Chapitre 1

Architecture de l'ordinateur Représentation des nombres

1.1 Architecture de l'ordinateur

Un ordinateur est un appareil électronique qui fonctionne à partir d'une alimentation électrique. Ce peut être ce peut être le secteur ou une batterie.

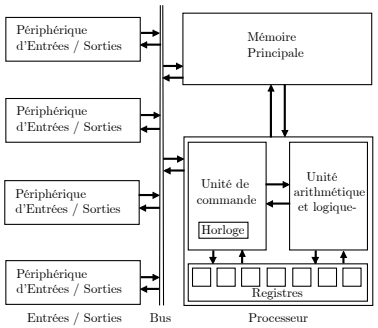
Avec un composant électronique il est facile de représenter deux états : chargé/déchargé, circuit ouvert/fermé, qu'on peut assimiler à deux valeurs, 0 et 1. Aussi dans un ordinateur, toutes donnée, quelle qu'elle soit est représentée par une suite de 0 et de 1 qu'on appelle des bits (en anglais, contraction de BInary digiTS, ou chiffres binaires).

Dans un ordinateur toutes les données sont stockées sous forme d'une suite *bits*. Chaque bit peut prendre 2 valeurs, 0 ou 1.

1.1.1 Architecture de Von Neumann

Tous les ordinateurs depuis 1948, sont modelés sur le même principe de fonctionnement, qui bien qu'il ait été complexifié depuis, repose sur le schéma proposé par John Von Neumann : l'architecture de Von Neumann :

Tous les ordinateurs sont conçus sur la même architecture, dite *architecture de Von Neumann*, qui a été proposée par le mathématicien John Von Neumann en 1948.



Dans cette architecture l'ordinateur est composé du processeur, de la mémoire principale, des périphériques d'entrées/sorties, et du bus qui permet leur communication. Par analogie avec un être humain, l'alimentation électrique forme les organes vitaux (coeur, poumons, estomac, ...) le processeur et la mémoire principale forment le cerveau, (capacité d'action, de calcul, donneur d'ordre pour le processeur, et mémoire pour la MP) les périphériques d'entrée sortie forment les membres (mains, pieds, ...) les sens (vue, ouïe, toucher, goût, odorat) et les cordes vocales, et le bus forme le système nerveux permettant la communication du cerveau avec tous les organes.

Dans l'architecture de Von Neumann :

- La **mémoire principale** contient les données et les programmes. Le processeur y accède directement pour l'exécution de tous les programmes.
- Le **processeur** (ou micro-processeur) contient :
  - une **horloge**, qui cadence toutes les opérations,
  - l'**unité de commande** qui séquence l'exécution des instructions contenues dans les programmes en mémoire,
  - l'**unité arithmétique et logique** qui effectue tous les calculs et les opérations logiques, et
  - les **registres**, unités de mémoires d'accès ultra rapide.

Le processeur communique avec les autres composants par le biais du bus.

- La **mémoire principale**, contient toutes les données et les programmes au cours de leur exécution. Elles est constituée de :
  - la mémoire dite "morte", perpétuelle, qui ne peut être que lue, et qui contient les données et programmes nécessaires au démarrage de l'ordinateur. C'est la **ROM** (Read-Only Memory).
  - La **mémoire vive**, ou **RAM** (Random Access Memoy), volatile, qui se charge à l'exécution des programmes et au chargement de données.

Le processeur a un accès direct à la mémoire principale. On parle de mémoire à accès direct.

• Le **bus**, permet d'échanger les données entre les différentes unités. Il est constitué :

- d'un **bus d'adresse**, qui spécifie ou se situe la donnée.
- d'un **bus de donnée**, qui permet le transfert, lecture, ou écriture de la donnée à son adresse.

• L'**unité d'entrée / sortie** permet la connection des différents **périphériques d'entrées / sorties**. Chaque périphérique d'entrée/sortie est connecté sur un port. L'unité d'entrée/sortie fait le pont entre le bus et les différents périphériques.

1.1.2 La mémoire

Toutes les données en mémoire sont stockés sous forme de suites de 0 et de 1, des *bits*. Afin d'optimiser les temps d'accès, l'ordinateur ne peut pas accéder à des bits isolés, mais seulement à des regroupement de plusieurs bits, traditionnellement 8 bits. On les appelle des *octets*.

Un *octet* est un groupement contigü de 8 bits. On anglais un octet se dit *byte*.

Au fur et à mesure de l'augmentation des performances des ordinateurs, l'unité de stockage a évolué de 8 bits, à 16 bits (années 80), 32 bits (années 90), 64 bits (années 2010). C'est l'unité de stockage. Cela signifie que dans votre ordinateur 64bits, tous les composants de l'ordinateur, ne stockent et ne transfèrent que des groupement de 64 bits, soit 8 octets. Par exemple le bus d'adresse sera constitué de 64 fils électriques. Cela permet de représenter $2^{64} \approx 18 \times 10^{18}$ adresses, soit près de 20 milliards de milliards.

Dans un ordinateur 64 bits, qui est la norme actuelle des ordinateurs personnels, les données sont regroupées par unités de stockage de 64 bits, soit 8 octets.

Toutes les données seront stockées dans différents composants de mémoire. Elles peuvent être lues, et écrites, par le processeur, mais aussi parfois par des périphériques d'entrée/sortie.

- Les **registres**, situés au sein du microprocesseur. Volatiles (leur contenu est perdu à l'arrêt de l'ordinateur), ils peuvent être accédés en lecture et en écriture. En nombre restreint (de plusieurs dizaines à plusieurs centaines d'unités de stockage), ils ne sont utilisés pour les stockages temporaires durant les calculs et opérations du processeur. Leur temps d'accès est le plus rapide, de l'ordre d'une nano-seconde (1 milliardième de seconde) par unité de stockage.

  Lorsqu'on effectue un addition, le processeur stocke les deux opérandes dans deux registres, effectue le calcul au sein de l'unité arithmético-logique, qui stocke le résultat dans une registre privilégié, l'accumulateur. Le processeur peut ensuite communiquer ce résultat sur le périphérique de sortie souhaité, ou l'enregistrer dans la mémoire principale, selon les instructions de l'utilisateur, ou plutôt du programme utilisé.
- La **mémoire vive** (RAM) située dans la mémoire principale. Volatile, accessible en lecture et en écriture, c'est le lieu d'exécution de tous les programmes. Elle contiendra à la fois les programmes, lors de leur exécution, et les données qu'ils manipuleront. Le processeur y accède directement. Le temps d'accès y est très rapide, de l'ordre d'une dizaine de nano-secondes par unité de stockage. Sa quantité est en général limitée à quelques giga-octets (milliards d'octets).

  À l'exécution d'un programme, ce dernier se charge d'abord dans la mémoire vive sous forme d'instructions en langage machine, uniquement comprises par l'unité de commande. C'est l'unité de commande au sein du processeur qui séquence son exécution, en exécutant la suite de ses instructions. Les données qu'il manipule seront aussi stockées dans la mémoire vive. A l'arrêt de l'ordinateur toutes les données de la mémoire vive seront perdues.
- La **mémoire morte** (ROM) située dans la mémoire principale et d'accès direct rapide. Perpétuelle (non volatile), elle n'est accessible qu'en lecture. Elle sert à stocker le programme qui se charge au démarrage de l'ordinateur (le BIOS sous PC), et qui sert à vérifier la connexion des périphériques, à charger leurs pilotes, et à charger dans la RAM le programme d'exécution du système d'exploitation (shell).

  De nos jours il en fait possible d'écrire dans la ROM, mais seulement pour des mises à jours réservées aux spécialistes et au système d'exploitation. C'est la ROM EPROM.
- Le **disque dur**. C'est un périphérique d'entrée et sortie destiné au stockage à long terme du système d'exploitation et des programmes et données. Non volatile, son contenu demeure à l'extinction de l'ordinateur, il est accessible en lecture et en écriture. Le transfert des données est lent, de l'ordre de quelques milli-secondes par unité de stockage (1000 fois plus lent que la mémoire principale).

  Au démarrage, et à l'ouverture des programme, l'ordinateur charge les données utilisées du disque dur vers la RAM, pour l'exécution ou le traitement des données. On distingue de nos jours essentiellement

deux types de disque dur, HDD fragile et lent équipant la plupart des ordinateurs, SSD robustes et plus rapides, mais aussi plus chers, ils occupent les ordinateurs plus hauts de gamme.

- Les **périphériques de stockage externes**. Mémoires non-volatiles ; on peut citer les clés USB, cartes mémoires, disques durs externes, stockage sur réseau (cloud), disques CD, DVD, Blue-ray gravables, accessibles en lecture et écriture, ou les CD, DVD, Blueray non gravables, accessibles uniquement en lecture.

