Chapitre 1

Représentation des nombres

1.1 Introduction aux bases numériques

Définition

Soit b un entier naturel ≥ 2. On considère un ensemble C de b caractères, usuellement des chiffres à partir de 0, en complétant avec des lettres dans l'ordre alphabétique, donnés dans l'ordre croissant.
On représente un entier naturel en base b en écrivant des caractères de l'ensemble C en séquence, en précisant la valeur de b pour lever toute ambiguïté.
La notation proposée dans ce cours, qui n'est pas uniformisée, est $\overline{a_{n-1}a_{n-2}\dots a_1a_0}^b$, pour représenter le nombre $\sum_{i=0}^{n} a_i b^i$. Dans le nombre en question, le caractère dit de poids fort est le plus à gauche et le caractère dit de poids faible est le plus à droite.

Les bases usuelles sont 10 (base décimale, utilisée de façon naturelle actuellement), 2 (le binaire, fondement de l'informatique) et 16 (les nombres hexadécimaux, utilisés en informatique pour éviter d'avoir trop de chiffres). On trouve également les bases 8 et 12 de manière cependant moins prononcée.

Toutes les opérations arithmétiques apprises au primaire se font de manière similaire dans n'importe quelle base. Par exemple, en base 8, $\overline{35}^8 + \overline{56}^8 = \overline{113}^8$ car $5 + 6$, qui correspond au nombre 11, s'écrit $\overline{13}^8$. On note donc 3 et on retient 1, de même qu'on a une retenue pour les « huitaines », qui devient la « soixante-quatraine ».

Exercice

Calculer en base 8 les sommes $\overline{146}^8 + \overline{334}^8$ et $\overline{357}^8 + \overline{464}^8$, ainsi que le produit

$\overline{54}^8 \times \overline{465}^8$.

La représentation en base b s'étend aux entiers relatifs en précisant le signe.

Proposition

Soit b un entier naturel ≥ 2. On peut représenter un nombre rationnel en base b de manière exacte, c'est-à-dire avec un nombre fini de caractères, si et seulement si le nombre en question est le quotient d'un entier relatif par une puissance de b. Une autre formulation : le nombre en question a pour fraction la plus simplifiée $\frac{p}{q}$, où tous les diviseurs premiers de q sont des diviseurs de b. La représentation comporte alors éventuellement une virgule, et on écrit $\overline{a_{n-1}a_{n-2}\dots a_1a_0,a_{-1}a_{-2}\dots a_{-m}}^b = \sum_{i=-m}^{n-1} a_i b^i$.

Remarque : En base 10, on retrouve la notion de nombres décimaux. Bien entendu, quelle que soit la valeur de b, un nombre irrationnel aura toujours un nombre infini de caractères après la virgule dans sa représentation en base b.

Exemple : Le nombre $\frac{13}{3}$ s'écrit de manière exacte dans toute base multiple de 3.

Proposition

[Pour la culture] Quelle que soit la valeur de b, tout nombre rationnel r a une écriture en base b ultimement périodique, c'est-à-dire qu'à partir d'un certain rang fini, un même motif se répète.

Cette proposition se prouve simplement à l'aide du petit théorème de Fermat quand b est premier (sinon, c'est de l'arithmétique avancée). On peut l'illustrer en considérant les fractions $\frac{1}{p}$ en base b. Si p divise b, alors l'écriture de $\frac{1}{p}$ est exacte : $\frac{1}{p} = \overline{0,a}^b$, où a est le caractère correspondant à l'entier $\frac{b}{p}$, la période étant alors de 1 et le motif étant 0. Sinon, le petit théorème de Fermat dit que b^{p-1} est congru à 1 modulo p, donc diviser 1 par p laisse entrevoir une répétition après $p-1$ étapes. La période est donc $p-1$ ou un de ses diviseurs.

1.2 Représentation des entiers

1.2.1 Entiers naturels

Les informations dans les ordinateurs peuvent être vues comme des signaux électriques. On peut résumer la pratique à cela : chaque unité de mémoire peut être dans deux états, et donc aucun nombre n'est stocké en tant que tel, mais représenté à l'aide d'unités de mémoire dont l'interprétation dépend du contexte.

