Représentation des nombres en machine

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Représentation des entiers naturels

Écriture dans une base

Exemple

Considérons un nombre entier strictement positif, par exemple $N = 432$. Alors N s'écrit $N = 4 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$.

Théorème

Soit $N \in \mathbb{N}^*$. Alors $\exists n \in \mathbb{N}^*,\ \exists (a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$ tels que :

- $\forall i \in [\![0, n-1]\!], a_i \in [\![0, 9]\!]$;
- $a_{n-1} \neq 0$;
- $N = a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} + \cdots + a_0 \times 10^0$

$$= \sum_{k=0}^{n-1} a_k \times 10^k$$

De plus, l'entier n et les entiers (a_i) sont uniques.

Écriture dans une base

Généralisation

L'écriture précédente se généralise à une base quelconque.

Théorème

Soit $N \in \mathbb{N}^*$, soit $b \geq 2$. Alors $\exists n \in \mathbb{N}^*,\ \exists (a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$ tels que :

- $\forall i \in [\![0, n-1]\!], a_i \in [\![0, b-1]\!]$;
- $a_{n-1} \neq 0$;
- $N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \cdots + a_0 \times 0^0$
$$= \sum_{k=0}^{n-1} a_k \times b^k$$

De plus, l'entier n et les entiers (a_i) sont uniques.

Notations

Notation

On note un tel entier N dans la base b comme suit :

$$N = \overline{a_{n-1}a_{n-2}\cdots a_1 a_0}^b$$

Exemple

$$\begin{aligned}
17 &= 1\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 &&= \overline{10001}^2\\
&= 1\times 3^2 + 2\times 3^1 + 2\times 3^0 &&= \overline{122}^3\\
&= 1\times 4^2 + 0\times 4^1 + 1\times 4^0 &&= \overline{101}^4\\
&= 3\times 5^1 + 2\times 5^0 &&= \overline{32}^5\\
&= 2\times 6^1 + 5\times 6^0 &&= \overline{25}^6\\
&= 2\times 7^1 + 3\times 7^0 &&= \overline{23}^7\\
&= 2\times 8^1 + 1\times 8^0 &&= \overline{21}^8\\
&= 1\times 9^1 + 8\times 9^0 &&= \overline{18}^9
\end{aligned}$$

Cas des bases supérieures à 10 (Hexadécimal)

Base supérieure à 10

Pour représenter des nombres dans une base supérieure à 10, il est nécessaire d'introduire de nouveaux symboles pour exprimer les chiffres entre 10 et $b - 1$.

Exemple : Hexadécimal

Un exemple important en informatique est la base 16, appelée **hexadécimale**. Pour représenter les chiffres manquants, on utilise les lettres de A à F :

lettre	A	B	C	D	E	F
signification	10	11	12	13	14	15

Les bases utilisées au cours de l'histoire

Histoire

- Aujourd'hui, on utilise la base 10 (par ce qu'on a 10 doigts).
- Ça n'a pas toujours été le cas : les égyptiens, les mayas, les babyloniens, mésopotamiens et d'autres ont utilisé les bases 20, 24 et 60.
- Les mésopotamiens n'utilisaient cependant pas 60 symboles différents : chaque ń chiffre ż était lui même codé avec un certain nombre de chevrons (chacun comptant pour 10) et de clous (chacun comptant pour une unité).

Les bases utilisées au cours de l'histoire

En informatique

- En informatique, la base 2 (binaire) apparaît naturellement : le 1 et le 0 correspondent à une tension positive (supérieure à un certain seuil) ou une absence de tension (inférieure à un ce seuil) en un point d'un circuit électrique.
- Comme les écritures en binaire sont plutôt longues (on a vu qu'il fallait 5 chiffres pour représenter 17), l'idée de raccourcir les écritures en utilisant des bases de la forme 2^k a mené à l'**hexadécimal**, et plus marginalement à l'**octal** (base 8).
- On verra qu'il est très facile de passer du binaire à l'hexadécimal (ou à l'octal) et réciproquement.

Division euclidienne

Théorème (division euclidienne)

Soit N et M deux entiers, avec $M > 0$. Alors il existe deux entiers q et r tels que :

- $N = qM + r$;
- $0 \leq r \leq M - 1$.

De plus, le couple (q, r) est unique.

Preuve

Existence

- L'ensemble $E = \{a \in \mathbb{Z} \mid N - Ma \geq 0\}$ est un sous-ensemble de $\mathbb{Z}$.
- Il est non vide, car tout entier inférieur à $\frac{N}{M}$ est dans E.
- De plus, il est bornée supérieurement car tout entier strictement supérieur à $\frac{N}{M}$ n'est pas dans E.
- Ainsi, E possède un plus grand élément, que l'on note q. Posons alors $r = N - Mq \geq 0$.
- Par l'absurde, si on avait $r \geq M$, alors on aurait $q+1 \in E$, ce qui est impossible. Ainsi, l'existence du couple (q, r) est démontrée.

Preuve

Unicité

- Considérons un autre couple (q', r') qui satisfait les mêmes hypothèses.
- On a alors $M(q - q') = r' - r$. Donc $r' - r$ est un multiple de M.
- De plus, $-(M-1) \leq r' - r \leq M - 1$, donc $r' - r = 0$.
- Ainsi, $r = r'$, puis $q = q'$.

Preuve : écriture dans une base

Théorème

Soit $N \in \mathbb{N}^*$, soit $b \geq 2$.

Alors $\exists n \in \mathbb{N}^*,\ \exists (a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$ tels que :

- $\forall i \in [\![0, n-1]\!], a_i \in [\![0, b-1]\!]$;
- $a_{n-1} \neq 0$;
- $N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \cdots + a_0 \times 0^0$
$$= \sum_{k=0}^{n-1} a_k \times b^k$$

De plus, l'entier n et les entiers (a_i) sont uniques.

Preuve

On montre maintenant par récurrence le théorème ci-dessus.

On pose $P(N)$ la propriété "N admet une écriture comme dans le théorème ci-dessus, et elle est unique".