1.1.3 Le processeur

C'est le coeur de l'ordinateur. Il est constitué de :

- L'**horloge**. Elle détermine la rapidité du processeur, et par voie de conséquence de l'ordinateur. Sa rapidité est donné en Hz, MHz, GHz. Ainsi un processeur cadencé à 3 GHz effectue 3 milliards d'opérations à la seconde. Toutes les opérations de l'ordinateur sont cadencés par l'horloge.
- L'**unité de commande** ; elle exécute les instructions des programmes. Ces instructions sont lues séquentiellement dans la mémoire principale ; elles sont codées en langage machine. Les instructions du langage machine dépendent du microprocesseur et sont en nombre très restreints : elles consistent en quelques opérations élémentaires ; lire le contenu d'une unité de stockage située à une certaine adresse (dans la mémoire principale ou via le bus), écrire une donnée à une certaine adresse (idem), commander un calcul ou une opération logique à l'unité arithmétique et logique, passer à l'instruction suivante en mémoire, ou sauter à une instruction situées à une adresse donnée en mémoire principale à condition que la valeur d'un registre donné soit non-nul. Une opération est exécutée à chaque temps d'horloge.

  Tout ce qui s'exécute dans l'ordinateur, est commandé par l'unité de commande. C'est le donneur d'ordre. Tout programme, aussi complexe soit-il, s'obtient par l'exécution de ces quelques opérations élémentaires.
- L'**unité arithmétique et logique**. Commandée par l'unité de commande, son rôle est d'exécuter tous les calculs élémentaires, addition, soustraction, multiplication, division, et surtout toutes les opérations logiques, et, ou, non, xor, décalage, dont toutes les opérations mathématiques sont déduites. Les opérandes et résultats sont stockés dans les registres. L'unité arithmétique et logique dispose aussi d'un accès direct à la mémoire principale, et d'une connexion avec le bus.

Un ordinateur "multi-core" dispose de plusieurs processeurs qui peuvent travailler simultanément.

1.1.4 Périphériques d'entrées-sorties

Ils sont indispensables pour communiquer avec l'ordinateur. Ils étaient initialement réduits à des entrées et sorties sous forme de cartes perforées. Les périphériques d'entrée-sorties peuvent être, par exemple :

- Périphériques d'entrée : clavier, souris, trackpad, joystick, scanner, webcam, écran tactile...
- Périphériques de sortie : écran, vidéo-projecteur, haut-parleurs, imprimante, bras articulé, etc...
- Périphériques de stockage ; ce sont des périphériques d'entrées et sorties : disques externes, lecteur/graveur CD, DVD, Blueray, clé usb, etc...
- Périphériques de communication ; ce sont des périphériques d'entrées et de sorties : carte éthernet, carte wifi, bluetooth, clés 3G, etc...

1.2 Langage de programmation

1.2.1 Le langage machine

Un processeur ne comprend qu'un seul langage, le langage machine. C'est un langage séquentiel (les instructions sont exécutées les unes à la suite des autres) constitué d'une jeu d'instruction très réduit permettant d'accéder, lire et modifier les données stockées dans la mémoire principale ou via le bus dans les périphériques, d'effectuer des calculs et opération logiques basiques dans l'unité arithmétique et logique, et de poursuivre l'exécution du programme en sautant à une instruction située à une certaines adresses. C'est un langage de bas niveau, illisible pour le néophyte, et dont le jeu d'instruction dépend du processeur utilisé. Voilà à quoi peuvent ressembler quelques instructions (3 ici) en langage machine sur un ordinateur 8 bits :

Adresse	Contenu (binaire)	Contenu (hexadecimal)
00000000	01010101	55
00000001	11111111	FF
00000010	00000000	00
00000011	11001000	C8

```
00000100 |       00110001          |       31
```

Y figurent, l'adresse en mémoire, l'instruction sur un ou plusieurs octets. On a écrit le contenu en binaire, mais aussi en hexadécimal (base 16, très utilisée en informatique) pour faciliter sa lecture.

1.2.2 L'assembleur

L'assembleur a été le premier langage informatique lisible par un humain. Ce n'est qu'une retranscription du langage machine utilisant certaines commandes pour faciliter sa lecture et son écriture : `LD` (pour LoaD, charger un registre avec une donnée), `MOV` (pour MOVe, déplacer une donnée entre deux adresses ou registres), `ADD` (pour ajouter au contenu d'un registre le contenu situé à une adresse), etc... La conversion language machine/assembleur est très simple, c'est une traduction mot à mot, et devra être exécuté par un programme appelé assembleur. Voici un exemple d'instructions (3 ici) en assembleur et leur conversion en langage machine sur un ordinateur 16 bits ; les adresses sont données en hexadecimal :

```
Adresse | Language Machine |    Assembleur

0000    |       A1 0A FF   |    MOV AX, [0AFF]
0003    |    03 06 0B 02   |    ADD AX, [0B02]
0007    |       A3 0B 04   |    MOV [0B04], AX
```

La première instruction charge le registre `AX` avec la donnée à l'adresse `0AFF`.
La seconde ajoute à `AX` le contenu de `0B02`.
La troisième range le résultat à l'adresse `0B04`.

L'assembleur est un langage de bas niveau : son jeu d'instruction ne consiste qu'en des opérations élémentaires du microprocesseur.

1.2.3 Les langages évolués

Pour faciliter l'écriture de programmes, des langages de plus haut niveau ont été conçus : citons Fortran, Cobol, Langage C, Basic, Pascal, etc... ils permettent l'écriture de programme avec un jeu d'instructions bien plus étendu, permettant par exemple l'écriture de texte à l'écran, le tracé de graphiques, l'ouverture/ lecture d'un fichier sur le disque, ou des instructions permettant de répéter des jeux d'instructions (boucles). Ce sont en général des langages dits structurés, par opposition aux langages séquentiels que constituent le langage machine et l'assembleur.

L'écriture d'un programme dans un langage évolué est considérablement plus simple (car l'utilisateur n'a pas à connaître les adresses des différents périphériques), et concis (car le jeu d'instruction est considérablement plus large et réalise des opérations plus complexes). De nos jours, à part pour la programmation de pilotes de périphériques ou des systèmes d'exploitation, la programmation ne se fait plus que dans des langages évolués.

Un programme écrit dans un langage évolué doit être ensuite compilé par un programme appelé compilateur qui le traduit en langage machine. C'est ce code traduit qui constitue le programme exécutable qui sera séquencé par l'unité de commande (les choses sont en fait un rien plus complexes).

Depuis l'arrivée de l'internet, sont apparus des langages qui ne sont plus compilés, mais interprétés. Dans ce cas les instructions du programme sont traduites en langage machine et exécutés simultanément. L'exécution des programmes en est plus lente car l'interpréteur doit gérer la mémoire au fur et à mesure de l'exécution, et ne peut pas optimiser le code, mais cela permet d'exécuter un programme sous forme de scripts situés sur des sites web distants. Citons notamment PHP, Java script, et `python` qui est le langage de programmation évolué que nous utiliserons.

Le programme officiel en classes préparatoires requiert que la programmation soit faites dans le langage `python`.

1.2.4 Installation de `python`

On installera une distribution `python`, avec un EDI (Environnement de développement integré). Plusieurs versions du langage sont disponibles, il faudra prendre soin de télécharger la dernière version `python` 3 (à ce jour la dernière version est la 3.5.1 ; n'importe quelle version 3.x.x fera l'affaire). Attention à ne pas télécharger une version antérieure, `python` 2, dont le jeu d'instructions varie significativement.

Plusieurs possibilités sont offertes, toutes gratuites, et incluant le langage `python` sous sa dernière version :

- La distribution officielle, disponible en téléchargement à l'adresse :

  `https://www.python.org`

  Elle propose comme EDI, le logiciel Idle, un peu rudimentaire. C'est pourquoi nous la déconseillons, ou ne la conseillons qu'à l'utilisateur néophyte qui souhaiterait conserver son propre EDI (emacs, VIM, ou autres)
- La suite Anaconda, très complète, développé par le Massachusset Institute of Technology (MIT), et orientée pour le calcul scientifique, est disponible en téléchargement à l'adresse :

  `https://www.continuum.io/why-anaconda`

Elle utilise l'EDI Spyder, de très bonne facture. Cependant il s'avère qu'elle n'est pas exempte de bugs, et importe les modules scientifiques sans qu'on le lui demande, ce qui peut parfois réserver des surprises. C'est pourquoi nous ne la conseillons pas outre mesure.

- La distribution Pyzo : Python to the people. Simple, fonctionnelle, portable (peut être située sur une clé usb et utilisé sans installation préalable), elle est disponible à l'adresse :

  `http://www.pyzo.org`

  Elle utilise pour E.D.I. le logiciel IEP.
  Nous encourageons son usage.

1.3 Principe de la représentation des nombres

1.3.1 Représentation binaire d'un entier naturel

Principe mathématique de la représentation d'un entier naturel dans une base

Nous avons coutume de représenter les nombres entiers par leur écriture décimale, ou écriture en base 10, qui utilise les 10 chiffres de 0 à 9. Ce principe de représentation d'un nombre entier dans une base se généralise à 2 chiffres, 3 chiffres, etc..., ou à tout nombre arbitraire > 1 de chiffres. Cette représentation est basée sur les propriétés mathématiques que nous développons maintenant.

Tout d'abord nous rappelons le principe fondamental en arithmétique de la division euclidienne ; nous en profitons pour en donner une preuve.

Théorème 1 (Division euclidienne.) *Pour tout entiers $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$, il existe une unique couple d'entiers $(q,r) \in \mathbb{N}^2$ tels que :* $\begin{cases} a = q \times b + r \\ 0 \leqslant r < b \end{cases}$. *$q$ s'appelle le quotient et r le reste, dans la division euclidienne de a par b.*

Preuve.

