



Chap 07 · Représentation des nombres

Prise de note par Léo BERNARD en MP2I au lycée du Parc.

- **Chap 07 · Représentation des nombres**
 - I. Entiers
 - A. Entiers naturels en base b
 - B. Entiers de taille fixé
 - C. Entiers signés
 - D. Opérandes bit à bit
 - II. Représentation des flottants
 - A. Notation scientifique décimale
 - B. Notation scientifique binaire

I. Entiers

A. Entiers naturels en base b

Théorème: Soit b un entier ≥ 2 .

Tout entier $n \in \mathbb{N}$ peut s'écrire sous la forme:

$$n = \sum_{i=0}^p a_i b^i \text{ avec } a_i \in \llbracket 0, b \rrbracket$$

Si on impose $a_p \neq 0$, alors cette écriture est unique. On note $n = \overline{a_p \dots a_0}_b$.

C'est la décomposition de n en base b .

Remarque: la condition d'unicité pose problème si $n = 0$. Par convention, la décomposition de zéro est vide.

Vocabulaire:

- Les chiffres (i.e. les a_i) qui correspondent au plus grandes puissances de b sont dits plus significatifs (ou de poids fort)
- Les chiffres qui correspondent au plus *petites* puissances de b sont dits *moins* significatifs (ou de poids *faible*).
- En base 2, les chiffres possibles sont 0 et 1.
On parle de *chiffre binaire* ou *binary digit* ou *bit*.

Remarque: Si on écrit $n = 42$, ça signifie $n = \overline{42}^{10}$.

En informatique, on utilise principalement la *base 2 (dite binaire)*, la *base 16 (dite hexadécimale)* et parfois la *base 8 (dite octale)*.

En hexadécimal, les chiffres sont donnés par les symboles: 0123456789ABCDEF

🐴 En OCaml:

```
# 0b10010 (* binaire *)
- : int = 18

# 0xff      (* hexadécimal *)
- : int = 255
# 0xFF
- : int = 255

# 0o77      (* octal *)
- : int = 64
# 077 (* ⚠️ Un entier préfixé d'un 0 est octal ! *)
- : int = 64
```

📦 En C:

```
int n = 0b10010; // Pas dans la norme C. Dépend du compilateur. Donc à éviter !

int n = 0xff;

int n = 077;
```

Division euclidienne:

Soit $b \neq 0$.

Pour tout $a \in \mathbb{N}$, il existe un unique couple $(q, r) \in \mathbb{N}^2$

Tel que: $a = bq + r$ et $r \in [0, b]$.

🛠️ Preuve: par récurrence forte.

Initialisation: Pour $n = 0$, la représentation est vide par convention.

Hérédité: On prend $n > 0$ et on suppose l'existence et unicité de la décomposition pour tout $k < n$.

On divise avec la division euclidienne n par b :

$$n = bq + r, \text{ avec } r \in [0, b[$$

Existence: comme $b > 1$, on a $q < n$.

Par hypothèse de récurrence, on peut écrire $q = \sum_{i=0}^p a_i b^i$

Donc:

$$n = b \sum_{i=0}^p a_i b^i + r$$

$$n = \sum_{i=0}^p a_i b^{i+1} + r$$

$$n = \sum_{i=0}^{p+1} a_{i-1} b^i + r$$

$$n = \sum_{i=0}^{p+1} \alpha_i b^i \text{ avec } \begin{cases} \alpha_0 &= r \\ \alpha_i &= \alpha_i - 1 \end{cases}$$

On donc bien l'existence de la décomposition.

