{i-1} \times b^{i-1} + a_{i-2} \times b^{i-2} + ... + a_2 \times b^2 + a_1 \times b^1 + a_0 \times b^0$. L'écriture de n en base b est alors $(a_i\, a_{i-1}\, a_{i-2} \ldots a_2\, a_1\, a_0)_b$.

Remarque : le produit d'un nombre par b revient là aussi à ajouter un zéro à droite de ce nombre.

Exemple :

La base 5 utilise par exemple 5 chiffres 0, 1, 2, 3 et 4.
Déterminons 344 en base 5.
344 = 68 x 5 + 4
68 = 13 x 5 + 3
13 = 2 x 5 + 3
2 = 0 x 5 + 2

344 s'écrit 2334_5.

La base seize est particulièrement intéressante. On a besoin de 16 chiffres, ceux de 0 à 9 puis A(10), B(11), C(12), D(13), E(14) et F(15).

Si on travaille en base deux, l'écriture peut être très longue. Or, un **octet** (8 bits) s'écrit très simplement en base seize.
Par exemple, 11010101_2 en base deux s'écrit D5 en base seize. En effet, on a :
$11010101_2 = 1101_2 \times 10000_2 + 0101_2 = 1101_2 \times 10_{16} + 0101_2 = D_{16} \times 10_{16} + 5_{16} = D5_{16}$.

Ainsi, on peut coder un fichier en format hexadécimal (base seize) qui prendra quatre fois moins de place que le même en binaire : cela peut être utile pour des fichiers photos en format RGB par exemple.

III/ Représentation des entiers naturels en machine

Dans une machine, on utilise l'écriture binaire pour représenter un entier naturel.

Avec des octets, soit huit bits, on peut représenter les entiers naturels de 0 (00000000 en base deux) à 255 (11111111 en base deux). Donc 45 est représenté par 00101101.

Sur seize bits, on représentera 45 par 0000000000101101.

A savoir : D'une manière générale, **avec *n* bits**, on peut représenter les **nombres entre 0 et $2^n - 1$**.

Les ordinateurs ne connaissent réellement qu'une opération, l'addition. **Cela suffit car toutes les autres opérations se déduisent de l'addition.**
Pour additionner deux nombres en binaire, on procède comme en base dix.
A partir de 0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 0 = 1 ; 1 + 1 = 10, on pose l'addition comme en base dix avec le système de retenue.

Exemple :

		1	1	0	1
+		1	0	0	1
		1		1	
	1	0	1	1	0

treize
neuf
retenues
vingt-deux

A savoir : Si la **taille des entiers est limitée**, par exemple avec quatre bits, alors dans l'addition ci-dessus, **le bit de poids fort est perdu !!** Dans ce cas, 13 + 9 donnera 0110 soit 6 (et non 22). On parle de **dépassement de capacité** (*overflow* en anglais).

Voici un lien vidéo « Comprendre le système binaire » reprenant les bases du système binaire :
https://www.youtube.com/watch?v=Tp6-w3j2MP4
Auteur : Yvan Monka, durée : 19 min 21 sec

Ces **dépassements de capacités** ont eu (et auront encore !) des conséquences parfois graves :

- **Vol 501 d'Ariane V** : Le 4 juin 1996, la fusée Ariane V a été détruite en vol après 37 secondes après son lancement, à la suite d'un problème dans la centrale inertielle. Déjà utilisé sur Ariane IV, une fusée bien plus petite, le système a eu un comportement inattendu avec Ariane V.
  Initialement codée comme un nombre sur 64 bits, cette valeur de l'accélération était convertie, durant le processus de contrôle, en un nombre sur 16 bits. La valeur de l'accélération étant trop grande, la conversion du nombre a échoué provoquant une réaction en chaîne dévastatrice et ... l'autodestruction de la fusée).

  Un rapport très détaillé ici mettant très clairement en cause le logiciel et la mauvaise plage des valeurs :
  http://deschamp.free.fr/exinria/divers/ariane_501.html

- **Le bug de l'an 2000 :** Ce bug s'appuyait sur une représentation sur deux chiffres d'une année au lieu de quatre chiffres comme actuellement. Ainsi, l'an 2000, représenté par 00, revenait à l'an ... 1900 pour une machine.

- **Le bug de l'an 2038 :** Les machines UNIX suivent la norme POSIX qui spécifie, entre autres, que le temps est compté en secondes à partir du 1er janvier à 00 :00 :00 temps universel. De nombreux fichiers codent ce temps en un entier relatif de 32 bits (le signe permet de dater les fichiers antérieurs à 1970), sans compter l'affichage de la date des ordinateurs par ce système.

  Plus d'informations ici sur ces deux bugs : https://www.presse-citron.net/le-nouveau-bug-de-lan-2000-est-prevu-en-2038-explications/

IV/ Représentation des entiers relatifs en machine

1/ Une première idée ...