Montrons d'abord l'existence par récurrence forte. Fixons $b \in \mathbb{N}^*$ et considérons la proposition de récurrence : $\mathscr{P}(n)$: "pour tout entier naturel $a < n$, il existe un couple d'entiers (q,r) tels que $a = q \times b + r$ et $0 \leqslant r < b$".

Initialisation. Lorsque $n = 1$, $a = 0$ et il suffit de prendre $q = 0$ et $r = 0 < b$. Ainsi $\mathscr{P}(1)$ est vérifiée.

Hérédité. Supposons $\mathscr{P}(n)$ vraie. Soit $a < n+1$; c'est à dire $a < n$ dans quel cas il existe $(q,r) \in \mathbb{N}^2$ tels que $a = q \times b + r$ et $0 \leqslant r < b$ (HR), ou bien $a = n$. Supposons dans la suite que $a = n$.

Premier cas : $a < b$. Dans ce cas $a = 0 \times b + a$ et le couple $(0, a)$ convient.

Deuxième cas : $a \geqslant b$: par hypothèse de récurrence $a - b = q \times b + r$ avec $0 \leqslant r < b$. Donc $a = (q+1) \times b + r$. Le couple $(q+1, r)$ convient.

Ainsi $\mathscr{P}(n+1)$ est vraie. Ceci conclut la récurrence.

Montrons maintenant l'unicité. Supposons que $a = q_1 \times b + r_1 = q_2 \times b + r_2$ avec $0 \leqslant r_1, r_2 < b$. En soustrayant terme à terme les deux égalités on obtient : $(q_2 - q_1) \times b = r_1 - r_2$. Donc b divise $r_1 - r_2$; mais $-b < r_1 - r_2 < b$ puisque $0 \leqslant r_1, r_2 < b$. Ainsi nécessairement $r_1 - r_2 = 0$, et puisque $b \neq 0$, $q_1 - q_2 = 0$. Ainsi $q_1 = q_2$ et $r_1 = r_2$. Ceci prouve l'unicité. □

On déduit de ce résultat, le principe de la représentation d'un entier naturel à l'aide d'une base, qui peut être n'importe quel entier > 1.

Théorème 2 *Pour tout entier $m \in \mathbb{N} \setminus \{0,1\}$, pour tout entier $n \in \mathbb{N}$, il existe une unique suite finie $(u_n)_{n \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que :*

$$n = \sum_{i=0}^{N} u_i \times m^i$$

avec si $N > 0$, $u_N \neq 0$.

On peut alors définir ce qu'est l'écriture d'une entier naturel dans une base :

Définition. L'écriture en base m de l'entier n est :

$$u_N \, u_{N-1} \cdots u_1 \, u_0$$

pour la suite finie $(u_n)_{n \in [[0,N]]}$ donnée par le théorème 2.

Preuve du théorème 2. Existence : par récurrence forte. Fixons $m \in \mathbb{N} \setminus \{0,1\}$, et considérons pour proposition de récurrence :

$\mathscr{P}(n)$: "pour tout entier naturel $a < n$, il existe une suite finie $(u_n)_{n \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que $a = \sum_{i=0}^{N} u_i \times m^i$ avec si $N > 0$, $u_N \neq 0$".

Initialisation. La proposition $\mathscr{P}(1)$ est vraie. Soit $0 \geqslant a < 1$, c'est à dire $a = 0$. Alors la suite $u_0 = 0$ convient.

Hérédité. Supposons la proposition $\mathscr{P}(n)$ vraie. Soit $a < n+1$, on peut supposer que $a = n$ car autrement on a la suite requise par hypothèse de récurrence. Par division euclidienne de n par m, il existe un unique couple $(q,r) \in \mathbb{N}^2$ tel que $n = q \times m + r$ avec $0 \leqslant r < m$. Nécessairement $0 \leqslant q < n$ car autrement on aurait $n = q \times m + r > n$ puisque $m > 1$ et $r \geqslant 0$. Donc par hypothèse de récurrence il existe une suite finie $(v_n)_{n \in [[0,N]]}$ d'entiers compris entre 0 et $m-1$, tels que $q = \sum_{i=0}^{N} v_i \times m^i$ avec si $N > 0$, $v_N \neq 0$".

Ainsi :

- Si $q > 0$:

$$n = q \times m + r = \left(\sum_{i=0}^{N} v_i \times m^i \right) \times m + r = \sum_{i=0}^{N} v_i \times m^{i+1} + r = \sum_{i=0}^{N'} u_i \times m^i$$

  en posant $N' = N + 1$ et pour $1 \leqslant i \leqslant N+1$, $u_i = v_{i-1}$ et $u_0 = r$.
- Si $q = 0$: $n = r = u_0$.

Puisque $0 \leqslant r < m$, la suite $u_0, u_1, \ldots u_{N+1}$ est une suite d'entiers compris entre 0 et $m-1$. De plus si $N' > 0$, $u_{N'} = v_N \neq 0$.
Ainsi $\mathscr{P}(n+1)$ est vraie.

Montrons l'unicité. Supposons que :

$$n = \sum_{i=0}^{N_1} u_i \times m^i = \sum_{i=0}^{N_2} v_i \times m^i$$

avec $u_{N_1} \neq 0$ si $N_1 > 0$ et $u_{N_2} \neq 0$ si $N_2 > 0$.
Premier cas : si $n = 0$ alors $N_1 = N_2 = 0$ et $u_0 = v_0 = 0$.
Deuxième cas : si $n > 0$. Alors u_{N_1} et u_{N_2} sont non-nuls. En effectuant la division euclidienne de n par m, on obtient :

$$n = q \times m + r \qquad \text{avec} \qquad r = u_0 = v_0 \quad \text{et} \quad q = \sum_{i=0}^{N_1 - 1} u_{i+1} \times m^i = \sum_{i=0}^{N_2 - 1} v_{i+1} \times m^i$$

En appliquant le même procédé avec q à la place de n, on obtient $u_1 = v_1$, et en poursuivant ainsi de suite avec chaque quotient obtenu, $N_1 = N_2$ et $u_2 = v_2,\ u_3 = v_3, \ldots,\ u_{N_1} = v_{N_2}$. Ceci prouve l'unicité. □

Il apparaît implicitement dans la preuve (démonstration de l'unicité) un algorithme permettant d'obtenir la suite de chiffres de l'écriture d'un entier naturel arbitraire dans une base.

Théorème 3 (Algorithme d'écriture d'un entier dans une base)
Donné $m \in \mathbb{N} \setminus \{0,1\}$, donné une entier $n \in \mathbb{N}^*$, l'écriture $u_N\ u_{N_1} \cdots u_1\ u_0$ de l'entier n dans la base m s'obtient grâce à l'algorithme suivant :

- *Soit $k = 0$*
- *TANT QUE $n \neq 0$ FAIRE :*
  - *Soit u_k le reste dans la division euclidienne de n par m*
  - *Changer n par le quotient dans la division euclidienne de n par m*
  - *Changer k par $k+1$*
- *FIN TANT QUE*
- *$u_{k-1}\ u_{k-2} \cdots u_2\ u_1\ u_0$ est l'écriture de n dans la base m.*

lorsque $n = 0$, son écriture dans la base m est simplement 0.

- Exemple : Ecriture de 123 en base 10 :

$$\begin{array}{ll} 123 = 12 \times 12 + 3 & u_0 = 3 \\ 12 = 1 \times 10 + 2 & u_1 = 2 \\ 1 = 0 \times 10 + 1 & u_2 = 1 \end{array}$$

Sans surprise on obtient comme écriture de 123 en base 10 (ou décimal) : 123.

- Exemple : Ecriture de 72 en base 2 :

$$\begin{array}{ll} 72 = 2 \times 36 + 0 & u_0 = 0 \\ 36 = 2 \times 18 + 0 & u_1 = 0 \\ 18 = 2 \times 9 + 0 & u_2 = 0 \\ 9 = 2 \times 4 + 1 & u_3 = 1 \\ 4 = 2 \times 2 + 0 & u_4 = 0 \\ 2 = 2 \times 1 + 0 & u_5 = 0 \\ 1 = 2 \times 0 + 1 & u_6 = 1 \end{array}$$

72 s'écrit en base 2 (ou binaire) : $\overline{1001000}$.

Methodologie. Pour obtenir la représentation de nombre assez petits (jusqu'à quelques dizaines, centaines ou milliers) on peut aussi leur retrancher tant que possible les plus grandes puissances de 2.
Exemple : avec 72 : $72 - 64 = 8$; $8 - 8 = 0 \implies 72 = 2^6 + 2^3 = \overline{1001000}$.