Ainsi, un ordinateur travaille avec des nombres constitués de 0 et de 1, donc en binaire, et on appelle « *bits* » (pour "binary digits", soit chiffres binaires) les caractères de base du binaire quand ils sont utilisés en informatique. On regroupe éventuellement par paquets de 4 (pour former un caractère hexadécimal) ou de 8 (des *octets*) les bits dans un souci de concision.

Exercice

Au passage, jusqu'à combien peut-on compter sur les doigts ? Et en considèrant qu'on peut les plier selon différents degrés ? Quels problèmes pratiques aurait-on si on voulait utiliser autre chose que du binaire ? (Il s'avère que la logique ternaire a été envisagée et n'est pas totalement tombée dans l'oubli.)

Par exemple, le protocole WEP [1] utilise des clés de 64 ou 128 bits, dont un vecteur d'initialisation et la clé proprement dite, formée de respectivement 10 ou 26 caractères hexadécimaux, qu'on peut aussi écrire comme respectivement 5 ou 13 caractères alphanumériques à l'aide d'une table de 256 caractères.

Exercice

D'ailleurs, combien y a-t-il de bits dans le vecteur d'initialisation ? C'est en fait la méthode de cryptage utilisée avec ce vecteur qui engendre la faille du protocole.

En pratique, la limite de la mémoire d'un ordinateur empêche évidemment d'écrire n'importe quel entier naturel, et on parlera d'entiers sur n bits (souvent 32 ou 64).

1. très peu fiable

Par exemple, les dates d'évènements [2] sont usuellement stockées à l'aide de ce qu'on appelle "timestamp". Cette horloge compte les secondes écoulées depuis le premier janvier 1970 à minuit UTC (pour les systèmes Unix), ce qui causera des problèmes le 19 janvier 2038 pour les systèmes sur 32 bits (qui ont de toute façon déjà presque disparu actuellement). Ceci nous amène à parler de *dépassement arithmétique*.

Les opérations arithmétiques, en binaire comme en décimal et dans toutes les bases, font occasionnellement apparaître des retenues. Que se passe-t-il quand la retenue apparaît ou se propage au-delà du dernier caractère disponible ? Elle est tout simplement perdue. Ainsi, dans la représentation des entiers naturels sur 32 bits, $2^{31} + 2^{31}$ donne 0. C'est ce qu'on appelle le dépassement arithmétique.

1.2.2 Entiers relatifs

Naïvement, on peut penser à réserver un bit dans la représentation d'un entier naturel pour préciser le signe. Cette méthode a l'avantage de la simplicité, mais plusieurs inconvénients, dont l'existence de deux versions de zéro, font qu'en pratique elle est laissée de côté.

Une autre façon de faire pourrait être de considérer que puisqu'on peut écrire les nombres de 0 à $2^n - 1$ sur n bits, on soustrait 2^{n-1} à tous les nombres représentés. L'inconvénient majeur réside cette fois dans les opérations, qui n'ont rien d'évident.

On emploie donc pour représenter un entier relatif la notation dite en *complément à deux*. Il s'agit de distinguer les cas suivant que l'entier soit positif (auquel cas le premier bit est à zéro) ou strictement négatif (auquel cas le premier bit est à 1). Dans le premier cas, on écrit simplement l'entier que l'on veut représenter sur les $n - 1$ derniers bits, et dans le deuxième cas on écrit en fait l'entier plus 2^n sur les n bits.

La notation en complément à deux permet alors d'écrire sur n bits les entiers entre -2^{n-1} et $2^{n-1} - 1$.

Exercice

Il est utile de connaître la représentation des nombres les plus classiques. Elle est évidente pour les nombres positifs, retenir celles de -2^{n-1}, de -1 et d'autres opposés de puissances de 2 est incontournable.

2. notamment pour les logs, il ne s'agit pas d'histoire ici

Propriété : Dans la notation en complément à deux, pour calculer l'opposé d'un entier x, on remplace tous les 0 par des 1 et vice-versa, puis on ajoute 1 à la représentation. Ceci ne marche évidemment pas pour -2^{n-1} (qui s'écrit avec un 1 au début et des 0 partout ailleurs), car son opposé n'est pas représentable sur n bits avec cette notation[3]

Exercice

Puisque l'opposé est une opération involutive, c'est-à-dire que la répéter fait revenir au nombre de départ, on se rend compte que retirer 1 puis remplacer tous les 0 par des 1 et vice-versa donne aussi l'opposé. Exercice pour le lecteur : prouver ceci, ainsi que la propriété ci-dessus.