Preuve : écriture dans une base

Initialisation

$P(1), \ldots, P(b-1)$ sont vraies car pour $N \in [\![1, b-1]\!]$:

- L'écriture $N = \overline{N}^b$ convient.
- De plus, si $N = \overline{a_{n-1}a_{n-2}\cdots a_1 a_0}^b$ est une autre écriture, comme $a_{n-1} > 0$, nécessairement $n = 1$ car $N < b$.

Ainsi, $N = \overline{N}^b$ est la seule écriture convenable.

Hérédité

- Soit $N \geq b$, et supposons $P(M)$ pour tout $M \in [\![1, N-1]\!]$.
- Soit (q, r) le quotient et le reste de la division euclidienne de N par b.
- Puisque $N \geq b$, on a $0 < q < N$. On peut donc lui appliquer l'hypothèse de récurrence.

Preuve : écriture dans une base

Hérédité

$\exists p \geq 1$ tel que $q = \overline{c_{p-1} \cdots c_0}^b$, avec $c_i \in [\![0, b-1]\!]$ et $c_{p-1} \neq 0$.
Alors, en posant $n = p + 1$, on a :

$$\begin{aligned} N = bq + r &= b \times \left(\sum_{i=0}^{p-1} c_i b^i \right) + r \\ &= \left(\sum_{k=1}^{n-1} c_{k-1} b^k \right) + r \\ &= \overline{c_{n-2} c_{n-3} \cdots c_0 r}^b \end{aligned}$$

- On voit que $(a_{n-1}, \ldots, a_0) = (c_{n-2}, \ldots, c_0, r)$ vérifie les conditions du théorème.

Preuve : écriture dans une base

Hérédité

- De plus, cette écriture est unique :
  - $\hookrightarrow$ le dernier chiffre de N est nécessairement r,
  - $\hookrightarrow$ et et les autres chiffres sont donnés par l'écriture de q qui est unique par hypothèse de récurrence.

Ainsi, $P(N)$ est vraie.

Conclusion

Par principe de récurrence, $P(N)$ est vraie pour tout $N \geq 1$: l'existence et l'unicité de l'écriture dans la base b sont démontrées.

Changement de base

Remarque

En s'inspirant de la preuve précédente, on peut élaborer un algorithme de changement de base.

Exemple

- L'entier 1345 (en base 10), s'écrit 10101000001 en binaire et 541 en hexadécimal.
- Pour pouvoir passer d'une base à une autre, il est nécessaire de savoir calculer dans la base de départ, ou bien dans la base d'arrivée. On va voir deux algorithmes correspondant à ces deux situations.
- On va également voir qu'il est facile de passer du binaire à l'hexadécimal, et réciproquement.

Si on sait calculer dans la base de départ

Principe

On effectue des divisions euclidiennes tant qu'on ne tombe pas sur un quotient nul.

La suite des restes fournit les chiffres de l'écriture de N dans la base b, du moins significatif au plus significatif (i.e. dans l'ordre inverse).

Si on sait calculer dans la base de départ

base.h

```c
int *change_base(int n, int b);
```

base.c

```c
#include <stdlib.h>
#include <math.h>

int *change_base(int n, int b)
{
  int nb = (int)(log(n) / log(b) + 1); // nb de chiffres de n en base b
  int *tab = (int*)malloc(nb * sizeof(int));
  int m = n;
  int r;
  int i = nb - 1;
  while (m != 0)
  {
    r = m%b; // reste de la division euclidienne
    m = m/b; // quotient de la division euclidienne
    tab[i] = r; // r est le prochain chiffre
    i--;
  }
  return tab;
}
```

Si on sait calculer dans la base de départ

exemple.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "base.h"

int main()
{
  int n = 1345, b = 16;
  int *tab = change_base(n, b);
  int nb = (int)(log(n) / log(b) + 1); // nb de chiffres de n en base b
  for (int i = 0; i < nb; i++)
  {
    printf("%d", tab[i]);
  }
  printf("\n");
  free(tab);
  return EXIT_SUCCESS;
}
```

Console

```
janson@mp2i:~$ gcc exemple.c base.c -o test -lm
janson@mp2i:~$ ./test
541
```

Exemples

Exemple

Reprenons l'exemple du nombre 1345 que l'on veut convertir en base 16. On effectue les divisions euclidiennes successives :

$$1345 = 16 \times 84 + 1$$
$$84 = 16 \times 5 + 4$$
$$5 = 16 \times 0 + 5$$

On a donc bien $1345 = \overline{541}^{16}$.

Quelques remarques

Remarques

- En pratique, il est très courant de représenter un nombre dans une base b par la donnée de ses chiffres du **moins significatif** au **plus significatif** : par exemple $1345 = \overline{541}^{16}$ peut se représenter en hexadécimal par le tableau `{1, 4, 5}`.
- Cette représentation est assez pratique, car à un tableau de chiffres `{a_0, a_1, ..., a_p}` est associée le nombre $\sum_{k=0}^{p} a_k b^k$ en base b.
- On appelle les deux représentations possibles **big-endian** (les chiffres les plus significatifs en premier, on commence par "le gros bout") et **little-endian** (on commence par les moins significatifs, soit le "petit bout").

Quelques remarques

Remarques

- L'algorithme précédent donne la représentation big-endian, pour obtenir la représentation en little-endian, il suffit de commencer la boucle avec `i = 0` et d'incrémenter `i` à chaque passage dans la boucle.
- En interne sur un ordinateur, la représentation utilisée dépend du système d'exploitation.
- Les deux sont utilisées, mais la représentation little-endian est la plus répandue.

Si on sait calculer dans la base d'arrivée

Calcul dans la base d'arrivée

On suppose ici que l'on sait faire des additions et des multiplications dans la base d'arrivée, que l'on pourra voir comme la base 10.

Ainsi, pour évaluer $N = \overline{a_{n-1} \cdots a_0}^b = \sum_{k=0}^{n-1} a_k b^k$, il suffit d'évaluer les puissances de b dans la base d'arrivée, et de calculer la somme des $a_k b^k$.

En supposant que les valeurs de 1 et b dans la base d'arrivée sont donnés sans calcul, cela nous fait $2n-3$ multiplications :

- $n-2$ pour calculer les b^k,
- et $n-1$ pour multiplier chaque couple (a_k, b^k), la multiplication $a_0 \times 1$ étant gratuite.

Si on sait calculer dans la base d'arrivée

Algorithme de Hörner

Une méthode plus efficace pour calculer cette somme est l'**algorithme de Hörner**, qui repose sur l'égalité suivante :

$$N = \sum_{k=0}^{n-1} a_k b^k = a_0 + b \times (a_1 + b \times (a_2 + b \times (a_3 + \cdots b \times a_{n-1}) \cdots))$$

Ainsi, si L est une liste contenant les entiers $a_{n-1}, \ldots, a_0$ dans cet ordre, on obtient l'algorithme suivant.

Si on sait calculer dans la base d'arrivée

base.h