Unicité: On suppose que n possède deux décompositions:

$n = \sum_{i=0}^p a_i b^i$	$n = \sum_{j=0}^{p'} \alpha_j b^j$
$n = a_0 + b \underbrace{\sum_{i=1}^p a_i b^{i-1}}_{=q \in \mathbb{N}}$	$n = \alpha_0 + b \underbrace{\sum_{j=1}^{p'} \alpha_j b^{j-1}}_{=q' \in \mathbb{N}}$
$n = \underbrace{a_0}_{\in [0, b[} + bq$	$\underbrace{\alpha_0}_{\in [0, b[} + bq'$

Par unicité de la division euclidienne: $\begin{cases} a_s 0 = \alpha_0 \\ q = q' \end{cases}$

or q et q' sont $< n$ donc par hypothèse de récurrence, il y a unicité de la décomposition d'où: $p = p'$ et $a_i = \alpha_i$

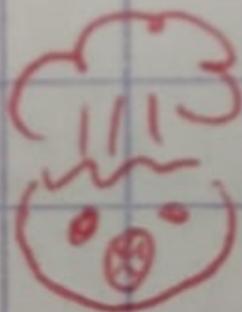
D'où l'unicité pour n .

Conclusion: Pour tout $n \in \mathbb{N}$, on a prouvé l'existence et l'unicité de la décomposition pour n .

Exemple: décomposition de 861 en base 10.

Note: De gauche à droite on a de moins vers plus significatif.

$$\begin{array}{r}
 861 | 110 \\
 \hline
 186 | 110 \\
 \hline
 6 | 110 \\
 \hline
 8 | 110
 \end{array}$$



$$\begin{array}{r}
 861 | 2 \\
 1 | 430 | 2 \\
 0 | 215 | 2 \\
 1 | 107 | 2 \\
 1 | 56 | 2 \\
 0 | 28 | 2 \\
 0 | 14 | 2 \\
 0 | 7 | 2 \\
 1 | 3 | 2 \\
 1 | 1 | 2 \\
 1 | 0
 \end{array}$$

Donc $\overline{861}^{10} = \underbrace{\overline{11}}_3 \underbrace{\overline{0101}}_5 \underbrace{\overline{1101}}_D^2 = \overline{35D}^{16}$

 Exemple: si on a la décomposition en base b et on veut recalculer n :

$$n = \sum_{i=0}^p a_i b^i = a_0 + b(a_1 + b(a_2 + \dots (a_{p-1} + ba_p)))$$

$$= a_0 + ba_1 + b^2a_2 + \dots + b^p a_p$$

$$= a_0 + b(a_1 + ba_2 + \dots + b^{p-1} a_p)$$

C'est l'algorithme de Hörner.

Petit-boutiste et gros-boutiste (small-??? en anglais): les chiffres faibles ou forts d'abord. Pour les humains, c'est plus naturel d'être gros-boutiste, mais pour les ordinateurs, il y a des avantages au petit-boutisme.

Exercices:

1) Écrire un OCaml `eval_msd : int -> int list -> int (*valeur du nombre en base 10*)`

La liste des chiffres commence par les chiffres les plus significatifs (`msd = most significant digit`)

```
let rec eval_msd b li =
  let rec eval_aux li res =
    match li with
    | [] -> res
    | x :: xs -> eval_aux xs (x + b * res) in
      eval_aux li 0
;;
```

2) `eval_lsd`

```
let rec eval_lsd b l =
  match l with
  | [] -> 0
  | x :: xs -> x + b * (eval_lsd b xs)
;;
```

Trouver la décomposition en base b à partir d'un nombre:

Implémentation gros boutiste:

```
let rec decomp_msd n b =
  let rec aux n l =
    if n = 0 then l
    else
      let r = n mod b in
      let q = n / b in
```

```

aux q (r :: l)
in
aux n []
;;

```

```

# decomp_msd 12 2;;
- : int list = [1; 1; 0; 0]

```

Implémentation petit-boutiste:

```

let rec decomp_lsd n b =
  if n = 0 then []
  else (n mod b) :: (decomp_lsd (n/b) b)
;;

```

```

# decomp_lsd 12 2
- : int list = [0; 0; 1; 1]

```

💡 Proposition: Soit $b \geq 2$ un entier

- Un nombre strictement positif s'écrit avec $1 + \lfloor \log_b(n) \rfloor$ chiffres en base b .
- Avec N chiffres en base b , on peut écrire b^N nombres différents $\llbracket 0, b^N \rrbracket$

📚 Exercice: Étant donné un octet 8 bits, donner les valeurs représentables.

