Représentation des données numériques

G. Dewaele

Novembre 2017

Déroulement de la présentation

Raison du binaire

Il est aisé de mémoriser/de transmettre des 0 et des 1 :

- trous/absence de trous dans des cartes perforées
- tension/absence de tension sur une ligne électrique
- interrupteur (relais) ouvert/fermé
- lumière/obscurité (cables optiques, télécommandes IR)
- orientation de moments magnétiques (bandes magnétiques, disquettes, disques durs mécaniques)
- mini-condensateurs (mémoire moderne)
- pièges à électrons (mémoire flash)
- cavités dans une couche métallique (CD, DVD)...

Avoir plus de deux états augmente les risques de confusion.

Conséquences

Il faut représenter les objets à manipuler par des suites de 0 et 1 :

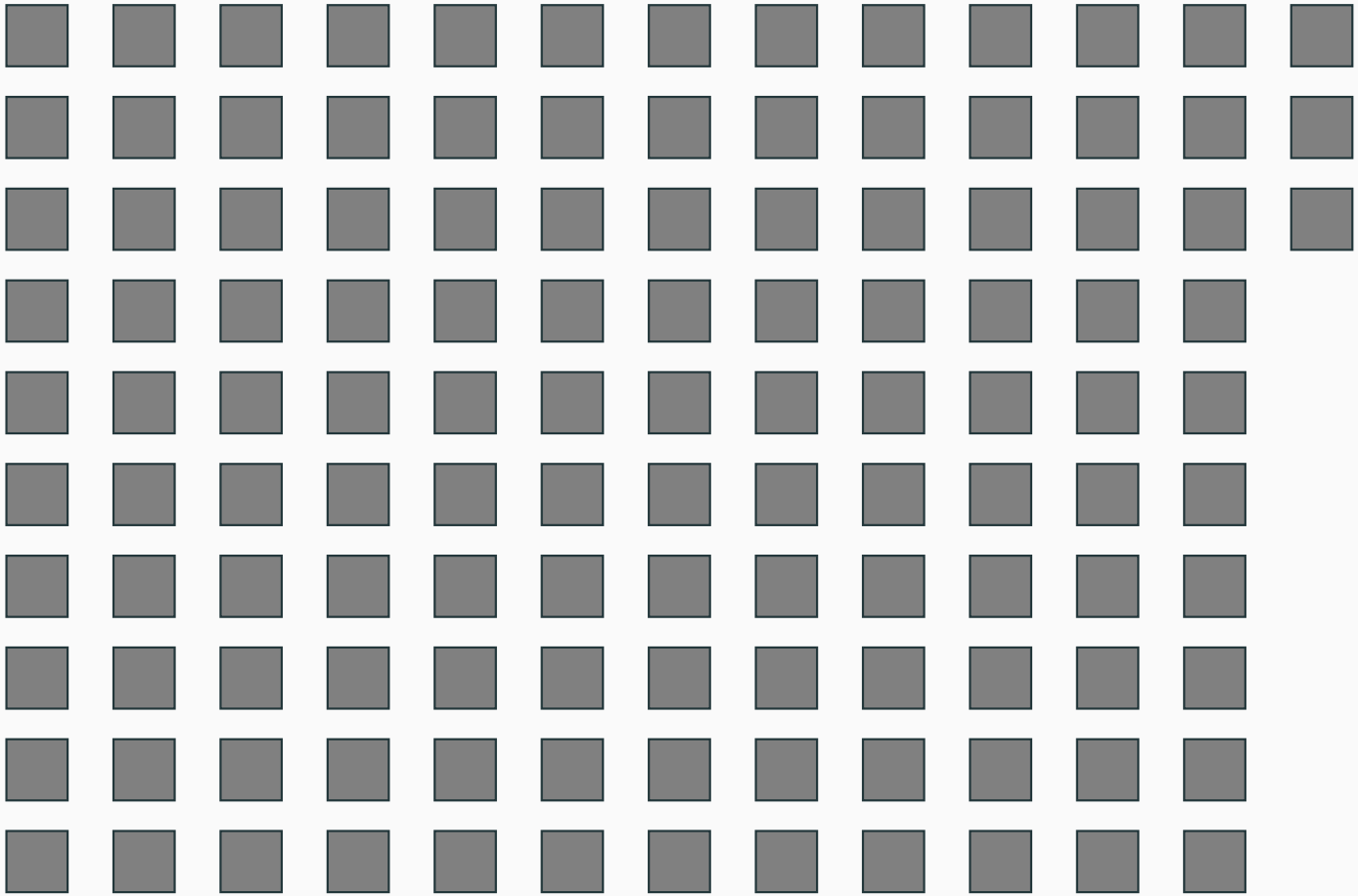
- valeurs numériques,
- caractères et chaînes de caractères,
- images,
- vidéos,
- sons,
- programmes...