```c
int horner(int *tab, size_t nb, int b);
```

base.c

```c
int horner(int *tab, size_t nb, int b)
{
  int res = 0;
  int i;
  for (i = 0; i < nb; i++)
  {
    res = b*res + tab[i];
  }
  return res;
}
```

exemple.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "base.h"

int main()
{
  int tab[] = {5, 4, 1}, b = 16;
  printf("%d\n", horner(tab, 3, b));
  return EXIT_SUCCESS;
}
```

Console

```
janson@mp2i:~$ gcc exemple.c base.c -o test -lm
janson@mp2i:~$ ./test
1345
```

Exemple

Exemple

Convertissons $N = \overline{6ABC}^{16}$ en base 10 :

$$\begin{aligned} N &= 12 + 16 \times (11 + 16 \times (10 + 16 \times 6)) \\ &= 12 + 16 \times (11 + 16 \times 106) \\ &= 12 + 16 \times 1707 \\ &= 27324 \end{aligned}$$

Cas particulier : l'une des bases est une puissance de l'autre

Cas particulier

On a dit précédemment qu'il était facile de passer du binaire à l'hexadécimal et réciproquement.

En fait, c'est le cas si l'une des bases est b et l'autre b^l, pour un certain $l > 1$.

Cas particulier : l'une des bases est une puissance de l'autre

Exemple

$N = 27324$ s'écrit $\overline{6ABC}^{16}$ mais aussi $\overline{110101010111100}^{2}$.

Comme $16 = 2^4$, il suffit d'écrire la correspondance entre les 16 chiffres hexadécimaux et les chaînes de 4 chiffres en binaire (complétés par des zéros à gauche).

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

Cas général

Il suffit d'établir une correspondance entre les paquets de l chiffres dans la base b et les chiffres dans la base $B = b^l$.

En effet, soit $N = \sum_{k=0}^{n-1} a_k b^k$ un nombre exprimé dans la base b, avec $a_k \in [\![0, b-1]\!]$. Quitte à ajouter des zéros au début de la représentation en base b, on peut supposer que n est un multiple de l : $n = l \times m$. Alors :

$$N = \sum_{k=0}^{n-1} a_k b^k = \sum_{i=0}^{m-1} \left(\sum_{j=0}^{l-1} a_{j+il} b^{j+il} \right) = \sum_{i=0}^{m-1} B^i \underbrace{\sum_{j=0}^{l-1} a_{j+il} b^j}_{A_i}$$

Comme $0 \leq a_{j+il} \leq b-1$, on a :

$$0 \leq A_i \leq \sum_{j=0}^{l-1} (b-1) b^j = b^l - 1 = B - 1$$

Les A_i sont donc les chiffres de N dans la base B.

Cas général

Conclusion

Ainsi, on obtient bien l'écriture de N dans la base $B = b^l$ en regroupant les chiffres de N dans la base b par paquets de l à partir de la droite, et en traduisant chaque groupe en le chiffre correspondant dans la base B.

Réciproquement, si on part d'un nombre en base B, il suffit de faire le processus inverse pour retrouver un nombre dans la base b, quitte à supprimer les chiffres nuls à gauche obtenus si le premier chiffre de N dans la base B est strictement inférieur à b^{l-1}.

Représentation des entiers relatifs en binaire

Représentation des entiers relatifs en binaire

Base 2

On se concentre maintenant sur les entiers en base 2.

On sait désormais que tout $N > 0$ s'écrit de manière unique

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

avec $n > 1$, $b_{n-1} = 1$, et $b_i \in \{0, 1\}$ pour $i < n - 1$.

Les entiers (b_k) sont appelés les **bits** de N.

Entiers naturels de taille fixée

En pratique

- En informatique, les entiers sont stockés dans des emplacements mémoire ayant une taille fixée.
- Aujourd'hui, les **registres** d'un microprocesseur ont une taille de 32 ou 64 bits.
- On suppose donc que nos entiers ont une taille fixée (par exemple $n = 64$), et on note toujours $N = \sum_{k=0}^{n-1} b_k 2^k$, mais on ne suppose plus que $b_{n-1} = 1$.
- Ainsi, le plus petit nombre que l'on peut représenter est $\underbrace{\overline{00\cdots0}}_{n}^{2} = 0$, et le plus grand est $\underbrace{\overline{11\cdots1}}_{n}^{2} = 1 = \sum_{k=0}^{n-1} 2^k = \frac{2^n-1}{2-1} = 2^n - 1$.
- Le bit b_0 est appelé le **bit de poids faible**, et b_{n-1} est le **bit de poids fort**.

Additions

L'addition sur entiers naturels se fait comme sur les entiers en base 10 : il suffit de savoir comment additionner deux chiffres et propager les retenues.

C'est particulièrement facile en binaire, puisqu'il n'y a que 2 chiffres. La table d'addition est la suivante :

$+$	0	1
0	0	1
1	1	10

Le 10 signifie que le résultat fait 0, et qu'il faut ajouter un bit de retenue.