- $2^{10} \approx 10^3$.
 - Quel est l'ordre de grandeur du plus grand entier représentable sur 32 bits ? Environ 4 milliards.
 - Sur 64 bits ? 16 milliards de milliards.

📚 Exercice: Effectuer l'addition $\overline{100110}^2 + \overline{1011}^2$ en base 2.

```

1 0 0 1 1 0
+   1 0 1 1
-----
1 1 0 0 0 1

```

Et pour la multiplication:

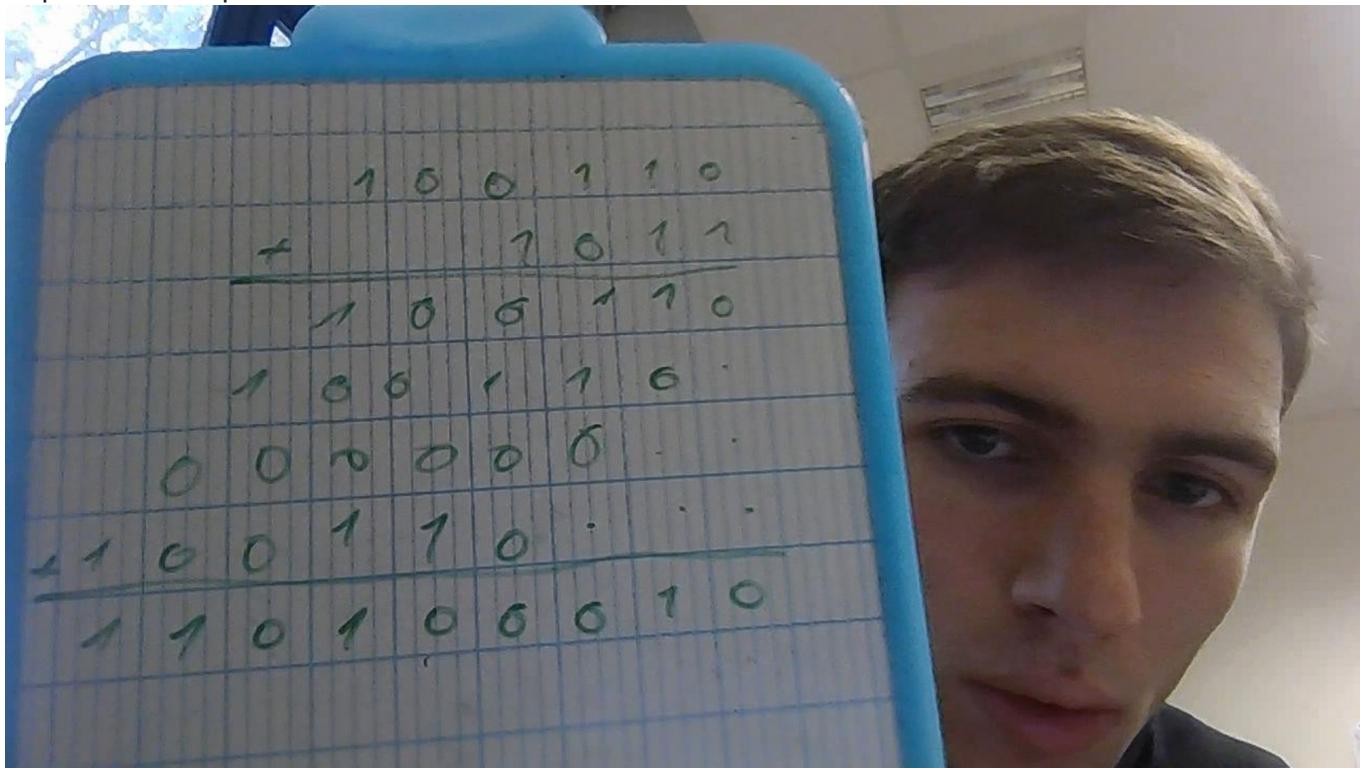


Photo du effectivement effet très BG noé

B. Entiers de taille fixé

En info, entier signifie presque toujours un entier de taille fixée. Chaque machine a une taille maximale d'entiers sur laquelle elle capabe d'opérer naturellement:

