Codage binaire

Antoine Louatron

Table des matières

I Représentation des nombres **3**
I.1 Nombres entiers . . . 3
I.2 Nombres réels quelconques . . . 4
I.3 Opérations sur les bits . . . 5

II Représenter les textes **5**
II.1 Principe de l'ASCII . . . 5
II.2 Etendre la table des caractères . . . 5

III Représentation d'autres données **5**
III.1 Images et vidéos . . . 5
III.2 Musique . . . 5
III.3 Compression . . . 5

I Représentation des nombres

I.1 Nombres entiers

I.1.1 Représentation décimale

La manière habituelle pour écrire les nombres est d'utiliser "la base 10". C'est à dire qu'un nombre comme 8736 est en fait écrit comme somme de puissances croissantes de 10 (avec un coefficient < 10) :

$$8736 = 6 \times 10^0 + 3 \times 10^1 + 7 \times 10^2 + 8 \times 10^3$$

(Rappel : pour tout nombre $x \neq 0, x^0 = 1$.

I.1.2 Proposition
Tout nombre entier naturel admet une unique représentation *binaire*, c'est à dire qu'il s'écrit d'une manière unique comme une somme de certaines puissances de 2. Plus précisément, pour tout entier naturel $n \neq 0$ il existe un unique $r \in \mathbb{N}$ et une unique famille $(b_0, \ldots, b_r) \in \{0,1\}^{r+1}$ telle que

$$n = \sum_{k=0}^{r} b_i 2^i$$

I.1.3 Exemple
Prenons $37 = 7 + 3 \times 10$.

On a également $37 = 32 + 5 = 32 + 4 + 1 = 1 + 2^2 + 2^5$. On note $100101_{(2)}$ cette représentation binaire. Les 1 marquent la position des puissances de 2 présentes. Le chiffre le plus a droite est la coefficient de 2^0.

I.1.4 Algorithme de conversion

— **Problème :** étant donné un nombre $n \in \mathbb{N}$, calculer sa représentation en base 2 (binaire).
— **Méthode :** effectuer la division euclidienne (avec reste qui vaut donc 0 ou 1) de n par 2 et noter le reste. Recommencer la procédure avec le quotient obtenu jusqu'à obtenir 0.
— **Solution :** noter les reste successifs de droite à gauche pour lire la représentation binaire.

I.1.5 Exercice
Appliquer à 357*,* 1278*,* 4096*.*

I.1.6 Application à l'informatique

Les mémoires informatiques sont ainsi faites que l'on ne peut stocker que des informations binaires (chaque "cellule" (bit) est dans l'un de deux états bien définis). On peut maintenant stocker des nombres entiers dans des mémoires informatiques. il suffit de les convertir en représentation binaires.

I.1.7 Questions

1. Quel est le plus grand nombre codable dans un octet (8 bits) ?
2. Dans 4 octets ? (c'est le cas classique des entiers dans les langages de programmation)
3. Comment faire pour stocker des entiers négatifs ? (réponse : on sacrifie le premier bit d'information pour indiquer le signe). Quel est le plus grand entier positif maintenant stockable dans 32 bits ?

I.1.8 Solutions pratiques

Pour dépasser cette capacité maximale, on a plusieurs solutions, dont par exemple :
— passer aux nombres "flottant", ie en notation scientifique pour les calculatrice.
— implémenter des modules de calculs sur les grands entiers, qui sont bien plus lent que les calculs sur les entiers "natifs".

I.1.9 Codage en pratique des négatifs

On sacrifie un bit pour représenter le signe. En pratique, on utilise pas exactement le premier bit pour le signe mais un représentation plus efficace pour calculer.

I.1.10 Remarque logarithmique

On a vu que sur k bits on peut stocker tous les nombres de 0 à $2^k - 1$ soit exactement 2^k nombres différents. Combien de chiffres décimaux au maximum possèdent ses nombres ? Et si on veut au contraire pouvoir stocker tous les nombres qui ont n chiffres décimaux, combien de bits utiliser au minimum ?

I.2 Nombres réels quelconques

I.2.1 Représentation finie

On ne peut bien évidemment pas retenir en mémoire toutes les décimales d'un nombre réel quelconque (cf $\pi, \sqrt{2}$). On travaille donc avec des *approximations*. Le principe est celui de la notation scientifique des calculatrices : on retient une partie décimale avec un nombre fixé de chiffre et un exposant (qui doit être dans un intervalle d'entier fixé au préalable).