Additions

Exemple

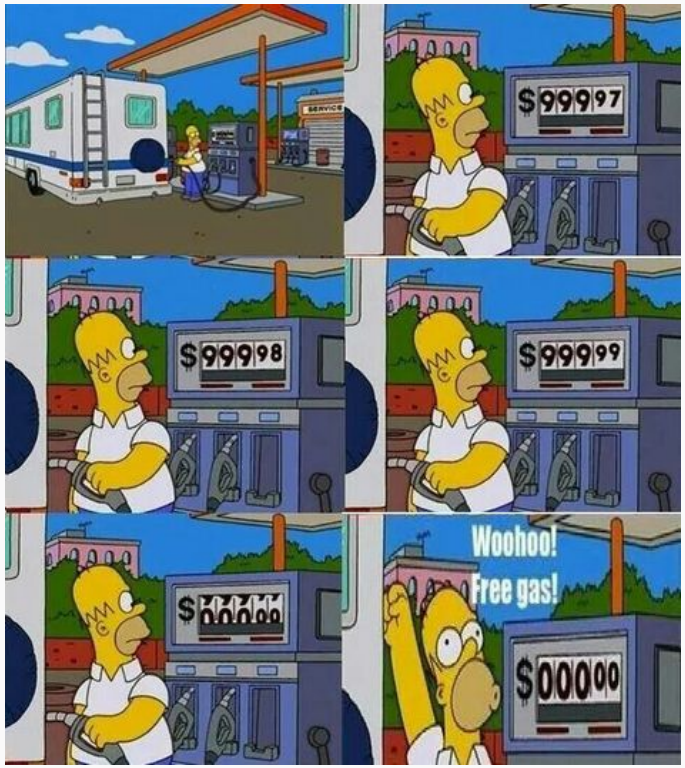
Sur l'exemple suivant, on obtient bien que $37 + 14 = 51$ (les 1 en exposants sont des retenues).

$$\begin{array}{cccccccc} & 1 & 0^{1} & 0^{1} & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

Dépassement de capacité sur les entiers naturels

Dépassement d'entier

Il n'est pas exclu que la dernière addition génère une retenue (on parle de retenue sortante).
Dans ce cas, le résultat de l'addition des deux entiers de n bits ne tient pas sur n bits : on parle alors de **dépassement d'entier** (**integer overflow** en anglais).



Dépassement de capacité sur les entiers naturels

Exemple

Par exemple, sur 8 bits, le 1 en rouge correspond à la retenue issue de l'addition des deux derniers bits :

$$\begin{array}{cccccccccc} & 1 & 0 & 0^{1} & 1^{1} & 0^{1} & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Il faudrait donc 9 bits pour représenter la somme.

En effet, $149 + 142 = 291 > 255 = 2^8 - 1$.

Dépassement de capacité sur les entiers naturels

Ariane 5

- Le vol inaugural de la fusée Ariane 5 a eu lieu le 4 juin 1996.
- Une partie du code du pilote automatique proviennent d'Ariane 4, qui fonctionnait sur 8 bits.
- Cependant, Ariane 5 est 5 fois plus puissante qu'Ariane 4 : 37 secondes après le décollage, un des capteurs mesure une accélération de $300 > 255$, ce qui provoque un dépassement d'entier, et l'explosion de la fusée.

Entiers relatifs

Représentation par valeur absolue

Première idée pour représenter des entiers relatifs sur n bits :

- utiliser le premier bit comme bit de signe ;
- utiliser les autres pour représenter la valeur absolue de l'entier.

Ainsi, on représente l'ensemble $[\![-(2^{n-1}-1), 2^{n-1}-1]\!]$ avec deux zéros.

Exemple

Si le signe de bit 1 est utilisé pour les nombres négatifs, et 0 pour les nombres positifs, on a (sur 6 bits) :

$$\overline{011010}^2 = 26 \quad \text{et} \quad \overline{100001}^2 = -1$$

Entiers relatifs

Problème

- Avec cette représentation, il y a deux zéros : le zéro "positif" $00\cdots0$ et le zéro "négatif" $10\cdots0$.
- Cette représentation ne permet pas de faire des additions facilement.

But

On cherche une représentation des entiers relatifs qui permet d'additionner en faisant exactement la même chose qu'avec des entiers naturels.

Représentation en complément à 2

Représentation en complément à 2

C'est la représentation des entiers relatifs la plus utilisée. Pour $a_{n-1} \cdots a_0$une séquence de n bits, on considère que ce nombre représente :

$$a_{n-1} \cdots a_0 = \begin{cases} \sum_{k=0}^{n-2} a_k 2^k & \text{si } a_{n-1} = 0 \\ -2^{n-1} + \sum_{k=0}^{n-2} a_k 2^k & \text{si } a_{n-1} = 1 \end{cases}$$

Avec cette représentation, on représente tous les entiers de $[\![-2^{n-1}, 2^{n-1} - 1]\!]$ de manière unique.

Représentation en complément à 2

Remarques

- Avec cette représentation, il suffit de regarder le bit de poids fort pour connaître le signe d'un nombre :s'il est nul, le nombre est positif, sinon, il est strictement négatif.
- En reformulant, on représente $N \in [\![-2^{n-1}, 2^{n-1}-1]\!]$ par :

$$\begin{cases} \text{la représentation de } N \text{ sur } n \text{ bits en entier naturel,} \\ \text{si } N \geq 0 \\ \text{la représentation de } 2^n + N = 2^n - |N| \text{ sur } n \text{ bits} \\ \text{en entier naturel, si } N < 0 \end{cases}$$

Entier naturel ou relatif ?

Remarque

Une même suite de bits $a_{n-1} \cdots a_0$ peut donc avoir deux significations différentes.

Savoir si elle représente un entier naturel ou relatif (ou autre chose) ne dépend pas de la mémoire (qui se contente de la stocker), ni du processeur (qui se contente d'effectuer des opérations), mais du programme qui manipule cette suite de bits.