- aujourd'hui, presque toujours 64 bits.
- le 32 bits reste encore présent (microcontrôleurs...)

Un language peut proposer des types d'entiers différents en faisant varier:

- la longueur: le nombre de bits
- le caractère signé ou non

🐫 En OCaml, on n'a qu'un type d'entiers, le type `int`. Il est:

- signé
- de longueur naturelle moins 1, typiquement 63 bits.

2^{63} valeurs représentables: $[-2^{62}, 2^{62}]$ (Il me semble que c'est ça mais avec un ± 1 qq part)
Les dépassements de capacité: les calculs se font modulo 2^{63} et sont ramenés dans $[\text{min_int}, \text{max_int}]$

📘 En C, dans le header `stdint.h`.

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
<code>int8_t</code>	Signed	8	1	-2^7 which equals -128	$2^7 - 1$ which is equal to 127

<code>uint8_t</code>	Unsigned	8 Bits	1 Bytes	0	Minimum Value	$2^8 - 1$ which equals 255
<code>int16_t</code>	Signed	16	2	-2 ¹⁵	which equals -32,768	$2^{15} - 1$ which equals 32,767

<code>uint16_t</code>	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535	
<code>int32_t</code>	Signed	32	4	-2 ³¹	which equals -2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
<code>uint32_t</code>	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295	
<code>int64_t</code>	Signed	64	8	-2 ⁶³	which equals -9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,808
<code>uint64_t</code>	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709	

Anciens types:

- un certain nombre de fonctions renvoient un `int` ou un `long int`
- quand on définit un littéral, c'est un `int`.

type	siginification usuelle
<code>unsigned int</code>	<code>uint32_t</code>
<code>int</code>	<code>int32_t</code>
<code>long int</code>	<code>int32_t</code> OU <code>int64_t</code>
<code>unsigned long int</code>	<code>uint32_t</code> OU <code>uint64_t</code>
<code>long long int</code>	<code>int64_t</code>
<code>unsigned long long int</code>	<code>uint64_t</code>

C. Entiers signés

Naïvement:

- le bit le plus significatif (*pour une taille fixée*) code le signe: 1 = positif, 0 = négatif
- la valeur absolue du nombre est codée sur les autres bits de manière usuelle.

 Exemple: `1000 1100 = +12` ; `0000 1100 = -12`

 Deux inconvénients:

- 2 représentations pour 0.
- 2 opérations arithmétiques sont moins naturelles: par exemple, il faut distinguer les cas suivant les signes.

 Exemple du complément en base 10:

$$\begin{array}{r} \dots 0 0 1 2 \\ \dots 9 9 8 8 \\ \hline \dots 0 0 0 0 \end{array}$$

Donc on peut dire en quelque sorte que $-12 = \dots 9988$.

Pour trouver le complément à 10, on effectue la transformation suivante:

$$\begin{array}{r} \dots 0 0 0 1 2 \\ | \\ | \text{ Complément à 9} \\ v \\ \dots 9 9 9 8 7 \\ | \\ | \text{ Complément à 10} \\ | \\ | (+1) \\ v \\ \dots 9 9 9 8 8 \end{array}$$

On effectue la même chose en base 2:

$$\begin{array}{r} \dots 0 0 0 1 1 0 0 \\ | \\ | \text{ Complément à 1} \\ v \\ \dots 1 1 1 0 0 1 1 \\ | \\ | (+1) \\ v \\ \dots 1 1 1 0 1 0 0 \end{array}$$