La première idée serait d'utiliser le bit de poids fort pour le signe (0 pour un nombre positif et 1 pour un nombre négatif) et les autres pour la *valeur absolue* (*).

(*) la valeur absolue d'un nombre, c'est le maximum entre lui et son opposé.
Par exemple, la valeur absolue de 4 est 4 car 4 > -4. La valeur absolue de -5 est 5 car 5 > -5.

Malheureusement, c'est une mauvaise méthode notamment pour effectuer des additions.

Exemple : on souhaite additionner 4 et -2 sur quatre bits.
Avec ce système, 4 s'écrirait 0100_2 et -2 s'écrirait 1010_2.
Effectuons maintenant la somme de 4 et de -2 :

	0	1	0	0	quatre
+	1	0	1	0	moins deux
	1	1	1	0	moins six

On voit qu'avec ce système, 4 + (-2) = -6 ce qui n'est pas acceptable.

2/ Le complémentaire à 2^n

Considérons l'addition de 15 et de 1 sur quatre bits.
On a alors :
$15 = 1111_2$
$1 = 0001_2$

	1	1	1	1	quinze
+	0	0	0	1	un
	1	1	1		retenues
1	0	0	0	0	seize

On obtient seize mais dans ce système à 4 bits, le bit à gauche est « perdu », on obtient alors zéro.

Ecrit en ligne, nous avons donc : 1111 + 0001 = 0000. On peut donc dire que **0001 est l'opposé de 1111** dans ce système à quatre bits. **1111** est alors la représentation de **-1**.

Voici un autre exemple où l'on cherche l'opposé de cinq sur 4 bits.
On a alors :
$5 = 0101_2$

	0	1	0	1	cinq
+	1	0	1	0	dix
+	0	0	0	1	un
	1	1	1		retenues
~~1~~	0	0	0	0	~~seize~~ zéro

Pour trouver l'opposé de 0101, on a :
- inversé les bits de cinq (voir l'exemple, cela donne 10 ici),
- ajouté 1.

Dans cet exemple, 11 est appelé **complément à 2^4** de 5 car leur somme vaut 2^4.

A savoir : De manière avec *n* bits**, le complément à 2^n d'un nombre r est $2^n - r$**.

Résumé à savoir :
- **avec *n* bits, on peut représenter les nombres compris entre -2^{n-1} et $2^{n-1} - 1$,**
- **les nombres négatifs ont tous un bit de poids fort égal à 1,**
- **le nombre -1 est représenté par une suite de 1 quel que soit le nombre de bits.**

Remarque (pour les matheux) : on retrouve ce principe sur le cercle trigonométrique graduée en degrés.

3/ Détermination de l'opposé d'un entier positif

Méthode à connaître : **Pour trouver la représentation en machine de l'opposé d'un nombre positif y, il faut :**
- **inverser les bits du nombre y,**
- **ajouter 1.**

Exemple : On souhaite trouver l'opposé de 9 sur 8 bits (1 octet).
$9 = 00001001_2$

a) Prenons l'inverse de chacun de ses bits,

00001001 => 11110110

b) Ajoutons 1,

11110110 = > 11110111

Vérification : Convertissons 11110111_2 en base dix.
Il vient : 11110111_2 => $1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6 + 1 \times 2^7 = 247$
On a bien $9 + 247 = 256$ soit $9 + 247 = 2^8$.
11110111 est bien la représentation en machine de l'opposé de 00001001 dans un système à un octet.

4/ Détermination d'un entier négatif

Méthode à connaître : **Pour trouver la représentation en machine d'un nombre négatif y, il faut :**

- **soustraire 1 à y,**
- **inverser les bits du nombre obtenu**

Exemple : On souhaite à quel entier relatif correspond 10110101_2.
On sait que c'est un nombre négatif car son bit de poids fort est 1.

a) Enlevons 1 à ce nombre,
   10110101 => 10110100
b) Prenons l'inverse de chacun des bits du nombre obtenu,
   10110100 = > 01001011
c) En base 10, on a 01001011_2 => $1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 = 75$.

10110101 correspond à l'écriture de -75 dans un système à un octet.

5/ Langages de programmation et nombres entiers

Dans certains langages, l'ensemble des nombres entiers utilisé (naturel ou relatif) est précisé grâce au **type de variables** utilisé grâce au mot clé « unsigned » qui signifie « sans signe ».

Rappelons qu'en **algorithmique**, il faut impérativement préciser le **type** des **variables**.

C'est le cas des langages **C** et **C++** (les meilleurs :)) qui possèdent un **typage statique** : on ne peut pas passer (en théorie du moins, surtout pour le C) d'un type à un autre de manière implicite.

A noter que ce typage indique aussi la quantité de mémoire allouée par variable reste fixe.

Le langage **Python** se charge lui-même de donner le type adapté à la variable en fonction des affectations : il s'agit d'un **typage dynamique**. La quantité de mémoire qui lui est allouée varie donc.