ATTENTION : le séparateur n'est pas la virgule mais le point.

I.2.2 Exemple

On fixe une partie décimale à 5 chiffres dont celui avant la virgule non nul. Alors la représentation de $\sqrt{2}$ est 1.4142×10^0.

Cette représentation est dite normalisée (1 chiffre avant la virgule, un nombre de chiffre total fixé).

I.2.3 Remarque

On doit avoir une règle précise pour arrondir les nombres réels en vue de leur représentation. Pour les machines, ces règles sont fixées par des *normes* telles que IEEE-764

I.2.4 Opérations en virgule flottante

Imaginons que l'on veuille effectuer l'opération $\sqrt{(2)} + e^{10}$.

on a d'abord $\sqrt{2} \approx 1.4142 \times 10^0$ et $e^{10} \approx 2.2026 \times 10^4$

Pour calculer le résultat, on doit d'abord utiliser le même exposant. On prend 4. $\sqrt{2} \approx 0.0001 \times 10^4$ Ainsi $\sqrt{2} + e^{10} \approx 2.2027 \times 10^4$.

Que s'est-il passé au niveau de la précision ? Et si on effectue $(10^7 + 1) - 10^7$ avec ces notations ? Et $(10^7 - 10^7) + 1$?

I.2.5 Conséquences des approximations

Les dernières décimales calculées peuvent être erronées.

On ne peut pas, en général, se fier aux comparaisons (égalité, supérieur, inférieur) entre nombres flottants, car des erreurs d'arrondis successives peuvent fausser le résultat.

I.2.6 Représentation machine

La représentation classique utilise 64 bits pour stocker les nombres flottants et la base utilisée est 2 :

— 1 bit pour le signe,

— 52 bits pour la partie significative (ou décimale, une astuce permet de connaître en fait 53 bits),

— 11 bits pour l'exposant (les valeurs $00000000000_{(2)}$ et $11111111111_{(2)}$ sont réservées pour traiter des cas particuliers).

Quel est l'intervalle théorique pour les exposants ? Pour pouvoir utiliser des exposants négatifs, on convient de retrancher 1023 à tous les exposants stockés.

Combien de décimales stocke-t-on dans ce cas ? Quelle est la plus grande puissance de 10 stockable ?

I.2.7 Remarque

Il existe dans cette représentation un plus petit nombre réel strictement positif ! on voit donc bien que deux réels distincts peuvent être considéré égaux, il suffit que leur différence soit plus petite que ce "plus petit réel".

I.2.8 Nombres binaires à virgule

Quel sens donner à $1.0011_{(2)}$? On adapte la définition de la base 10 :

$$3.1416 = 3 + \frac{1}{10} + \frac{4}{100} + \frac{1}{1000} + \frac{6}{10000}$$

Ainsi $1.0011_{(2)} = 1 + \frac{0}{2} + \frac{0}{4} + \frac{1}{8} + \frac{1}{16}$.

Il est ainsi impossible de donner une représentation binaire finie de certains nombres décimaux. Par exemple $0.4 = 0.011001100110011\ldots$.

Même certains nombres qui ont une représentation finie en base 10 seront en fait des valeurs approchées en mémoire.

I.3 Opérations sur les bits

I.3.1 Définition

On définit sur les bits mémoire les opérations logiques "classiques" : NON, ET, OU, XOR.

I.3.2 Remarque

On peut utiliser ces opérations sur les entiers, à condition de les prendre "bit à bit", c'est à dire que l'on effectue l'opération sur le(s) premier(s) bits entre eux, puis sur le(s) deuxième(s)...

I.3.3 Exemple

$\text{NON}(11010_{(2)}) = 00101_{(2)}$.
101100 ET 011011 = 001000.

II Représenter les textes

II.1 Principe de l'ASCII

On code chaque caractère par une suite de bits. Premières représentations : [0, 127]. cf tableau.

II.2 Etendre la table des caractères

Il manque des caractères : unicode et utf-8.

III Représentation d'autres données

III.1 Images et vidéos

Pixels, représentation des couleurs, taille en mémoire.

III.2 Musique

Echantillonage, profondeur (format WAV et assimilés).

III.3 Compression

Utiliser la redondance (flac, zip), utiliser les propriétés des données que l'on compresse (jpeg, mp3).