Addition d'entiers relatifs

Addition

S'il n'y a pas de dépassement d'entiers, le résultat de l'addition faite avec cette représentation comme on l'a vu avec les entiers naturels donne le bon résultat (à condition de ne pas tenir compte de la retenue sortante s'il y en a une).

Exemple

$3 + 4 = 7$:

$$\begin{array}{cccccc} & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \end{array}$$

Addition d'entiers relatifs

Addition

S'il n'y a pas de dépassement d'entiers, le résultat de l'addition faite avec cette représentation comme on l'a vu avec les entiers naturels donne le bon résultat (à condition de ne pas tenir compte de la retenue sortante s'il y en a une).

Exemple

$-1+6=5$:

$$\begin{array}{cccccc} & & 1^{1} & 1^{1} & 1 & 1 \\ + & & 0 & 1 & 1 & 0 \\ \hline 1 & & 0 & 1 & 0 & 1 \end{array}$$

Addition d'entiers relatifs

Addition

S'il n'y a pas de dépassement d'entiers, le résultat de l'addition faite avec cette représentation comme on l'a vu avec les entiers naturels donne le bon résultat (à condition de ne pas tenir compte de la retenue sortante s'il y en a une).

Exemple

$-2 + -3 = -5$:

$$\begin{array}{cccccc} & & 1^{1} & 1 & 1 & 0 \\ + & & 1 & 1 & 0 & 1 \\ \hline 1 & & 1 & 0 & 1 & 1 \end{array}$$

Addition d'entiers relatifs

Théorème

Soit n un entier strictement positif, et N et M deux nombres entiers appartenant à l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$.

Si $N + M$ appartient lui aussi à cet intervalle, alors la représentation sur entiers relatifs en complément à 2 de $N+M$ se déduit de celles de N et de M par addition sur entiers naturels, en ignorant l'éventuel bit de retenue sortante.

Preuve

On va distinguer les cas suivants si N et M sont positifs ou strictement négatifs.

- Supposons que $N \geq 0$ et $M \geq 0$.
  Alors leur représentation sur entiers relatifs en complément à 2 sur n bits correspond à celle sur entiers naturels, et le résultat de l'addition est celui de $N + M$ comme entier naturel sur n bits. Comme le résultat est supposé appartenir à $[\![-2^{n-1}, 2^{n-1} - 1]\!]$, le bit de poids fort de la somme est zéro, et correspond bien à la représentation sur entiers relatifs en complément à 2 sur n bits de $N + M$.

Preuve

- Supposons $N \geq 0$ et $M < 0$.
  Remarquez que dans ce cas, la somme $N + M$ appartient forcément à l'intervalle $[\![-2^{n-1}, 2^{n-1} - 1]\!]$: il ne peut y avoir dépassement de capacité en additionnant deux nombres de signes contraires.
  L'addition sur entier naturels correspond à l'entier $2^n + N + M$.
  Deux cas sont à distinguer :
  - Si $N + M \geq 0$, alors 2^n correspond à la retenue sortante : si on l'ignore on obtient bien $N + M$ comme entier positif en représentation sur entiers relatifs en complément à 2.
  - Si $N + M < 0$, alors $2^n + N + M$ correspond précisément à la représentation sur entiers relatifs en complément à 2 de $N + M$.

  Dans les deux cas, le résultat est correct.

Preuve

- Si $N < 0$ et $M \geq 0$, le résultat est correct : il suffit de reprendre le raisonnement précédent en échangeant N et M.
- Enfin, si N et M sont tous deux strictement négatifs, l'addition correspond à l'addition sur entiers naturels de $2^n + N + 2^n + M \geq 2^n$.
  Il y a donc nécessairement une retenue sortante et l'ignorer revient à considérer l'entier naturel $2^n + N + M$, qui correspond bien à la représentation sur entiers relatifs en complément à 2 de $N + M$ puisque $N + M$ est strictement négatif.

Quelques remarques

Remarques

- On aurait pu affiner le théorème précédent en montrant de plus qu'il y a **dépassement d'entier**, si et seulement si la retenue sortante et la retenue sur le bit de poids fort sont différentes.
- C'est comme ça que fonctionne l'additionneur d'une **unité arithmétique et logique** d'un processeur : l'additionneur est réalisé en connectant des additionneurs 1 bits, de plus on peut détecter les dépassements de capacité sur entiers naturels (la retenue sortante vaut 1) et sur entiers relatifs (la retenue sortante est différente de la dernière retenue).
- Le programme qui a lancé l'opération peut récupérer ces informations pour éventuellement prendre en compte le dépassement de capacité, par exemple pour avertir l'utilisateur.

En pratique

Dans un ordinateur, on utilise maintenant des registres de 32 ou 64 bits, ce qui autorise la représentation d'entiers relatifs dans les intervalles $[\![-2^{31}, 2^{31}-1]\!]$ ou $[\![-2^{63}, 2^{63}-1]\!]$.

Si le résultat d'un calcul ne rentre pas dans l'intervalle, le résultat est erroné.

En C

```c
#include <stdio.h>

int puissance(int x, int n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return x * puissance(x, n-1);
  }
}

int main()
{
  printf("3^18 = %d\n", puissance(3,18));
  printf("3^19 = %d\n", puissance(3,19));
  printf("3^20 = %d\n", puissance(3,20));
}
```

Console

```
3^18 = 387420489
3^19 = 1162261467
3^20 = -808182895
```

Exemple

Dans un langage de bas niveau comme le C, le type **int** correspond à des entiers relatifs codés sur 32 bits.

On a $3^{19} < 2^{31} - 1 < 3^{20}$, donc le calcul de 3^{20} produit un dépassement d'entier comme ci-contre (et cela ne provoquera pas de message d'erreur).

En Python

Python

- En Python, les entiers sont **non bornés**.
- Par exemple, Python n'a aucun mal à calculer et afficher correctement 4444^{4444}.
- Les longs entiers sont en fait codés par des paquets de bits de longueur fixée, et il peut y avoir un nombre potentiellement infini de paquets (limité par la mémoire, naturellement).
- Tout ceci se fait de manière transparente pour l'utilisateur, on n'aura donc pas à se soucier des dépassements d'entiers en Python.

