1 Des 0 et des 1

Un ordinateur possède de la mémoire vive, appelée RAM pour « Random Access Memory ». C'est cette mémoire qui représente physiquement l'état de l'ordinateur. Elle est en perpétuelle évolution lors de l'exécution des programmes qui tournent sur la machine.

Pour enregistrer des données, un ordinateur possède un très grand nombre de « bits » qui ne peuvent être que dans deux états : 0 ou 1. En pratique, ce sont des condensateurs qui sont soient chargés, soit déchargés. Un bit suffit pour stocker un booléen. On prend la convention qui consiste à dire que si le bit est dans l'état 0, le booléen est faux et que si le bit est dans l'état 1, le booléen est vrai.

Supposons maintenant que l'on souhaite stocker la direction de déplacement d'un joueur sur une carte et que seulement 4 directions sont possibles : nord, sud, est, ouest. Pour cela, il nous suffit d'avoir 2 bits. On peut dire que par convention `00` représente le nord, `11` représente le sud, `10` représente l'est et `01` représente l'ouest.

On comprend alors que 3 bits suffisent pour représenter 8 valeurs et plus généralement n bits suffisent pour représenter 2^n valeurs. En particulier, avec 8 bits, vous pouvez donc représenter 256 valeurs. Un octet est formé de 8 bits. C'est la plus petite quantité de mémoire avec laquelle les ordinateurs travaillent. Un octet permet donc de stocker un nombre entre 0 et 255 inclus. En se mettant d'accord sur une correspondance entre les nombres et les caractères les plus élémentaires, on peut donc stocker un caractère dans un octet. La table de correspondance la plus utilisée est la table ASCII. Un octet, c'est aussi la mémoire demandée pour stocker la luminosité d'un pixel d'une image en noir et blanc. L'oeil peut difficilement discerner plus de 256 niveaux de gris et un ordinateur stocke donc le niveau de gris d'un pixel avec un nombre entre 0 et 255. On comprend pourquoi l'octet est devenu l'unité de base pour décrire la quantité de mémoire. En anglais, un octet se dit « byte » qu'il ne faut surtout pas confondre avec le « bit ».

La bible contient a peu près 4 millions de caractères. Il faut donc le même nombre d'octets pour la stocker. Comme en physique avec le mètre et le kilomètre, on aimerait avoir une unité pour spécifier 1000 octets. Seulement, il se trouve que les puissances de 2 sont très utiles en informatique et que $2^{10} = 1024$. Certaines personnes ont décidé de choisir qu'un kilooctet représente 1000 octets. D'autres ont décidé qu'un kilooctet représente 1024 octets. Certains organismes ont tranché et décidé qu'un kilooctet devait faire 1000 octets et que 1024 octets s'appeleraient un kibioctet. Autant vous dire que ces organisations sont autant écoutées que l'Académie Française quand elle demande d'écrire un CD-ROM, Cédérom. Après tout, la différence n'est que de 2%, mais nous allons voir que les choses de gâtent quand on passe au mégaoctet. Pour certaines personnes, le mégaoctet représente 1024 ko (celui qui représente 1024 octets), soit 2^{20} octets. Pour d'autres, un mégaoctet, c'est simplement 1000 ko (celui qui représente 1000 octets) soit un million d'octets. La différence est maintenant d'à peu près 5%. Quand on passe au gigaoctet, qui représente selon les personnes 2^{30} octets et selon les autres un milliard d'octets, la différence est de 7%. Au téraoctet, la différence est de 10%. Je vous laisse imaginer quel choix ont fait les vendeurs de disques durs. Dans ce cours, nous ne calculerons que des ordres de grandeur et la différence ne sera pas importante. On retiendra que la bible encodée en ASCII occupe à peu près 4 Mo de mémoire. C'est à peu près la taille nécessaire pour stocker une chanson en MP3 dans une qualité standard. Si vous regardez un film sous Netflix en qualité haute définition, il faudra compter 1 Go. Le même film tel qu'il est diffusé dans un cinéma sera très peu compressé et utilisera à peu près 1 To. Pour information, un ordinateur portable actuel a souvent une RAM de quelques Go (disons entre 2 Go et 32 Go) et un disque dur entre 256 Go et 2 To.

Bref, on retiendra qu'avec suffisamment de 0 et de 1, on peut stocker à peu près n'importe quelle information. Il suffit juste de construire une correspondance entre les suites de 0 et de 1 et ces informations. Nous l'avons donné pour les booléens. Dans ce chapitre, nous allons décrire cette correspondance pour les entiers et le nombres flottants.

Exercice 1

Combien de bits sont-ils nécessaires pour représenter tous les élèves de la classe ?

---

2 Représentation en mémoire des entiers

Proposition 1. *Soit b un entier tel que $b \geqslant 2$, $n \in \mathbb{N}$ et $m \in [\![0, b^n - 1]\!]$. Alors il existe un unique n-uplet $(d_0, d_1, \dots, d_{n-1})$ d'éléments de $\{0, 1, \dots, b-1\}$ tel que :*

$$m = \sum_{k=0}^{n-1} d_k b^k.$$

On dit que $(d_0, \dots, d_{n-1})$ est la décomposition en base b de m.