Comme pour les entiers naturels, le problème du dépassement arithmétique demeure. Sur n bits, ajouter 1 à $2^{n-1}-1$ donne donc -2^{n-1} et non 2^{n-1}. Cela donne en particulier des calculs modulo 2^n.

Pour pallier ce problème, lorsque des grands nombres sont requis, et pour éviter de faire des opérations sur des nombres comprenant trop de chiffres, on coupe les nombres en paquets de $\lfloor \frac{n}{2} \rfloor - 1$ bits[4]. Les langages de programmations disposent généralement d'un type appelé "long"[5], qui reprend cette idée. Pour $n = 32$, les nombres sont donc découpés en paquets de 15, dont le premier paquet donne deux informations : le signe du nombre (premier bit) et le nombre de paquets de 15 bits (sur 15 bits). Les paquets de 15 sont stockés dans des tableaux de taille 16 (en répercutant les retenues aux bons endroits pour que le premier bit de chaque tableau soit à 0 à la fin de chaque calcul).

Le nombre le plus grand qui peut ainsi être représenté pour $n = 32$ est alors $2^{15\times(2^{15}-1)} - 1$, et son écriture tient sur environ 1 Mb. Conseil : ne pas tester à

3. Par conséquent, l'opération proposée redonnerait nécessairement le même nombre. C'est heureusement aussi le cas pour 0.

4. Ceci permet de ne pas avoir de dépassement arithmétique pour les multiplications.

5. Python 3 a le bon goût de fusionner ce type avec le type "int", et nous échappons donc généralement aux dépassements.

faire afficher le nombre le plus grand qui peut être représenté lorsque $n = 64$ [6]. En pratique, pour $n = 64$, le dépassement arithmétique interviendrait alors après un dépassement de mémoire.

1.3 Représentation des réels

Le nom donné en informatique à la représentation des nombres non entiers est la *virgule flottante* (le type correspondant est "float").

Rappelons qu'aucun nombre irrationnel ne peut être représenté de manière exacte avec des caractères de n'importe quelle base. D'ailleurs, tous les nombres représentables de manière exacte en base 2 sont en particulier décimaux, tandis que dans l'autre sens, le nombre $0,2$ par exemple, aussi innocent soit-il, a un développement infini en base 2, à savoir $\overline{0,001100110011\ldots}^2$.

La représentation des nombres réels, qui est habituellement approximative, se fonde sur la base 2 et sur l'écriture dite scientifique, normalement connue pour les nombres décimaux [7].

Grâce à la représentation à virgule flottante, les nombres très grands ou très petits peuvent s'écrire sans une surcharge de caractères peu représentatifs au niveau de la virgule. De toute façon, les limites de la mémoire imposent de procéder rapidement à un arrondi.

Selon la norme IEEE754, avec 64 bits, un nombre en virgule flottante est alors un produit $(-1)^s \times m \times 2^n$, où s est bit de signe, m est la *mantisse* $\overline{1,b_1b_2\ldots b_{52}}^2$ (puisque le 1 est systématique, il n'entre pas dans la représentation qui est donc sur 52 bits) et n est un entier relatif entre -1022 et 1023 écrit sur 11 bits et représenté comme $n + 1023$. Avec 32 bits, la taille de la mantisse passe à 23 bits et l'exposant est sur 8 bits. La disposition des bits est la suivante : d'abord s, puis n, puis m.

On notera que pour comparer deux nombres écrits en virgule flottante, on regarde d'abord le signe, puis à même signe on regarde l'exposant, puis à même exposant on regarde la mantisse.

---

6. L'ordinateur pourrait remplacer avantageusement un radiateur en panne...

7. Rappel : l'écriture scientifique revient à écrire un nombre non nul, éventuellement en tant qu'arrondi, sous la forme $x \times 10^k$, où $k \in \mathbb{Z}$ et $1 \leq x < 10$, et l'écriture est unique.