Mais pour cela il faut connaitre le plus grand nombre possible de premiers termes de la suite des puissance de 2. On tentera de se rappeler de :
$2^0 = 1$; $2^1 = 2$; $2^2 = 4$; $2^3 = 8$; $2^4 = 16$; $2^5 = 32$; $2^6 = 64$; $2^7 = 128$; $2^8 = 256$; $2^9 = 512$; $2^{10} = 1024$

Remarque. Il ne faut pas confondre un nombre, c'est la quantité représentée, et son écriture dans une base.
Par exemple le nombre 10, s'écrit :

- $10 = 1 \times 10^1 + 0 \times 10^0$ 10 en décimal.
- $10 = 1 \times 9^1 + 1 \times 9^0$ 11 en base 9.
- $10 = 1 \times 8^1 + 2 \times 8^0$ 12 en base 8.
- $10 = 1 \times 7^1 + 3 \times 7^0$ 13 en base 7.
- $10 = 1 \times 6^1 + 4 \times 6^0$ 14 en base 6.
- $10 = 2 \times 5^1 + 0 \times 5^0$ 20 en base 5.
- $10 = 2 \times 4^1 + 2 \times 4^0$ 22 en base 4.
- $10 = 1 \times 3^2 + 0 \times 3^1 + 1 \times 3^0$ 101 en base 3.
- $10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ 1010 en binaire.

Dans tout base > 10 il faudra des chiffres additionnels pour les chiffres $\geqslant 10$, par exemple des lettres de l'alphabet : A, B, etc...

- $10 = 10 \times 11^0$ s'écrit A en base 11.
- $10 = 10 \times 12^0$ s'écrit A en base 12.

etc...
La confusion vient de ce que nous notons habituellement les nombres en base 10. Mais ce n'est qu'une représentation : le nombre dix ce n'est pas '1' suivi de '0', c'est plutôt le nombre des doigts de la main, le nombre des orteils des pieds, deux multiplié par cinq, neuf augmenté de un, etc... Lorsqu'il y ambiguité il faut préciser dans quelle base la représentation est donnée.

Pour analogie avec l'écriture : les chiffres sont l'analogue des lettres de l'alphabet, les bases sont l'analogue des langues, et les nombres l'analogue des signification des mots dans la langue.

Représentation binaire des entiers naturels

Définition. On appelle *écriture* (ou *représentation*) *binaire* d'un entier naturel, son écriture en base 2.

D'après le théorème 3, l'algorithme suivant permet à partir d'un entier $n \in \mathbb{N}^*$ d'obtenir son écriture en binaire :

Algorithme d'écriture binaire d'un entier naturel.

- Soit $k = 0$
- TANT QUE $n \neq 0$ FAIRE :
  - Soit u_k le reste dans la division euclidienne de n par 2
  - Changer n par le quotient dans la division euclidienne de n par 2
  - Changer k par $k+1$
- FIN TANT QUE
- $u_{k-1}\ u_{k-2}\ \cdots u_2\ u_1\ u_0$ est l'écriture de n dans la base 2.

lorsque $n = 0$, son écriture dans la base 2 est simplement 0.

Par exemple on a vu que l'écriiture binaire du nombre 72 est : 1001000. Pour lever tout ambiguité sur la base utilisée on pourra utiliser plutôt la notation :

$$72_{10} = 1001000_2 \quad \text{ou} \quad \overline{1001000}$$

Définition. Les chiffres $u_N\ u_{N-1}\ \cdots u_1\ u_0$ dans l'écriture *binaire* d'un nombre sont appelés des *bits*. Le chiffre u_k est appelé le *bit de poids k*.

Remarque. Un nombre entier représenté en binaire est pair si et seulement si son bit de poids faible vaut 0, et impair sinon.

Addition binaire des entiers.
L'algorithme d'addition de deux nombres écrits en binaire est le même que celui de nombres écrits en décimaux. On a besoin pour cela de la table d'addition :

$$\begin{aligned} 0+0 &= 0\\ 0+1 &= 1\\ 1+0 &= 1\\ 1+1 &= 10 \end{aligned}$$

La dernière entrée de la table, donne pour règle dans l'algorithme d'addition $1+1$: 0 avec retenue de 1. Pour le reste tout se passe comme pour l'addition de décimaux.

Exemple : Addition : $23 + 7 = 30$.
$23 = \overline{10111}$ et $7 = \overline{111}$. On complète les écritures avec des zéros pour qu'elles aient même longueur et on pose l'addition :

$$\begin{array}{rr} & 23\\ + & 7\\ \hline = & ? \end{array} \qquad \begin{array}{rccccc} & 1 & 0 & 1 & 1 & 1\\ + & 0 & 0 & 1 & 1 & 1\\ \hline \end{array}$$

$$\begin{array}{rr} & \\ & 23\\ + & 7\\ \hline = & 30 \end{array} \qquad \begin{array}{rccccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & 1 & 0 & 1 & 1 & 1\\ + & 0 & 0 & 1 & 1 & 1\\ \hline & 1 & 1 & 1 & 1 & 0 \end{array}$$

Résultat : $\overline{11110} = 2^4 + 2^3 + 2^2 + 2^1 = 16 + 8 + 4 + 2 = 30$.

Multiplication binaire des entiers. La multiplication d'entiers est analogue à la multiplication décimale. On a besoin de la table de multiplication :

$$\begin{aligned} 0\times 0 &= 0\\ 0\times 1 &= 0\\ 1\times 0 &= 0\\ 1\times 1 &= 1 \end{aligned}$$

Exemple : Multiplication de $18 = \overline{10010}$ par $5 = \overline{101}$:

$$\begin{array}{rccccc} & 1 & 0 & 0 & 1 & 0\\ \times & 0 & 0 & 1 & 0 & 1\\ \hline & 1 & 0 & 0 & 1 & 0\\ 1 & 0 & 0 & 1 & 0 & .\ \ .\\ \hline 1 & 0 & 1 & 1 & 0 & 1\ \ 0 \end{array}$$

Résultat : $18 \times 5 = \overline{1011010} = 2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 = 90$.

1.3.2 Principes de la représentation des nombres en mémoire

Stockage en octets

L'unité de mémoire en informatique est le bit (physiquement constitué d'une unité électronique pouvant prendre deux états, par exemple un condensateur, déchargé ou chargé).

En informatique toutes les données, quelles qu'elles soient, sont stockées sous

la forme abstraite d'une suite de zéros et de uns.

Afin d'optimiser l'accès à la mémoire, un ordinateur manipule des unités de stockage plus importantes. Traditionnellement, sur les premiers ordinateurs personnels l'unité de stockage était l'octet, constitué d'un groupement de 8 bits.

Les premiers ordinateurs personnels (transportables) étant '8 bits', c'est à dire que processeur, bus, et mémoires avaient pour unité de stockage l'octet, soit 8 bits.

Un *octet* est un groupement de 8 bits contigus en mémoire. En anglais bit se dit 'bit' et octet se dit 'byte'.

Un *Kilo-octet* (Ko), *Mega-octet* (Mo), *Giga-octet* (Go), *Tera-octet* (To) correspond respectivement à 10^3 octets, 10^6 octets, 10^9 octets, 10^{12} octets.

A mesure de l'augmentation des performances des ordinateurs, l'unité de stockage a été successivement doublée : 16 bits (2 octets), 32 bits (4 octets) pour atteindre aujourdhui 64 bits (8 octets).

Les ordinateurs personnels modernes utilisent comme unité de stockage 8 octets, soit 64 bits.

Définition. L'espace de stockage des données étant fixée à un groupement de un ou plusieurs octets, le premier bit de l'espace de stockage (le plus à gauche) est appelé **bit de poids fort**. Le dernier bit (le plus à droite) est appelé le **bit de poids faible**.

Remarque. Le mésusage demeure chez les professionnels de prendre pour les unités Ko, Mo, Go, les valeurs respectives $2^{10} = 1024$, 2^{20} et 2^{30} octets, alors qu'il faudrait dans ce cas parler de Kibioctets (Kio), Megioctets (Mio) et Gigioctets (Gio) (selon les recommandations du Système International d'Unités). Cette entorse aux normes en vigueur, et l'existence simultanée des deux usages explique en partie pourquoi lorsque l'on achète un disque dur ou une clé usb, on constate au branchement que la quantité de stockage est souvent un peu moindre qu'annoncée...

1.3.3 Représentation des entiers positifs

Pour représenter en mémoire un entier positif, un ordinateur peut utiliser le format "entier non signé" ou "unsigned int" qui consiste à représenter son écriture binaire sur l'unité de stockage.

La représentation des entiers non signés (`uint`) consiste à représenter leur écriture binaire sur l'unité de stockage, en général 64 bits.

• Sur un octet, 8 bits, il est possible de représenter tous les entiers de :
$\overline{00000000} = 0$ à $\overline{11111111} = 2^8 - 1 = 255$.
C'est le type souvent désigné `uint8` (entier non signé représenté sur 8 bits).

• Sur 2 octets, soit 16 bits, on peut représenter tous les entiers de :
$\overline{0000000000000000} = 0$ à $\overline{1111111111111111} = 2^{16} - 1 = 65535$.
C'est le type souvent désigné `uint16` (entier non signé représenté sur 16 bits).

• Sur 4 octets, soit 32 bits, on peut représenter tous les entiers de :
0 à $2^{32} - 1 = 4294967295$.
C'est le type souvent désigné `uint32` (entier non signé représenté sur 32 bits).

• Sur 8 octets, soit 64 bits, on peut représenter tous les entiers de :
0 à $2^{64} - 1 = 18446744073709551615$.
C'est le type souvent désigné `uint64` (entier non signé représenté sur 64 bits).