 **Définition:** On fixe une largeur w .

La valeur en complément à 2 de la suite de bits $(b_{w-1}, b_{w-2}, \dots, b_1, b_0)$ est:

$$b - (w-1)2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

i.e. le bit de poids fort a un poids négatif.

i Remarque: Une même suite de bits $(b_{w-1}, \dots, b_1, b_0)$ peut s'interpréter de 2 manières différentes:

- Manière non signée:

$$v_w(b) = \sum_{i=0}^{w-1} (b_i 2^i)$$

- Manière non signée:

$$vs_w(b) = -b_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} (b_i 2^i)$$

Exemple:

Avec $w = 4$,

- $v_4(0000) = v_4(0000) = 0$
- $vs_4(0100) = v_4(0100) = 4$
- $v_4(1100) = 8 + 4 = 12$ et $vs(1100) = -8 + 4 = -4$
- $v_4(1111) = 8 + 4 + 2 + 1 = 15$ et $vs_4(1111) = -8 + 4 + 2 + 1 = -1$

Exemple:

Avec $w = 8$,

- Le plus grand entier non signé représentable est 255, avec la suite 1111 1111
- $vs_8(0111 1111) = 127$ le plus grand
- $vs_8(1000 0000) = -128$ le plus petit

Proposition:

Pour une largeur w fixée:

- $2^{w-1} - 1$ est le plus *grand* entier représentable.
- -2^{w-1} est le plus *petit* entier représentable.

Proposition:

Pour toute largeur w , pour toute suite de bits b :

$$v_w(b) \equiv vs_w(b) [2^w]$$

Exemple:

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

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \\ + \ 0 \ 0 \ 1 \ 1 \\ \hline \end{array}$$

$$1 \ 1 \ 0 \ 0$$

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \\ + \ 1 \ 0 \ 0 \ 1 \\ \hline \end{array}$$

$$1 \ 0 \ 0 \ 1 \ 0$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \\ + \ 0 \ 0 \ 1 \ 1 \\ \hline \end{array}$$

$$1 \ 0 \ 0 \ 0$$

- Interprétation de ces résultats avec $w = 4$, non signé

$$11 + 7 = 2$$

$$9 + 3 = 12$$

$$9 + 9 = 2$$

$$5 + 3 = 8$$

Cela revient à faire des calculs modulo $2^w = 16$.

- Interprétation de ces résultats avec $w = 4$, signé

$$-5 + 7 = 2$$

$$-7 + 3 = -4$$

$$-7 - 7 = 2$$

$$5 + 3 = -8$$

Les résultats sont ramenés dans $[-2^{w-1}, 2^{w-2} - 1]$ en ajoutant ou retirant 2^w au besoin.
La méthode fonctionne quelque soit le signe des opérandes.

D. Opérandes bit à bit

Il est naturel de définir des opérations bit à bit (*bitwise*) qui opèrent directement sur la représentation binaire des nombres. (Très simple et très rapide pour les processeurs).

Opération	en C	en OCaml	Exemple	Table de vérité
-----------	------	----------	---------	-----------------