Nous utilisons tous la notation décimale qui est une décomposition en base 10. En effet, le nombre 81 s'écrit

$$85 = 5 \times 10^0 + 8 \times 10^1$$

mais on peut aussi décomposer ce nombre en base 2

$$85 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6$$

en base 8, ou en base 16 :

$$85 = 5 \times 8^0 + 2 \times 8 + 1 \times 8^2 \text{ et } 85 = 5 \times 16^0 + 5 \times 16^1$$

En mettant les « chiffres de poids fort » en tête, on écrit donc

$$85 = \underline{85}_{10}, \quad 85 = \underline{1010101}_2 \quad 85 = \underline{125}_8$$

Enfin, pour la base 16 (hexadécimale), on utilise A pour représenter 10, B pour représenter 11, jusqu'à F pour représenter 16. Par exemple, $85 = \underline{55}_{16}$ et $31 = \underline{1F}_{16}$.

Exercice 2

Expliquer comment obtenir d_k à partir de m, b et k.

Pour $b = 2$, la proposition précédente nous dit simplement que l'application de $\{0,1\}^n$ dans $[\![0, 2^n - 1]\!]$ qui à $(d_0, \dots, d_{n-1})$ associe

$$\sum_{k=0}^{n-1} d_k 2^k$$

est une bijection. La décomposition en base 2 nous donne donc un moyen de représenter les nombres positifs avec des 0 et des 1. Comme on ne peut représenter que les entiers positifs, on parle de représentation des entiers *non signés*. Généralement, les ordinateurs actuels codent les entiers sur 8, 16, 32, ou 64 bits. Avec 8 bits, on peut coder tous les entiers entre 0 et 255. Avec 32 bits on peut coder tous les entiers entre 0 et $2^{32} - 1 = 4\,294\,967\,295$ soit quelques milliards. Avec 64 bits on peut coder les entiers entre 0 et quelques milliards de milliards.

Si on veut à la fois stocker des entiers positifs et négatifs, une astuce est nécessaire. Si on représente les nombres sur n bits, on va choisir de représenter tous les entiers entre et -2^{n-1} et $2^{n-1} - 1$ inclus. On parle de représentation des entiers *signés*.

— Tous les nombres entre 0 et $2^{n-1} - 1$ sont représentés par leur décomposition binaire.
— Pour les nombres m entre -2^{n-1} et -1, on les représente par la décomposition binaire de $m + 2^n$. Autrement dit, on les stocke modulo 2^n.

Cette manière de stocker les nombres est très pratique pour les ordinateurs. On l'appelle méthode du *complément à 2*. Par exemple, si $n = 8$, -1 est représenté par $\underline{11111111}_2$.

Dans de nombreux langages de programmation de bas niveau comme le C, seuls les nombres stockés sur n bits (avec $n = 8, 16, 32$ ou 64) sont disponibles et les opérations d'addition et de multiplication sont toutes écrites modulo 2^n. Par exemple, si on travaille en 8 bits et que l'on ajoute 1 à 127, on tombe sur -128 ! En OCaml, pour des raisons d'optimisation, les entiers sont des entiers signés sur 63 bits. L'avantage de ce stockage en mémoire pour les nombres négatifs est que le processeur peut ajouter ou multiplier les nombres de la même manière sans se soucier de leur signe.

Enfin, certains ordinateurs stockent les bits de poids faible en premier (Little Indian) alors que d'autres stockent les bits de poids fort en premier (Big Indian). Ces dénominations proviennent des « Voyages de Gulliver » de Jonathan Swift où certains indiens ouvrent les oeufs du côté le plus pointu tandis que d'autres les ouvrent du côté le plus rond. La plupart des ordinateurs actuels utilisent le format Little Indian et on a moins de pagaille lorsque les ordinateurs s'échangent des données.

Exercice 3

On suppose que l'on utilise une machine représentant les entiers signés sur 8 bits.

1. Quels sont les entiers représentables ?
2. Comment s'écrivent -7, 7,13,-1, 3 ?

En Python, le type `int` permet de représenter les entiers sans limite de taille. En interne, le langage adapte automatiquement la représentation en fonction de la taille de l'entier. Pour faire simple, on peut dire que Python stocke le signe de l'entier sur 1 bit. Il stocke ensuite sa valeur absolue en tant qu'entier non signé en utilisant 64 bits si il est inférieur à $2^{64} - 1$, 128 bits si il est inférieur à $2^{128} - 1$, etc. La représentation exacte est légérement différente, mais l'idée est là.