• Remarquer que :

$$\underbrace{\overline{111\cdots1111}}_{n \text{ fois } 1} = \sum_{k=0}^{n-1} 2^k = \frac{1-2^n}{1-2} = 2^n - 1$$

ou encore que :

$$\underbrace{\overline{111\cdots1111}}_{n \text{ fois } 1} + \overline{1} = \overline{1\,\underbrace{000\cdots0000}_{n \text{ fois } 0}} = 2^n$$

Limitations de la représentation

Lorsque les nombres sont représentés dans un espace alloué de taille fixé, on peut être confronté à des problèmes de dépassement dans les calculs.

Exemple : Dans la représentation sur 8 bits des entiers non signés :

$$\overline{1111\ 1111} + \overline{0000\ 0001} = \overline{1|\ 0000\ 0000}$$

le résultat de l'addition de deux nombres d'un octet peut donner un résultat qui ne peut pas s'écrire sur 8 bits. Si le résultat de l'opération est représenté sur 8 bits, le bit de poids 8 sera perdu, le résultat retourné sera erroné, ce sera 0 !

Il faut faire très attention à la façon dont sont codés les nombres dans le langage utilisé lorsque l'on effectue des calculs complexes. En fait on calcule dans $\mathbb{Z}/2^N\mathbb{Z}$ ($N = 8, 16, 32, 64$). C'est à dire que tous les calculs seront réalisés modulo 2^N.

Représentation normalisée des entiers signés

Les différentes alternatives

Pour représenter un entier signé (c'est à dire un entier relatif) en binaire par une suite de 0 et de 1 de N bits ($N = 8, 16, 32, 64$), le bit de poids fort (le plus à gauche) est utilisé pour coder le signe.
Il y aurait essentiellement 3 alternatives :

• Première alternative : un bit de signe suivi de la représentation binaire de la valeur absolue.

Principe :

- le bit de poids fort (le plus à gauche) code le signe : 0 pour positif, 1 pour négatif.
- les $N-1$ bits les plus faibles sont utilisés pour coder la valeur absolue du nombre, en notation binaire.

Exemple : sur 8 bits, $\overline{0000\,0001}$ représente 1 et $\overline{1000\,0001}$ représente -1.

Cette approche est simple, et permet de représenter sur N bits tous les entiers relatifs allant de $-(2^{N-1}-1)$ à $2^{N-1}-1$, et pour les positifs l'écriture coïncide avec la représentation binaire des entiers positifs.
Mais elle présente plusieurs désavantages :

- Le zéro admet deux représentations $\overline{0000\,0000}$ et $\overline{1000\,0000}$ (exemple sur 8 bits). Or la comparaison à zéro est une opération très souvent réalisée ; elle consisterait alors en 2 tests au niveau du processeur.
- L'algorithme d'addition demeure correct entre entier positifs, mais est incorrect en général.
  Exemples sur 8 bits :
  $2 + 1 = \overline{0000\,0010} + \overline{0000\,0001} = \overline{0000\,0011} = 3.$ Correct.
  $(-2) + (-1) = \overline{1000\,0010} + \overline{1000\,0001} = \overline{1|0000\,0011} = \overline{0000\,0011} = 3.$ Incorrect.
  $(-2) + 1 = \overline{1000\,0010} + \overline{0000\,0001} = \overline{1000\,0011} = -3.$ Incorrect.
- De même l'algorithme de multiplication est incorrect en général.
  Exemples sur 8 bits :
  $2 \times 1 = \overline{0000\,0010} \times \overline{0000\,0001} = \overline{0000\,0010} = 2.$ Correct.
  $(-2) \times 2 = \overline{1000\,0010} \times \overline{0000\,0010} = \overline{0000\,0100} = 4.$ Incorrect.
  $(-2) \times (-3) = \overline{1000\,0010} \times \overline{1000\,0011} = \overline{1000\,0110} = -6.$ Incorrect.

• Deuxième alternative : la translation, représenter sur N bits $n + 2^{N-1} - 1$ écrit en binaire non signé.

Permet de représenter sur N bits tous les entiers relatifs allant de $-(2^{N-1}-1)$ à 2^{N-1}.
Exemple sur 8 bits, pour représenter $n \in [[-127, +128]]$ on représente en binaire n+127. Ainsi : 0 est représenté par 0+127 = $\overline{0111\,1111}$, −127 est représenté par $-127 + 127 = \overline{0000\,0000}$ et 128 est représenté par $128 + 127 = \overline{1111\,1111}$.

1. Le bit de poids fort donne le signe : 1 pour les strictement positifs, 0 pour les négatifs.
2. Il n'y a plus de pertes en espace : sur N bits on peut représenter exactement 2^N nombres.

Cependant :

1. La représentation est entiers positifs ne coïncide pas avec l'écriture binaire des entiers non signés.
2. L'algorithme d'addition est incorrect.
   Exemples sur 8 bits :
   $2 + 1 = \overline{1000\,0001} + \overline{1000\,0000} = \overline{0000\,0001} = -126$ Incorrect
3. L'algorithme de multiplication est incorrect.
   Exemples sur 8 bits :
   $2 \times 1 = \overline{1000\,0001} \times \overline{1000\,0000} = \overline{1000\,0000} = 1$ Incorrect

• Troisième alternative : la notation en complément à deux.

Elle permet de représenter sur N bits tous les entiers allant de -2^{N-1} à $2^{N-1}-1$.

C'est la représentation des entiers relatifs utilisée sur les ordinateurs, car elle ne présente que des avantages :

- Aucune perte en espace : sur N bits on représente exactement 2^N nombres.
- Le bit de poids fort donne le signe : 1 pour les strictement négatifs et 0 pour les positifs.
- Pour les entiers positifs, la représentation coïncide avec la représentation binaire des entiers non signés.
- L'algorithme d'addition demeure correct.
- L'algorithme de multiplication demeure correct.

Nous allons la définir dans le paragraphe suivant. Ce n'est en fait qu'une translation, mais par 2^N et non pas $2^{N-1}-1$ comme précédemment.

La notation en *complément en deux* est la représentation des entiers relatifs la plus largement utilisée par les ordinateurs.

La représentation des entiers signés en complément à deux

Définition. En **complément à deux**, on représente sur N bits tous les entiers allant de -2^{N-1} à $2^{N-1}-1$, selon le principe suivant :

- Un entier positif dans $[[0, 2^{N-1}-1]]$ est représenté par son écriture binaire sur N bits.
- Pour un entier strictement négatif n dans $[[-2^{N-1}, -1]]$:
  - On considère l'écriture binaire de sa valeur absolue $|n|$ sur N bits,
  - On l'inverse (changer 0 en 1 et 1 en 0 : c'est prendre le *complément à 1*),
  - On rajoute 1 au résultat obtenu.

• Exemples sur $N = 8$ bits :

– Pour représenter −9 en complément à deux :
On part de $9 = \overline{0000\,1001}$. On l'inverse $\overline{1111\,0110}$. On ajoute 1 : −9 s'écrit $\overline{1111\,0111}$.

– Pour représenter −1 en complément à deux :
On part de $1 = \overline{0000\,0001}$; son complément à 1 est $= \overline{1111\,1110}$ auquel on ajoute 1 : −1 s'écrit $\overline{1111\,1111}$.

Appelons *complémentation à deux* l'application qui à une suite de 0 et de 1 applique :
– Prendre son complément à un (changer les 1 et 0 et les 0 en 1).
– Puis ajouter 1.

C'est l'opération appliquée à l'écriture binaire de $n > 0$ pour obtenir l'écriture en complément à 2 de $-n$.

Propriété. *La complémentation à deux est une opération involutive sur les écritures en complément à deux des entiers de* $[[-2^{N-1}, 2^{N-1}-1]]$.
Elle envoie l'écriture de 0 sur elle-même, l'écriture de -2^{N-1} *sur elle-même, et pour toutes les autres sur l'écriture de l'opposé du nombre représenté.*

Exemples sur 8 bits :

– La complémentation à deux de $-9 = \overline{1111\,0111}$ donne : $\overline{0000\,1000} + 1 = \overline{0000\,1001} = 9$.

– La complémentation à deux de $-1 = \overline{1111\,1111}$ donne $\overline{0000\,0000} + 1 = \overline{0000\,0001} = 1$.

– La complémentation à deux de $0 = \overline{0000\,0000}$ donne $\overline{1111\,1111} + 1 = \overline{1|0000\,0000} \to \overline{0000\,0000} = 0$.

– La complémentation à deux de $-2^7 = \overline{1000\,0000}$ donne $\overline{0111\,1111} + 1 = \overline{1000\,0000} = -2^7$.

Une méthode permet de calculer "de tête" une complémentation à deux :

Algorithme de complémentation à deux.
A partir de l'écriture du nombre. Par exemple $12 = \overline{0000\,1100}$.

1. Conserver tous les nombres à partir de la droite jusqu'au premier 1 inclus : $\overline{0000\,1\mathbf{100}}$.
2. Inverser tous les autres : $\overline{1111\,0\mathbf{100}}$. C'est l'écriture en complément à 2 de −12.
   (Complément à deux de 12 : $\overline{1111\,0011} + 1 = \overline{1111\,0100}$.)