Opération	en C	en OCaml	Exemple	Table de vérité																
Conjonction (AND)	&	land	$ \begin{array}{r} 0001101 \\ \& 0101001 \\ \hline 0001001 \end{array} $	<table border="1"> <tr><td>&</td><td> </td><td>0</td><td>1</td></tr> <tr><td></td><td>-</td><td>-</td><td>-</td></tr> <tr><td>0</td><td> </td><td>0</td><td>0</td></tr> <tr><td>1</td><td> </td><td>0</td><td>1</td></tr> </table>	&		0	1		-	-	-	0		0	0	1		0	1
&		0	1																	
	-	-	-																	
0		0	0																	
1		0	1																	
Conjonction (OR)		lor	$ \begin{array}{r} 0001101 \\ 0101001 \\ \hline 0101101 \end{array} $	<table border="1"> <tr><td> </td><td> </td><td>0</td><td>1</td></tr> <tr><td></td><td>-</td><td>-</td><td>-</td></tr> <tr><td>0</td><td> </td><td>0</td><td>1</td></tr> <tr><td>1</td><td> </td><td>1</td><td>1</td></tr> </table>			0	1		-	-	-	0		0	1	1		1	1
		0	1																	
	-	-	-																	
0		0	1																	
1		1	1																	
Ou exclusif (XOR)	^	lxor	$ \begin{array}{r} 0001101 \\ ^ 0101001 \\ \hline 0100100 \end{array} $	<table border="1"> <tr><td>^</td><td> </td><td>0</td><td>1</td></tr> <tr><td></td><td>-</td><td>-</td><td>-</td></tr> <tr><td>0</td><td> </td><td>0</td><td>1</td></tr> <tr><td>1</td><td> </td><td>1</td><td>0</td></tr> </table>	^		0	1		-	-	-	0		0	1	1		1	0
^		0	1																	
	-	-	-																	
0		0	1																	
1		1	0																	
Négation (NOT)	~	lnot	$ \begin{array}{r} ~ 0101001 \\ \hline 1010110 \end{array} $	<table border="1"> <tr><td>~</td><td> </td><td>0</td><td>1</td></tr> <tr><td></td><td>-</td><td>-</td><td>-</td></tr> <tr><td>1</td><td> </td><td>0</td><td>0</td></tr> </table>	~		0	1		-	-	-	1		0	0				
~		0	1																	
	-	-	-																	
1		0	0																	
Décalage à gauche (Left bitshift)	n << i	n lsl i	1001 << 3 = 1001000																	
Décalage à droite (Right bitshift)	n >> i	n lrl i	1001011 >> 3 = 1001																	

Exercice: XOR

On se place dans l'ensemble $\mathbb{U} = \{\text{entiers binaires non signés sur } N \text{ bits}\}$, avec $N \geq 1$ fixé.

L'opération \wedge est une LCI sur \mathbb{U} .

- 1. \wedge est-elle commutative ? associative ?

\wedge	0	1
0	0	1
1	1	0

Commutatif.

x	y	z	$(x \wedge y) \wedge z$	$x \wedge (y \wedge z)$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Associatif.

- 2. \wedge a-t-elle un élément neutre ?

$\underbrace{000\dots 0}_N$ est l'élément neutre.

- 3. Que vaut $x \wedge x$?

$\forall x \in \mathbb{U}, x \wedge x = 0 \dots 0$

x est son propre inverse pour \wedge .

- 4. Que peut-on dire de l'ensemble (\mathbb{U}, \wedge) ?

C'est un groupe abélien (commutatif).

 **Exercice:** On considère un entier n non signé codé sur N bits.

- 1. Comment déterminer si n est pair ?

$n \& 1 == 0$

- 2. Comment obtenir un entier p dont la représentation binaire est $\underbrace{11\dots 1}^k$?

k uns avec $k \leq N$

$(1 \ll k) - 1$

- 3. Comment déterminer si n est divisible par 2^k ?

$n \& (1 \ll k) - 1 == 0$

→ Ceci est un masque

```
n 1 0 0 1 0 0 0
& M 0 0 0 0 1 1 1
```

```
0 0 0 0 0 0 0
^~~~~~ ^~~~
```

forcément 0 Si un bit de n était 1, le résultat du `&` serait 1.

- 4. Comment récupérer le quotient de la division euclidienne de n par 2^k ? Le reste ?