Sur vos calculatrices

Calculatrice

Selon votre modèle, le nombre de bits maximal est différent, et bien qu'assez élevé, est limité.

Vous pouvez faire le test vous même, par exemple en testant les puissances de 3 et voir si vous trouvez une erreur ou un résultat faux.

Représentation des nombres réels

Représentation des nombres réels en machine

Nombres réels

On va maintenant voir comment exprimer des nombres réels en machine, ce qui sera utile dès qu'on manipulera des quantités physiques (résultats d'une mesure par exemple).

Exemple

Voici quelques constantes physiques célèbres :

- La vitesse de la lumière : $c_0 = 2.99792458 \times 10^8 \,\mathrm{m.\,s^{-1}}$
- Charge élémentaire : $e = 1.602176565 \times 10^{-19} \,\mathrm{A.\,s}$
- Constante gravitationnelle : $G = 6.67384 \times 10^{-11} \,\mathrm{m^3\,.\,kg^{-1}\,.\,s^{-2}}$
- Nombre d'Avogadro : $N_A = 6.02214129 \times 10^{23} \,\mathrm{mol^{-1}}$
- Constante des gaz parfaits : $R_0 = 8.3144621 \,\mathrm{J.\,K^{-1}\,.\,mol^{-1}}$

Représentation des nombres réels en machine

Précision des valeurs

Certaines quantités sont **exactes**, d'autres ont été **mesurées**.

Lorsqu'on fait un calcul en physique, les résultats des mesures ne sont connus qu'avec une certaine précision.

L'important est donc de pouvoir représenter des réels d'ordres de grandeur très différents, en gardant une précision suffisante pour chacun.

La représentation **scientifique** utilisée précédemment s'y prête bien : on garde un certain nombre de **chiffres significatifs**, et on peut représenter des nombres très petits (en valeur absolue) ou très grands en jouant sur l'**exposant** de la puissance de 10, qui peut être négatif ou positif.

Représentation des nombres dyadiques en binaire

Nombres décimaux

Vous connaissez déjà les **nombres décimaux**.

Par exemple : 12.34, 3.14159, ou -5.2.

Ce sont les nombres réels ayant un nombre fini de chiffres après la virgule **en base 10**.

Généralisation

De même que pour l'écriture des entiers, l'écriture des nombres à virgule se généralise à toute base.

Représentation des nombres dyadiques en binaire

Nombres dyadiques

Les **nombres dyadiques** sont les nombres qui s'écrivent comme une somme finie de puissances de 2, ces puissances pouvant être à exposants positifs ou négatifs.

On généralise l'écriture des entiers en base 2 à celle des nombres dyadiques.

Exemple

Le nombre $\overline{101.001}^2$ s'interprète comme :

$$\underbrace{2^2 + 2^0}_{\text{partie entière}} + \underbrace{0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}}_{\text{partie dyadique non entière}}$$

Au final, $\overline{101.001}^2 = 5.125$.

Représentation en machine

De la même manière qu'un nombre décimal en **notation scientifique** en base 10 s'écrit sous la forme :

- d'un signe,
- multiplié par un nombre décimal de l'intervalle $[1, 10[$,
- multiplié par une puissance de 10,

en base 2, un nombre dyadique non nul s'écrit comme :

- un signe,
- multiplié par un nombre à virgule de l'intervalle $[1, 2[$,
- multiplié par une puissance de 2.

C'est sous cette forme que sont représentés les nombres en machine.

Représentation des nombres dyadiques en binaire

Définition

Dans l'écriture signe $\times$ nombre à virgule de $[1, 2[\times 2^{\text{exposant}}$, le nombre à virgule s'appelle la **mantisse**.

Représentation en mémoire

Dans la représentation des nombres en mémoire, un bit est réservé au signe, et on notera m le nombre de bits réservés à la mantisse et e le nombre de bits réservé à l'exposant.

En mémoire, on a donc $1+e+m$ bits consécutifs, comme ceci :

s	E_{e-1}	E_{e-2}	$\cdots$	E_0	M_1	M_2	$\cdots$	M_m

Représentation des nombres dyadiques en binaire

s	E_{e-1}	E_{e-2}	$\cdots$	E_0	M_1	M_2	$\cdots$	M_m

Flottants

Avec un nombre de bits fixés, il n'est pas possible de représenter tous les réels, mais seulement un nombre fini d'entre eux.

Ceux-ci sont appelés **nombres flottants**, ce sont tous des nombres dyadiques. Il y a trois cas à distinguer :

- si les bits $E_0, \ldots, E_{e-1}$ ne sont ni tous nuls ni tous égaux à 1, on parle de flottant **normalisé** (c'est le plus courant) ;
- si $E_0 = \cdots = E_{e-1} = 0$, on parle de flottant **dénormalisé** ;
- le cas $E_0 = \cdots = E_{e-1} = 1$ est utilisé pour représenter les **infinis** et les **NaN**.

Nombres flottants normalisés

s	E_{e-1}	E_{e-2}	$\cdots$	E_0	M_1	M_2	$\cdots$	M_m

On se place pour l'instant dans le cas où les $E_0, \ldots, E_{e-1}$ ne sont ni tous nuls ni tous égaux à 1. Dans ce cas, la suite de bits ci-dessus représente le nombre :

$$x = S \times M \times 2^{E-D}$$

où :

- $S = (-1)^s \in \{\pm 1\}$ est le **signe** de x, représenté par le bit s, avec la convention 0 pour un nombre positif et 1 pour un nombre négatif.