3 Représentation mémoire des flottants

Proposition 2. *Soit b un entier tel que $b \geqslant 2$ et x un réel non nul. Alors il existe un unique triplet (s, m, e) où $s \in \{+1, -1\}$, (m_k) est une suite d'éléments de $[\![0, b-1]\!]$ qui n'est pas égale à $b-1$ à partir d'un certain rang et $e \in \mathbb{Z}$ tels que :*

$$x = s \left(\sum_{k=0}^{+\infty} \frac{m_k}{b^k} \right) b^e$$

On dit que cette décomposition est l'écriture scientifique de x en base b.

Le somme infinie fait référence à limite de la suite de terme général

$$u_n = \sum_{k=0}^{n} \frac{m_k}{b^k}$$

qui est bien convergente comme suite croissante, majorée par b. Pour $b = 10$, on obtient l'écriture classique sous forme scientifique. Par exemple :

$$e^\pi = 2.314069 \cdots \times 10^1$$

Il est à noter qu'il convient bien d'imposer à la suite de ne pas être égale à $b-1$ à partir d'un certain rang pour avoir l'unicité. En effet

$$1 = 1.000 \cdots \times 10^0 \text{ et } 1 = 9.999 \cdots \times 10^{-1}$$

On dit qu'un nombre est décimal lorsque (m_k) est nulle à partir d'un certain rang pour $b = 10$. Remarquons pour que les nombres rationnels, la suite (m_k) est périodique à partir d'un certain rang. Par exemple :

$$\frac{22}{7} = 3.142857\underline{142857} \cdots \times 10^0$$

On peut même montrer que cette propriété caractèrise les nombres rationnels.

Pour calculer effectivement les m_k pour un nombre strictement positif x, on commence par déterminer e tel que $y = x/b^e \in [1, b[$.

Exercice 4

Donner une manière de calculer m_k à partir de y et b.

Exercice 5

Donner le développement décimal de 0.1 et 0.3 en base 10, puis en base 2.

Définition 1. *Soit $p \in \mathbb{N}^*$. On dit qu'un réel x est un nombre flottant représentable avec une mantisse de p bits lorsqu'il existe $m_0, \dots, m_{p-1}$, éléments de $\{0,1\}$, et $m \in \mathbb{Z}$ tels que :*

$$x = \pm \left(\sum_{k=0}^{p-1} \frac{m_k}{2^k} \right) 2^m$$

Si x est non nul et si on impose $m_0 = 1$, cette écriture est unique. L'ensemble des nombres flottants représentables avec une mantisse de p bits est noté $\mathcal{F}_p$.

Par exemple $2.5 = (1 + 0/2 + 1/4) \times 2^1 \in \mathcal{F}_3$. On note que tous les entiers compris entre $-2^p + 1$ et $2^p - 1$ sont des éléments de $\mathcal{F}_p$. Enfin, les rationnels $r = \pm a/b$ (avec a et b premiers entre eux) tels que b n'est pas une puissance de 2 ne sont pas dans $\mathcal{F}_p$. En particulier $0.1 = 1/10$ et $1/3$ ne sont pas dans $\mathcal{F}_p$.

Les nombres flottants sont construits pour pouvoir approcher de manière assez fine n'importe quel nombre avec une *erreur relative* assez faible.

Proposition 3. *Soit $x \in \mathbb{R}$. Alors il existe au moins un élément f de $\mathcal{F}_p$ minimisant la distance de x à $\mathcal{F}_p$. De plus :*

$$|x - f| \leqslant u\,|x| \quad \text{avec } u = 2^{-p}$$

Cet élément f est unique sauf lorsque f est au milieu de deux éléments successifs de $\mathcal{F}_p$. Dans ce cas, un seul de ces deux éléments a une décomposition dyadique normalisée telle que $m_{p-1} = 0$. On définit alors $\mathcal{R}_p(x)$ comme étant cet unique élément.

Définition 2. *Si $p, q \in \mathbb{N}^*$, on note $\mathcal{F}_{p,q}$ l'ensemble formé de :*

- *Les réels non nuls x de la forme*

$$x = \pm \left(m_0 + \frac{m_1}{2} + \frac{m_2}{4} + \cdots + \frac{m_{p-1}}{2^{p-1}} \right) 2^e$$

  avec $m_0, \dots, m_{p-1} \in \{0,1\}$ et $-2^{q-1} + 2 \leqslant e \leqslant 2^{q-1} - 1$.
- *Des éléments 0^+, 0^-, $+\infty$ et $-\infty$.*
- *Un élément noté* NaN *(Not a Number)*

Les ordinateurs actuels proposent en général deux types de nombres flottants. Le premier format, codé sur 32 bits, est appelé simple précision et dans ce format $p = 24$ et $q = 8$. Dans ce format $u = 2^{-24} \approx 5.9 \times 10^{-8}$. Dans l'autre format, codé sur 64 bits et appelé double précision, on a $p = 53$ et $q = 11$. C'est le format auquel Python nous donne accès avec son type `float`. Dans ce format $u = 2^{-53} \approx 1.1 \times 10^{-16}$. Le plus petit nombre représentable est de l'ordre de 10^{-324} et le plus grand nombre représentable est de l'ordre de 10^{308}.