Quotient: $n \gg k$

Reste: $n \& (1\ll k - 1)$

`1 1 0 1 1 0 1 0`

`~~~~~ ^~~~~~ ^~~~~~`

$n \gg k$ $n \& (1\ll k - 1)$

- 5. Comment calculer $i2^k$, où $i2^k \leq n < i2^{k+1}$?

$$i2^k \leq n < i2^{k+1}$$

$$\iff i \leq \frac{n}{2^k} < 2i$$

$\iff i$ est le quotient de la division de n par 2^k

$$i2^k = (n \gg k) \ll k$$

(On efface les k bits de poids faible)

- 6. Comment extraire le $k^{\text{ème}}$ bit de n ? (Le bit de poids 2^k)

$(n \gg k) \& 1$

- 7. Comment extraire les bits de poids $2^k, \dots, 2^{k+l-1}$ (où $1 \leq l \leq N - k$) ?

Exercice: Popcount

On considère un entier n non signé codé sur N bits.

- 1. Comment vérifier si n est une puissance de 2 ?

On considère $n \& (n-1)$.

`0 0 1 0 1 1 1 0`
`& 0 0 1 0 1 1 0 1`

`—————`
`0 0 1 0 1 1 0 0`

`1 0 1 1 1 0 0 0`
`& 1 0 1 1 0 1 1 1`

`—————`
`1 0 1 1 0 0 0 0`

`0 0 1 0 0 0 0 0`
`& 0 0 0 1 1 1 1 1`

`—————`
`0 0 0 0 0 0 0 0`

$n \& (n-1)$ permet d'annuler le bit non nul de poids le plus faible.

- 2. Écrire un algorithme qui calcule le nombre de 1 dans l'écriture binaire de n (complexité indépendante de N , mais linéaire en le résultat).

```

def popcount(n):
    compteur = 0
    while n != 0:
        compteur += 1
        n = n & (n-1)
    return compteur

```

Pour la correction de cet algo: $\text{popcount}(n_i) + c_i = \text{popcount}(n_0)$

II. Représentation des flottants

A. Notation scientifique décimale

Tout réel x non nul peut s'écrire sous la forme:

$$x = (-1)^S \times 10^E \times (d_0 + d_1 10^{-1} + d_2 10^{-2} + \dots)$$

avec
$$\begin{cases} S = 0 \text{ ou } 1 \text{ (signe de } x\text{)} \\ E \text{ entier relatif} \\ d_i \in [0, 9] \text{ et } d_0 \neq 0 \end{cases}$$

On appelle cette notation la notation scientifique décimale.

Exemples:

- 3.141593×10^0
- 2.997925×10^8
- $6.62607015 \times 10^{-34}$
- -6.6742×10^{-11}

B. Notation scientifique binaire

On peut généraliser cette idée en binaire.

Tout nombre x peut s'écrire sous la forme:

$$x = \underbrace{(-1)^S}_{\text{signe}} \times \underbrace{2^E}_{E=\text{exposant}} \times \underbrace{(b_0 + b_1 2^{-1} + b_2 2^{-2} + \dots)}_{\text{mantisse}}$$

avec
$$\begin{cases} S = 0 \text{ ou } 1 \text{ (signe de } x\text{)} \\ E \text{ entier relatif} \\ b_i \in [0, 1] \text{ et } b_0 \neq 0 \end{cases}$$

Toutefois, on ne dispose pas de mémoire infinie. On sera donc contraint à imposer des limites sur la représentation :

- E ne peut prendre ses valeurs que dans un intervalle fixé,
- l'ensemble des b_i doit être fini.

 **Définition :** On appelle la mantisse M la quantité:

$$M = (b_0) + (b_1 \cdot 2^{-1}) + (b_2 \cdot 2^{-2}) + \dots + (b_m \cdot 2^{-m})$$

- Par construction, M appartient à l'intervalle $[1, 2[$.
- b_0 est non nul, et en binaire on n'a qu'un seul tel chiffre : 1. Il est donc inutile de le garder en mémoire. On dira alors que la mantisse est codée sur m bits.