Nombres flottants normalisés

s	E_{e-1}	E_{e-2}	$\cdots$	E_0	M_1	M_2	$\cdots$	M_m

On se place pour l'instant dans le cas où les $E_0, \ldots, E_{e-1}$ ne sont ni tous nuls ni tous égaux à 1. Dans ce cas, la suite de bits ci-dessus représente le nombre :

$$x = S \times M \times 2^{E-D}$$

où :

- M est la **mantisse**. C'est, pour un flottant normalisé, un nombre appartenant à $[1, 2[$. La partie entière (égale à 1) est implicite et non représentée, si bien que les m bits de mantisse s'interprètent en $M = \overline{1, M_1 \cdots M_m}^2 = 1 + \sum_{k=1}^{m} M_i \times 2^{-i}$.

Nombres flottants normalisés

s	E_{e-1}	E_{e-2}	$\cdots$	E_0	M_1	M_2	$\cdots$	M_m

On se place pour l'instant dans le cas où les $E_0, \ldots, E_{e-1}$ ne sont ni tous nuls ni tous égaux à 1. Dans ce cas, la suite de bits ci-dessus représente le nombre :

$$x = S \times M \times 2^{E-D}$$

où :

- $E - D$ est l'**exposant**. Les e bits s'interprètent comme l'entier naturel $E = \overline{E_{e-1} \cdots E_0}^2 = \sum_{k=0}^{e-1} E_k \times 2^k$, appelé **exposant décalé**. Puisque les E_i ne sont ni tous nuls, ni tous égaux à 1, l'exposant décalé $E \in [\![1, 2^e - 2]\!]$. Le décalage D ne dépend que du nombre de bits e, et a pour valeur $D = 2^{e-1} - 1$. Ainsi, $E - D \in [\![-2^{e-1} + 2, 2^{e-1} - 1]\!]$.

Nombres flottants normalisés

En pratique

- Classiquement, on utilise une représentation des flottants en simple précision (32 bits) ou double précision (64 bits).
- Maintenant que les processeurs ont tous 64 bits, c'est plutôt la double précision qui s'impose.
- Notons qu'on trouve également des représentations avec plus de bits, pour une plus grande précision.
- Même si les entiers e et m changent, le principe est toujours le même.

Nombres flottants normalisés

On donne dans le tableau suivant le nombres de bits dévolus à la mantisse et à l'exposant décalé dans les représentations sur 32 et 64 bits :

format	signe	taille e de E	décalage D	taille m de la mantisse
32 bits	1 bit	8 bits	$2^{8-1}-1=127$	23 bits
64 bits	1 bit	11 bits	$2^{11-1}-1=1023$	52 bits

Nombres flottants normalisés

Exemple

Donnons la représentation du nombre 21.625 sur 32 bits. Ce nombre est dyadique, qui s'écrit :

$$1\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}$$

Ainsi, $21.625 = \overline{10101.101}^2 = \overline{1.0101101}^2 \times 2^4$.

- Le signe est positif, le bit correspondant est donc 0.
- Les 23 bits de la mantisse sont obtenus en complétant 0101101 avec des zéros.
- L'exposant décalé E est obtenu en rajoutant à 4 le décalage (127). Ainsi, $E = 131 = \overline{10000011}^2$.

Au final, 21.625 est représenté sur 32 bits par :
$01000001101011010000000000000000$.

Nombres flottants normalisés

Exemple

Inversement, considérons le nombre représenté en 32 bits par $10010011100100000000000000000000$.

- Son bit de signe est 1 : c'est un nombre négatif.
- Son exposant décalé est 00100111, soit 39. En retirant le décalage, on a dont $E - D = -88$.
- Sa mantisse est représentée par $0010\cdots$, soit $\overline{1.001}^2 = 1 + 2^{-3} = 1.125$.

Au final, on obtient le nombre :

$$-1.125 \times 2^{-88} \approx -3.6350710512584224 \times 10^{-27}$$

Exceptions

Nombre dénormalisé $(\star)$

Si $E = 0$, avec la représentation normalisée on aurait un nombre de la forme :

$$S \times \underbrace{\overline{1.\cdots}^2}_{\text{mantisse}} \times 2^{-D}$$

Autrement dit, avec les bits de la mantisse tous nuls, le plus petit nombre positif représentable serait 2^{-D}.

Il est plus intéressant de se rapprocher de zéro. Ainsi, si $E = 0$, on ne suppose plus que la mantisse est $\overline{1.M_1 \cdots M_m}^2$, mais au contraire $\overline{0.M_1 \cdots M_m}^2$.

Nombre dénormalisé $(\star)$

Mais, en faisant ainsi, on crée un fossé entre le plus petit nombre normalisé positif (2^{1-D}), et le plus grand nombre que l'ont peut obtenir avec un exposant nul : $\overline{0.111\cdots}^2 \times 2^{-D}$ est très proche de 2^{-D}.

Pour compenser ceci, on suppose que le décalage pour un nombre normalisé est $D' = D - 1 = 2^{e-1} - 2$.

L'interprétation d'un **nombre dénormalisé** est donc, puisque E est nul :

$$(-1)^s \times \overline{0.M_1 \cdots M_m}^2 \times 2^{1-D}$$

En particulier, si tous les M_i sont nuls, on représente zéro.
Il y a donc deux zéros : l'un positif, l'autre négatif.

Exceptions

Infinis et NaN ($\star$)

Les nombres ayant un exposant décalé $E = 2^e - 1$ sont utilisés pour représenter les **infinis** et les **NaN**.

- **NaN** signifie "*not a number*", et est utilisé pour les calculs produisant des erreurs, par exemple le calcul de $\sqrt{-1}$.
- les **infinis** sont utilisés pour exprimer le fait qu'un calcul dépasse le plus grand nombre représentable par valeur positive ($+\infty$), ou le plus petit par valeur négative ($-\infty$).

La règle est la suivante :

- si les bits de la mantisse sont tous nuls , c'est un infini ($+\infty$ ou $-\infty$ selon le bit de signe) ;
- sinon, c'est un NaN.

Infinis et NaN

Exemple

```c
#include <stdio.h>

int main()
{
  float a = 1.0;
  int i;
  for (i = 0; i < 1000; i++)
  {
    a *= 3.0;
  }
  float b = -a;
  float c = a + b;
  printf("a : %f\n", a);
  printf("b : %f\n", b);
  printf("c : %f\n", c);
}
```

Output

```
a : inf
b : -inf
c : -nan
```

Exemple

En C, les lignes ci-dessus produisent les infinis et NaN.