• Exemple sur 8 bits : L'écriture en compléments à deux sur 8 bits des entiers de $[[-128, +127]]$, représentés par nombre croissant (à gauche), ou par écriture binaire croissante (à droite).

Compléments à 2 sur 8 bits	
-128	$\overline{10000000}$
-127	$\overline{10000001}$
-126	$\overline{10000010}$
⋮	⋮
-3	$\overline{11111101}$
-2	$\overline{11111110}$
-1	$\overline{11111111}$
0	$\overline{00000000}$
1	$\overline{00000001}$
2	$\overline{00000010}$
⋮	⋮
125	$\overline{01111101}$
126	$\overline{01111110}$
127	$\overline{01111111}$

Compléments à 2 sur 8 bits	
$\overline{00000000}$	0
$\overline{00000001}$	1
$\overline{00000010}$	2
⋮	⋮
$\overline{01111101}$	125
$\overline{01111110}$	126
$\overline{01111111}$	127
$\overline{10000000}$	-128
$\overline{10000001}$	-127
$\overline{10000010}$	-126
⋮	⋮
$\overline{11111101}$	-3
$\overline{11111110}$	-2
$\overline{11111111}$	-1

On peut constater qu'en complément à 2, sur 8 bits, on représente l'entier relatif $n \in [[-128; +127]]$ par les 8 bits de poids faibles de $2^8 + n$:

Exemple, pour représenter :
$-128 : 2^8 + (-128) = 256 - 128 = 128 = \overline{10000000}$
$+127 : 2^8 + 127 = 256 + 127 = \overline{1|01111111} \to \overline{01111111}$.

Plus généralement :

Propriété. *La représentation en complément à deux sur N bits consiste à représenter tout entier $n \in [[-2^{N-1}; 2^{N-1}-1]]$ par les N bits les plus faibles de l'écriture binaire (non signée) de $n+2^N$, ou dit différemment à représenter le nombre $(n+2^N)$ modulo 2^N en binaire (non signé).*

Preuve. C'est clair pour $0 \leqslant n \leqslant 2^{N-1}-1$. En effet $n+2^N$ et n ont mêmes N bits de poids les plus faibles.

Pour $-2^{N-1} \leqslant n < 0$: $n = -|n|$.

Le complément à un consiste à représenter en binaire $\overline{1111\cdots1111} - |n| = 2^N - 1 - |n|$,
et le complément à deux représente en binaire $2^N - 1 - |n| + 1 = 2^N - |n| = 2^N + n = \overline{1\,0000\cdots0000} + n$ et n'en garde que les N bits de poids faibles. □

En corollaire, on peut montrer que addition et multiplication fonctionnent avec l'écriture en complément à deux :

Propriété. *L'addition binaire est compatible avec l'écriture en complément en deux. Plus précisément, si $a, b, a+b \in [[-2^{N-1}; 2^{N-1}-1]]$, alors appliquer l'addition binaire aux écritures en complément à deux des nombres a et b produit l'écriture en complément à deux du nombre $a+b$.*

• Exemple :

$(-2) + (-1) = \overline{1111\,1110} + \overline{1111\,1111} = \overline{1|1111\,1101} = \overline{1111\,1101} = -3$ Correct

$(-2) + 1 = \overline{1111\,1110} + \overline{0000\,0001} = \overline{1111\,1111} = -1.$ Correct

Propriété. *La multiplication binaire est compatible avec l'écriture en complément en deux. Plus précisément, si $a, b, a \times b \in [[-2^{N-1}; 2^{N-1}-1]]$, alors appliquer la multiplication binaire aux écritures en complément à deux des nombres a et b produit l'écriture en complément à deux du nombre $a \times b$.*

• Exemple :

	0	0	0	0	0	1	1	0	(6)
×	1	1	1	1	1	0	1	1	(−5)
						1	1	0	
					1	1	0	.	
				0	0	0	.	.	
			1	1	0	.	.	.	
		1	1	0	.	.	.	.	
	1	1	0	.	.	.	.	.	
*	1	0	.	.	.	.	.	.	
*	0	.	.	.	.	.	.	.	
*	1	1	1	0	0	0	1	0	(-30)

Représentation des nombres entiers sur un ordinateur

Dans les langages de programmation les plus courants les nombres entiers sont représentés sous forme d'entiers signés, en complément à deux.

Pour pallier aux problèmes de dépassement dans les calculs, la plupart des langages de programmation proposent 2 formats d'entiers signés ; pour les utiliser il faut déclarer au préalable les variable utilisées dans le type requis.

- Le type entier court, 'int' ou encore 'short int' : les entiers sont codés signés sur N bits, habituellement N = 32 bits ou 64 bits sur des équipements récents. On peut représenter tous les entiers signés sur la plage de valeurs :

  de $-2^{31} = -2\,147\,483\,648$ à $2^{31}-1 = 2\,147\,483\,647$ (N = 32, 4 octets, type `int32`).

  de $-2^{63} = -9\,223\,372\,036\,854\,775\,808$ à $2^{63}-1 = 9\,223\,372\,036\,854\,775\,807$ (N = 64, 8 octets, type `int64`).

- Le type entier long, 'long' ou encore 'long int' : les entiers sont codés signés sur $2N$ bits (N = 32 bits ou 64 bits). On peut représenter tous les entiers signés :

  de $-2^{63} = -9\,223\,372\,036\,854\,775\,808$ à $2^{63}-1 = 9\,223\,372\,036\,854\,775\,808$ (N = 32).

  de $-2^{127} = -170\,141\,183\,460\,469\,231\,731\,687\,303\,715\,884\,105\,728$ à $2^{127}-1$ (N = 64).

- Certains admettent aussi un type 'unsigned' ou 'uint, unint' pour coder des entiers non signés ; la limite de dépassement est doublée quand le nombre de bits augmente de 1 :

  de 0 à $2^{32}-1 = 4\,294\,967\,295$ (N = 32, 4 octets, type `uint32`).

  de 0 à $2^{64}-1 = 18\,446\,744\,073\,709\,551\,615$ (N = 64, 8 octets, type `uint64`).

Tout calcul dont le résultat dépasserait ces limites, aboutirait à un résultat erroné.

Dans le langage `python` il n'y a en général aucun problème de dépassement pour les entiers :

- `python` pratique le typage dynamique, c'est l'interpréteur qui bascule automatiquement en cas de dépassement d'un format à l'autre. Pour l'utilisateur tout est transparent, et corresponds au seul type 'int'.
- en format court, le nombre est codé signé (en complément à deux) sur 32 ou 64 bits (sur un ordinateur ancien 32 bits/ récent 64 bits).

– en format long, le nombre d'octet alloué est fixé par l'interpréteur selon le besoin et n'a pas de limité prédéfinie : dans la pratique aucun calcul raisonnable ne devrait aboutir à une erreur de dépassement de capacité. Cependant il existe toujours un capacité limitée par les capacités de stockage de l'ordinateur.

Dans le langage `python` il n'y a en général aucun problème de dépassement dans les calculs sur les entiers. L'interpréteur augmente automatiquement l'unité de stockage de façon à éviter toute erreur de dépassement de capacité. Attention ce n'est vrai que pour les entiers.

1.3.4 Représentation des nombres à virgule flottante

Dans la plupart des langages informatiques les nombres "réels" sont représentés par des nombres à virgule flottante. La représentation des nombres à virgule flottante est normalisée par la norme IEEE 754.

Représentation normalisée des nombres à virgule flottante

La plupart des langages proposent deux formats de nombres à virgule flottante : les nombres en précision simple, codés sur $N = 32$ bits, et les nombres en double précision codés sur $2N = 64$ bits.

Théorème 4 *Soit $x \in \mathbb{R}^*$. Il existe un unique couple (m, e) où $m \in [1, 2[$ est appelée la mantisse et $e \in \mathbb{Z}$ est appelé l'exposant, et tel que $x = \pm m \times 2^e$.*

Preuve. On peut supposer que x est strictement positif.
Existence. Soit $d = \log_2(x)$ et $e = \lfloor \log_2(x) \rfloor$. Alors $e \leqslant d < e + 1$, $x = 2^d$ et puisque $x \mapsto 2^x$ est strictement croissante :

$$2^e \leqslant x < 2^{e+1} = 2 \times 2^e \implies 1 \leqslant \frac{x}{2^e} < 2$$

En posant $m = \dfrac{x}{2^e}$, on a bien $x = m \times 2^e$ avec $m \in [1, 2[$ et $e \in \mathbb{Z}$.

Unicité. Supposons que $x = m_1 \times 2^{e_1} = m_2 \times 2^{e_2}$ avec $m_1, m_2 \in [1, 2[$ et $e_1, e_2 \in \mathbb{Z}$. Alors :

$$\frac{m_1}{m_2} = 2^{e_2 - e_1}$$

Puisque $1 \leqslant m_1, m_2 < 2$ alors $\dfrac{1}{2} < \dfrac{m_1}{m_2} < 2$. Puisque $e_2 - e_1 \in \mathbb{Z}$, la seule possibilité est $2^{e_2 - e_1} = 1$, donc $e_1 = e_2$ et $m_1 = m_2$. □

Définition. L'écriture d'un nombre en binaire avec virgule est la donnée de 2 suites finies de 0 et de 1 : u_N, $u_{N-1}, \dots u_1, u_0$ et v_1, $v_2, \dots, v_M$. L'écriture :

$$\underbrace{\overline{u_N\, u_{N-1} \cdots u_1\, u_0}}_{\text{partie entière}}, \underbrace{\overline{v_1\, v_2 \cdots v_{M-1}\, v_M}}_{\text{partie non entière}}$$

représente le nombre (dyadique) :

$$\underbrace{\sum_{k=0}^{N} u_k \times 2^k}_{\text{partie entière}} + \underbrace{\sum_{k=0}^{M} v_k \times 2^{-k}}_{\text{partie non entière}}$$

L'écriture binaire à virgule est semblable à l'écriture décimale en changeant 10 par 2, et les 10 chiffres : 0,1,... 9 par les deux chiffres 0 et 1. Le même principe s'applique pour toute base.