 **Définition :** On note e le nombre de bits sur lequel est codé l'exposant E .

- Interprété comme un *entier non signé*, il permet de définir un entier (l'exposant décalé) :

$$E' \in \llbracket 0, 2^e \rrbracket = \llbracket 0, 2^e - 1 \rrbracket$$

- On retrouve la valeur de E à partir de E' en retranchant $\underline{2^{e-1} - 1}$ à ce dernier. Ainsi :

$$E \in \llbracket -2^{e-1}, 2^{e-1} \rrbracket = \llbracket -2^{e-1} + 1, 2^{e-1} \rrbracket$$

 Attention, ça ne fonctionne pas comme pour la signature classique d'un entier en complément à deux.

 **Exemple :** Avec E codé sur $e = 8$ bits, on a:

La valeur interprétée comme entier non signé:

$$E' \in \llbracket 0, 2^8 - 1 \rrbracket \iff E' \in \llbracket 0, 255 \rrbracket$$

On retrouve E en retranchant $2^{e-1} - 1 = 2^7 - 1 = 127$:

$$\begin{aligned} E &\in \llbracket -2^{e-1}, 2^{e-1} \rrbracket = \llbracket -2^{e-1} + 1, 2^{e-1} \rrbracket \\ &\iff E \in \llbracket -128, 128 \rrbracket = \llbracket -127, 127 \rrbracket \end{aligned}$$

Un réel x non nul est représenté par un code sur $1 + e + m$ bits de la forme :

$$\underbrace{s}_{\text{signe}} \underbrace{c_{e-1}c_{e-2}\dots c_1c_0}_{\text{exposant (sur } e \text{ bits)}} \underbrace{b_1b_2\dots b_m}_{\text{mantisse (sur } m \text{ bits)}}$$

Pour les exposants, on va du bit de poids fort vers le plus faible

On obtient la valeurs de x avec:

$$x = (-1)^s \times 2^{\left(\sum_{k=0}^{e-1} c_k 2^k - 2^{e-1} + 1\right)} \times \left(1 + \sum_{k=1}^m b_k 2^{-k}\right)$$

Ou encore :

$$x = (-1)^s \times 2^{\overline{c_{e-1} \dots c_1 c_0}}^2 - (2^{e-1} - 1) \times \overline{1.b_1 b_2 \dots b_m}$$

En OCaml ou en C (avec le type `double`), les flottants sont codés sur 64 bits avec la répartition suivante :

- 1 bit pour le signe,
- 11 bits pour l'exposant,
- 52 bits pour la mantisse.

Avec 11 bits pour l'exposant, à priori E devrait pouvoir prendre ses valeurs entre -1023 et 1024. En pratique, les valeurs extrêmes sont réservées à des cas particuliers que nous allons détailler ensuite.

Ainsi, $E \in \llbracket -1022, 1023 \rrbracket$.

On représente une valeur nulle avec la convention :

- $E = -1023$,
- Tous les bits de la mantisse sont nuls.

Il y a alors deux possibilités de représenter zéro, suivant le bit de signe ; ce qui permet de distinguer -0 et +0.

D'autres cas particuliers :

- Si $E = -1023$ et que la mantisse est non nulle, on a un nombre dénormalisé, qui vaut :

$$(-1)^s \times 2^{-1022} \times \overline{0.m_1 \dots m_{52}}^2$$

- Si $E = 1024$ et que la mantisse est nulle, on a $+\infty$ ou $-\infty$ suivant le bit de signe.
- Si $E = 1024$ et que la mantisse est non nulle, on a `NaN`.

La valeur théorique de l'exposant est :

$$E = \lfloor \log_2 |x| \rfloor$$

Calcul manuel:

```

E = 0
M = x
while M >= 2:
    E = E + 1
    M = M / 2

```

```
while M < 1:  
    E = E - 1  
    M = M * 2
```

Calcul manuel des b_i pour $M \in [1, 2[$:

```
y = M - 1  
for i = 1->m:  
    if y >= 0.5:  
        b_i = 1  
        y = 2*y - 1  
    else:  
        b_i = 0  
        y = 2*y
```

>>> Voir  Chap 7 · Exos