Arrondis

Problème

En général, un calcul faisant intervenir deux nombres flottants sur n bits ne donne pas un nombre représentable exactement sur n bits.

Exemple

Il suffit de prendre un nombre décimal non dyadique, comme $1/10 = 0.1$. Celui-ci s'écrit $\overline{1.100110011001100\cdots}^{2} \times 2^{-4}$.

La mantisse n'ayant qu'un nombre fini de bits, il est nécessaire de couper ce développement infini.

Ainsi, la représentation de 0.1 par un flottant ne sera qu'une approximation. Elle est obtenue en prenant le flottant le plus proche.

Erreurs d'arrondis

Attention

Le fait que les réels ne soient représentés qu'approximativement fait que les égalités mathématiques ne tiennent plus avec les flottants.

Exemple

```c
#include <stdio.h>

int main()
{
  float a = 0.1, b = 0.0;
  int i;
  for (i = 0; i < 10; i++)
  {
    b += a;
  }
  if (b != 1.0)
  {
    printf("oups !\n");
  }
}
```

Output

```
oups !
```

Exemple

La boucle ci-contre essaye de calculer 10×0.1 en faisant 10 additions, mais les erreurs d'approximation se cumulent, et au final on obtient un résultat très proche de 1, mais qui n'est pas 1.

Erreurs d'arrondis

Exemple

```c
#include <stdio.h>
#include <math.h>

int main()
{
  float a = pow(2.0, 1000);
  if (a == a+1)
  {
    printf("oups !\n");
  }
}
```

Output

```
oups !
```

Exemple

Sur 64 bits, il y a 52 bits de mantisse.

Le plus petit flottant $> 2^{1000}$ est donc : $(1 + 2^{-52}) \times 2^{1000}$.

Ainsi, $2^{1000} + 1$ est indiscernable de 2^{1000}.

Plus exactement, la valeur de $2^{1000} + 1$ est arrondie au flottant le plus proche, à savoir 2^{1000} lui-même.

Erreurs d'arrondis

Exemple

```c
#include <stdio.h>
#include <math.h>

int main()
{
  float a = 1.0, b = pow(2.0, -53), c = 1.0;
  if (a+b-c != a-c+b)
  {
    printf("oups !\n");
  }
}
```

Output

```
oups !
```

Exemple

Les opérations + et - ayant la même priorité, elles sont effectuées de gauche à droite.

Or ici, $1 + 2^{-53}$ est arrondi au flottant le plus proche, à savoir 1 lui-même. Donc **a+b-c** est évalué à zéro.

Par contre, **a-c+b** vaut **b**, car **a** et **c** sont tous deux égaux.

Arrondis et flottants

Remarque

Il ne faut surtout pas croire à la lecture de ces exemples que les résultats obtenus via des opérations sur les nombres à virgule sont complètement faux en informatique.

Néanmoins, il faut être conscient que dans le monde des flottants, les égalités mathématiques ne sont plus vérifiées “qu’à ε près”, à cause des erreurs d’arrondi et de l’impossibilité de représenter de manière exacte les réels.

Arrondis et flottants

En pratique

La leçon à retenir des exemples et la suivante : sauf cas particuliers bien précis, **le test d'égalité entre deux flottants n'est, en général, pas pertinent**.

On se contentera d'un test de la forme $|a-b| < \varepsilon$, où ε est un "petit" flottant dépendant du problème.

Exemple

Par exemple, pour tester si x est un racine de P, on pourra tester si $|P(x)| \leq 2^{-20}$.

Exemple

Exemple

Pour conclure, considérons le code C suivant, qui résout une équation du second degré en donnant les racines réelles.

Exemple

```c
void trinome(float a, float b, float c)
{
  float delta = b*b -4*a*c;
  if (delta < 0)
  {
    printf("Pas de racines !\n");
  }
  else if (delta > 0)
  {
    float r = sqrt(delta);
    float r1 = (-b - r) / (2 * a);
    float r2 = (-b + r) / (2 * a);
    printf("Il y a deux racines : %f et %f\n", r1, r2);
  }
  else
  {
    printf("Il y a une racine double : %f\n", -b/(2*a));
  }
}
```

Exemple

Exemple

Sur le premier exemple ci-dessous, on obtient des valeurs approchées très correctes de $\frac{1 \pm \sqrt{5}}{2}$.

Exemple

```
1  int main()
2  {
3    trinome(1, -1, -1);
4  }
```

Output

```
Il y a deux racines : -0.618034 et 1.618034
```

Exemple

Exemple

Cherchons maintenant les racines du polynôme $x^2 + 2^{-600}x$.

On travaille sur 64 bits, ainsi les coefficients et les racines (0 et -2^{-600}) sont tous représentables de manière exacte par un flottant.

Exemple

```
1  int main()
2  {
3    trinome(1, pow(2,-600), 0);
4  }
```

Output

```
Il y a une racine double : -0.000000
```

Explication

Le discriminant du trinôme, qui est 2^{-1200}, n'est pas représentable sur 64 bits. Il est alors arrondi à zéro.

Ainsi, le programme conclut à l'existence d'une racine double alors qu'il y a deux racines distinctes très proches.

Exemple

Exemple

Voici un dernier exemple, avec le polynôme :

$$(x - 0.1)^2 = x^2 - 0.2x + 0.01$$

Exemple

```
1  int main()
2  {
3    trinome(1, -0.2, 0.01);
4  }
```

Output

```
Il y a deux racines : 0.099969 et 0.100031
```

Explication

Ici, l'erreur est légèrement différente : les coefficients du polynôme ne sont pas représentables exactement, et le discriminant du polynôme "flottant" est non nul.

Ce n'est pas du à une erreur d'arrondi dans l'opération, mais à une erreur d'arrondi dès le départ.

On renvoie donc deux racines distinctes, très proches de 0.1.