Exemple : $\overline{1010,11}$ représente le nombre : $2^3 + 2^1 + 2^{-1} + 2^{-2} = 8 + 2 + 1/2 + 1/4 = 10,75$.

Bien sûr tous les nombres réels ne peuvent pas se représenter sous cette forme ; les nombres le pouvant sont les nombres dyadiques, c'est-à-dire qui peuvent s'écrire sous la forme $a/2^p$ avec $a, p \in \mathbb{N}$. Cependant, tout comme pour les nombres décimaux, tout nombre réel être approché aussi près que souhaité par une écriture binaire à virgule.

Exemple : Représenter en binaire à virgule le nombre dyadique $a = 123/2^4$:

On commence par représenter 123 en binaire : $123 = 64 + 32 + 16 + 8 + 2 + 1 = \overline{1111011}$, puis on décale la virgule ("imaginairement située à droite de la partie entière") de 4 crans vers la gauche :

$$\frac{123}{2^4} = 7,6875 = \overline{111,1011}$$

Algorithme d'écriture en binaire à virgule.

Pour la partie entière du nombre on applique l'algorithme d'écriture en binaire.
Pour la partie après la virgule. On multiplie la partie non-entière par 2 tant que ce qui est après la virgule reste non-nul.
A chaque étape un 0 ou un 1 apparaît juste avant la virgule : cette suite de 0 et 1, dans l'ordre d'apparition donne la partie binaire après la virgule.

Exemple : Écriture binaire de 7,6875 :
• La partie entière 7 s'écrit en binaire $\overline{111}$.
• Partie après la virgule :
$0,6875 \times 2 = \mathbf{1},375$
$0,375 \times 2 = \mathbf{0},75$
$0,75 \times 2 = \mathbf{1},5$
$0,5 \times 2 = \mathbf{1},0$
La partie après la virgule est $\overline{\ldots,1011}$.
• On obtient l'écriture binaire : $\overline{111,1011}$, (comme ci-dessus).

Remarque. Attention, tout comme pour l'écriture décimale, il n'est pas possible d'écrire tout réel comme un nombre binaire à virgule. Par exemple $0,1 = \dfrac{1}{10}$ n'est pas un nombre dyadique, et sa représentation binaire à virgule serait infinie :
$0,1 \times 2 = \mathbf{0},2$
$0,2 \times 2 = \mathbf{0},4$
$0,4 \times 2 = \mathbf{0},8$
$0,8 \times 2 = \mathbf{1},6$
$0,6 \times 2 = \mathbf{1},2$
etc...
$0,1 = \overline{0,00011\ 00011\ 00011\ 00011\ etc....}$ Représentation infinie.

Il y a une condition nécessaire et suffisante pour la finitude d'une écriture à virgule :
• En décimal, l'écriture à virgule d'un nombre x est finie si et seulement si x est un rationnel $x = p/q$ et son dénominateur q n'a pour facteurs premiers que des 2 et des 5.
• En binaire, l'écriture à virgule d'un nombre x est finie si et seulement si x est un rationnel $x = p/q$ et son dénominateur q n'a pour facteurs premiers que des 2. C'est ce que l'on appelle les *nombres dyadiques*.
• De plus lorsque la représentation est infinie, la partie après la virgule est périodique si et seulement si x est rationnel.

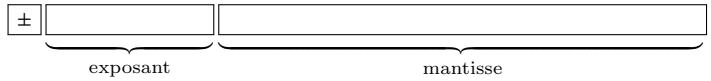
Définition. Représentation d'un nombre en virgule flottante.

La représentation d'un nombre x non-nul en virgule flottante est constituée de trois éléments :

1. son signe $\epsilon = \pm 1$,
2. son exposant $e \in \mathbb{Z}$,
3. sa mantisse m avec $1 \leqslant m < 2$,

tels que $\boxed{x = \epsilon.m.2^e}$.

Signe, exposant et matrice sont encodés sur 32 bits en simple précisions, 64 bits en double précision :



- Le bit de poids fort est réservé pour le signe, 0 : positif, 1 : négatif.
- Les bits suivants sont réservés pour l'exposant, de la façon suivante :
  - En simple précision : 8 bits sont réservés pour l'exposant, on code l'exposant décalé $e + 127$ en binaire, ce qui permet de représenter un exposant e compris entre -127 et 128.
  - En double précision : 11 bits sont réservés pour l'exposant, on code l'exposant décalé $e + 1023$ en binaire, ce qui permet de représenter un exposant e compris entre -1023 et 1024.
- Les bits restants (23 bits en simple précision, 52 bits en double précision) sont réservés pour la mantisse. On y code en binaire la partie non-entière (c.à.d. après la virgule) de l'écriture binaire à virgule de $m - 1$, en l'alignant à gauche.

Attention ! La partie mantisse ne contient que la partie non entière de l'écriture binaire à virgule de la mantisse. Il ne faut pas oublier de rajouter le 1 implicite avant la virgule.

Remarque. La norme IEEE754 définit aussi la représentation en nombre flottant en quadruple précision, sur 128 bits. Le principe est le même, avec un exposant représenté sur 15 bits.

On parle souvent de *flottant* pour désigner un nombre représenté en virgule flottante.

Exemples :

• Cherchons la représentation en nombre flottant simple et double précisions de $x = 125,75$, on a :

$x = 125,75 = 64+32+16+8+4+1+\frac{1}{2}+\frac{1}{4} = \overline{1111101,11} = \overline{1,11110111}.2^6$.

Le signe est positif : 0,

la mantisse est $\overline{1,11110111}$ mais sera représenté $\overline{11110111}$, sans le préfixe '1,'. Les nombres sont derrière la virgule, et le nombre devra être aligné à gauche.

L'exposant est 6, ce qui donne :

– en simple précision : $6+127 = 133 = 128+4+1 = \overline{1000\,0101}$ sur 8 bits,

– en double précision : $6+1023 = 1029 = 1024+4+1 = \overline{100\,0000\,0101}$ sur 11 bits.

Ainsi en simple précision, 125,75 est représenté sur 32 bits :

$$\underbrace{0}_{signe}\ \underbrace{10000101}_{exposant}\ \underbrace{11110111\,000000000000000}_{mantisse}$$

En double précision, 125,75 est représenté sur 64 bits :

$$\underbrace{0}_{signe}\ \underbrace{10000000101}_{exposant}\ \underbrace{11110111\,0000000000000000000000000000000000000000}_{mantisse}$$

Remarque. Pour représenter les nombres négatifs, on utilise pour les flottants les deux modes de représentations autres que l'écriture en complément à deux, que nous avons évoqués en **2.4.1** :
– Pour le nombre représenté, 1 bit de signe suivi de la représentation de sa valeur absolue. Ainsi les représentations en virgules flottantes sont parfaitement symétrique pour les nombres positifs et négatifs.
– Pour l'exposant, la représentation par translation en le décalant de 2^7-1 (simple précision) ou $2^{10}-1$ (double précision).

• Cherchons le nombre flottant dont l'écriture en simple précision est :

$$1\,10000110\,10110111011000000000000$$

Le signe est négatif 1, et l'exposant est $\overline{10000110} = 128+4+2 = 134$ qui donne $134-127 = 7$. L'exposant est 7.

La mantisse est $\overline{10110111011000000000000}$, elle est égale à :

$$1+\frac{1}{2}+\frac{1}{2^3}+\frac{1}{2^4}+\frac{1}{2^6}+\frac{1}{2^7}+\frac{1}{2^8}+\frac{1}{2^{10}}+\frac{1}{2^{11}} = 1,716308594.$$

Le nombre représenté est $-1,716308594\times 2^7 = \boxed{-219,6875}$.

En `python` les nombres flottants sont représentés sur un ordinateur récent en format double précision (sur 64 bits).

Écriture dénormalisée de nombres à virgules flottantes

Il a été nécessaire à la définition de la norme, d'établir quelques exceptions. En effet sans cela on ne peut par même représenter 0 en flottant. Il y a 4 exceptions à l'écriture normalisée (écritures dénormalisées, elles sont hors programme) :

– La première exception définit le(s) zéro.
– La deuxième exception permet la représentation de nombres plus petits.
– La troisième exception définit les deux "infinis".
– La quatrième exception définit les "Not a number" `nan`.

Voici ces quatres exceptions (elles ont hors-programme !) :

1. Si exposant décalé et mantisse sont tous deux nuls : le nombre est 0. Attention il y a deux zéros :

$$\underbrace{0}_{signe}\ \underbrace{000\cdots000}_{exposant}\ \underbrace{0000000000\cdots0000000000}_{mantisse} = 0^+$$

$$\underbrace{1}_{signe}\ \underbrace{000\cdots000}_{exposant}\ \underbrace{0000000000\cdots0000000000}_{mantisse} = 0^-$$

2. Si l'exposant décalé est 0 et la mantisse est non nulle on n'ajoute non plus 1 mais 0 à la mantisse et on prend l'exposant $e=-126$ en précision simple ($e=-1022$ en précision double) plutôt que $e=-127$ (respectivement $e=-1023$).

3. Si l'exposant décalé est $2^N-1 = \overline{111\cdots111}$ ($N=32$ ou 64) et la mantisse est nulle : on obtient les 2 infinis selon le signe :

$$\underbrace{0}_{signe}\ \underbrace{111\cdots111}_{exposant}\ \underbrace{0000000000\cdots0000000000}_{mantisse} = \texttt{+inf}$$

$$\underbrace{1}_{signe}\ \underbrace{111\cdots111}_{exposant}\ \underbrace{0000000000\cdots0000000000}_{mantisse} = \texttt{-inf}$$

4. Si l'exposant décalé est $2^N-1 = \overline{111\cdots111}$ et la mantisse est non nulle on obtient un `NaN` : Not a Number.

Limitations de la représentation des nombres en virgules flottantes

Ces limitations dépendent bien sûr du format, simple précision, double précision, etc.. Nous nous intéressons au format double précision, sur 64 bits, puisque c'est celui que l'on manipulera en `python`.

Limitations de la représentation des nombres en virgules flottantes sur 64 bits :

• En virgule flottante double précision, on ne peut par représenter de nombre supérieur ou égal à 2^{1024}. En effet l'exposant le plus grand est $\overline{111\,1111\,1110} = 2047$ qui donne comme exposant $2046 - 1023 = 1023$.

• Le plus grand nombre que l'on peut représenter en format double précision s'écrit :

$$\underbrace{0}_{+}\ \underbrace{111\,1111\,1110}_{e=1023}\ \underbrace{1111111111111111111111111111111111111111111111111111}_{\text{mantisse}}$$

et vaut :

$$2^{1023} \times \sum_{k=0}^{52} \frac{1}{2^k} = 2^{1023} \times \frac{1 - \dfrac{1}{2^{53}}}{1 - \dfrac{1}{2}} = 2^{1024} - 2^{971} \approx 1,798 \times 10^{308}$$

• Le plus petit écart en deux nombres entre 1 inclu et 2 exclu vaut 2^{-52}. En effet il y a exactement 2^{52} nombres flottants dans $[1, 2[$ équi-répartis.
Le plus petit écart entre deux nombres entre 2 inclu et 4 exclu vaut 2^{-51} ; entre 4 et 8, l'écart est au moins de 2^{-50} ; entre 0,5 (inclu) et 1 exclu, le plus petit écart vaut 2^{-53}.
De façon générale, pour k compris entre -1022 et 1023 : entre les nombres 2^k inclu et 2^{k+1} exclu, il y a exactement 2^{52} flottants équi-répartis, et l'écart minimal vaut 2^{k-52}.

• Le plus petit nombre strictement positif que l'on peut représenter est 2^{-1074}. Il s'écrit :

$$\underbrace{0}_{+}\ \underbrace{000\cdots000}_{e=-1022}\ \underbrace{000\cdots0001}_{\text{mantisse}}$$

On est dans le cas (2) des écritures dénormalisées : cette écriture représente $2^{-1022} \times \left(0 + 2^{-52}\right) = 2^{-1074}$.

• Ainsi comparer deux nombres flottants n'a en général que peu de sens.

• Il faut prendre garde aux problèmes de dépassements de capacités dans la représentation des nombres flottants.

Remarque. Contrairement au cas de la représentation des entiers, l'interpréteur `python` représente les flottants en format double précision (sur un ordinateur 64 bits) et n'adapte pas le format en cas d'erreur de dépassement. Il faut prendre bien garde aux problèmes de dépassement de capacité lors de calculs en nombres flottants.

Illustration graphique de la répartition de flottants

Pour produire une illustration graphique de la répartition des nombres flottants, nous avons considéré un format de flottant -qui n'existe pas- le format flottant sur 8 bits :

– 1 bit, celui de poids le plus fort, pour le signe,
– les 2 bits suivants pour l'exposant $e \in [[-1, 2]]$. On code $e + 1$ sur 2 bits en binaire.
– les 5 bits restants pour la mantisse.
Le principe du codage est par ailleurs le même que dans l'écriture normalisée des nombres à virgules flottantes.

Par exemple :
• $\overline{0\,10\,11111}$: le signe est +, l'exposant est $2 - 1 = 1$, la mantisse est $1,11111 = \displaystyle\sum_{k=0}^{5} \frac{1}{2^k} = \frac{1 - \dfrac{1}{2^6}}{1 - \dfrac{1}{2}} = 2 - \frac{1}{2^5} = 1,96875.$
Le nombre représenté est $2 \times 1,96875 = 3.9375$.

• Pour représenter $2,75$: $2,75 = 2 \times 1,375$. L'exposant est 1, et la mantisse est $1,375 = 1 + 0,25 + 0,125 = \overline{1,011}$. On obtient pour représentation en flottant 8 bits : $\overline{0\,10\,01100}$.

• Le plus petit nombre positif que l'on peut représenter s'écrit $\overline{00000000}$ et donne $= +2^{-1} \times 1 = 0,5$.

• Le plus grand nombre que l'on peut représenter s'écrit $\overline{01111111}$. L'exposant est 2, et la mantisse $\overline{1,11111} = 1,96875$ et donne $= +2^2 \times 1,96875 = 7,875$.

On constate au passage pourquoi ce format flottant sur 8 bits est sans intérêt : on ne peut représenter que des nombres dans $]-8\ ;\ -0,5] \cup [0,5\ ;\ 8[$.

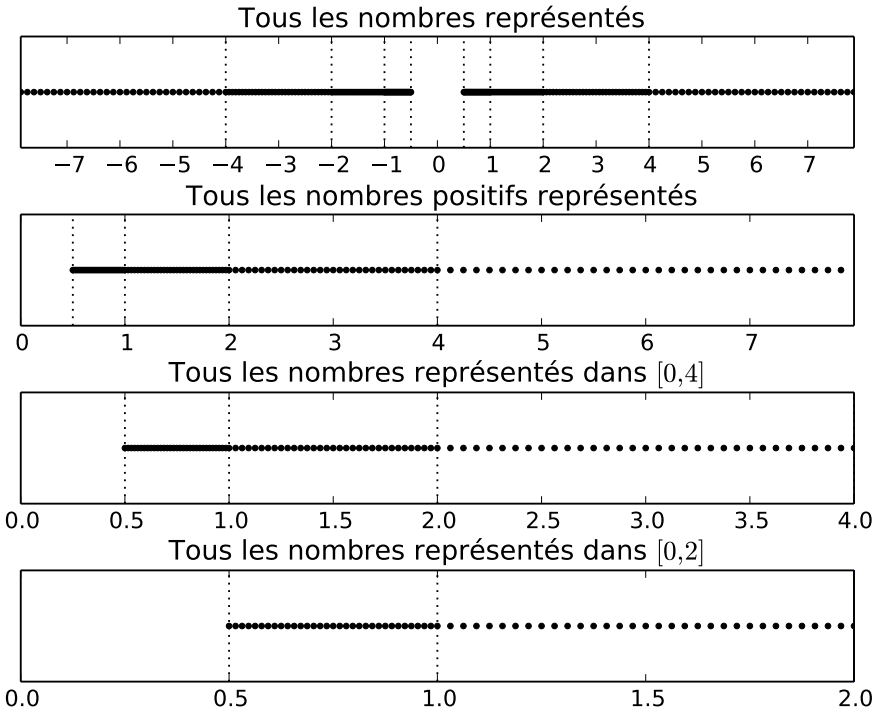
On a tracé dans le plan sur 4 graphiques des points sur l'axe réel pour chaque nombre représenté en flottant 8 bits.

– Sur le premier graphique, tous les nombres représentés,

– Sur le deuxième graphique, tous les nombres positifs représentés,

– Sur le troisième graphique, tous les nombres de $[0, 4]$ représentés :

– Sur le quatrième graphique, tous les nombres de $[0,2]$ représentés :



On constate :

– La parfaite symétrie entre les positifs et négatifs représentés.

– L'absence du zéro, et de nombre voisins de zéros. C'est dû à ce que l'écriture utilisée est uniquement celle normalisée.

– Le plus petit nombre strictement positif représenté est 0,5. Le plus grand est $7,875$.

– Il y a autant de nombres représentés entre 0,5 inclus et 1 exclu, qu'entre 1 et 2, entre 2 et 4, et entre 4 et 8 ; chaque fois $2^5 = 32$.

– L'écart est constant sur chaque intervalle $\left[2^k, 2^{k+1}\right]$; il double de $[0,5,1]$ à $[1,2]$, puis à $[2,4]$ et $[4,8]$: cet écart vaut respectivement 2^{-6}, 2^{-5}, 2^{-4} et 2^{-3}.