On s'intéresse pour le moment à la représentation :

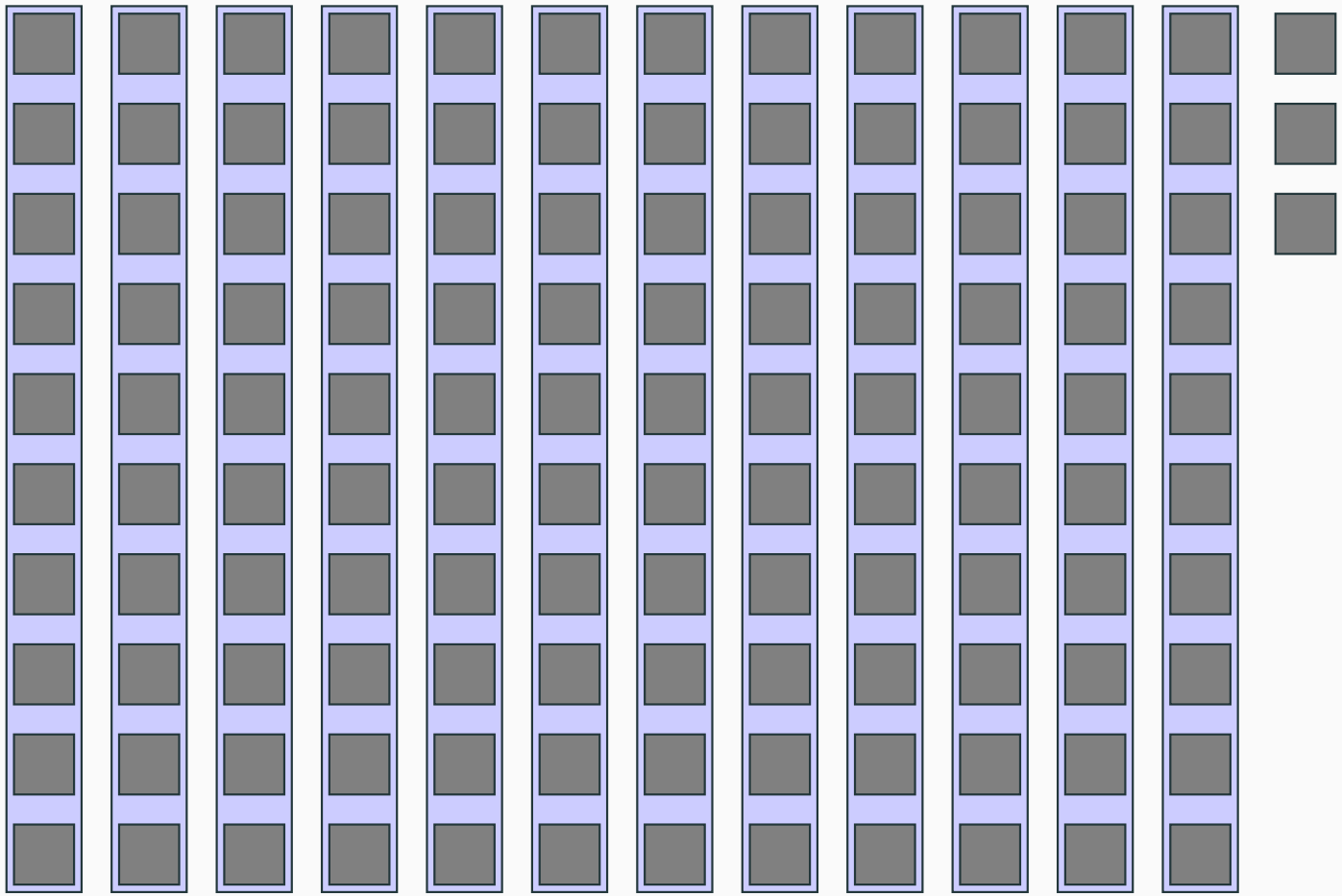
- des entiers ;
- des réels.

Déroulement de la présentation

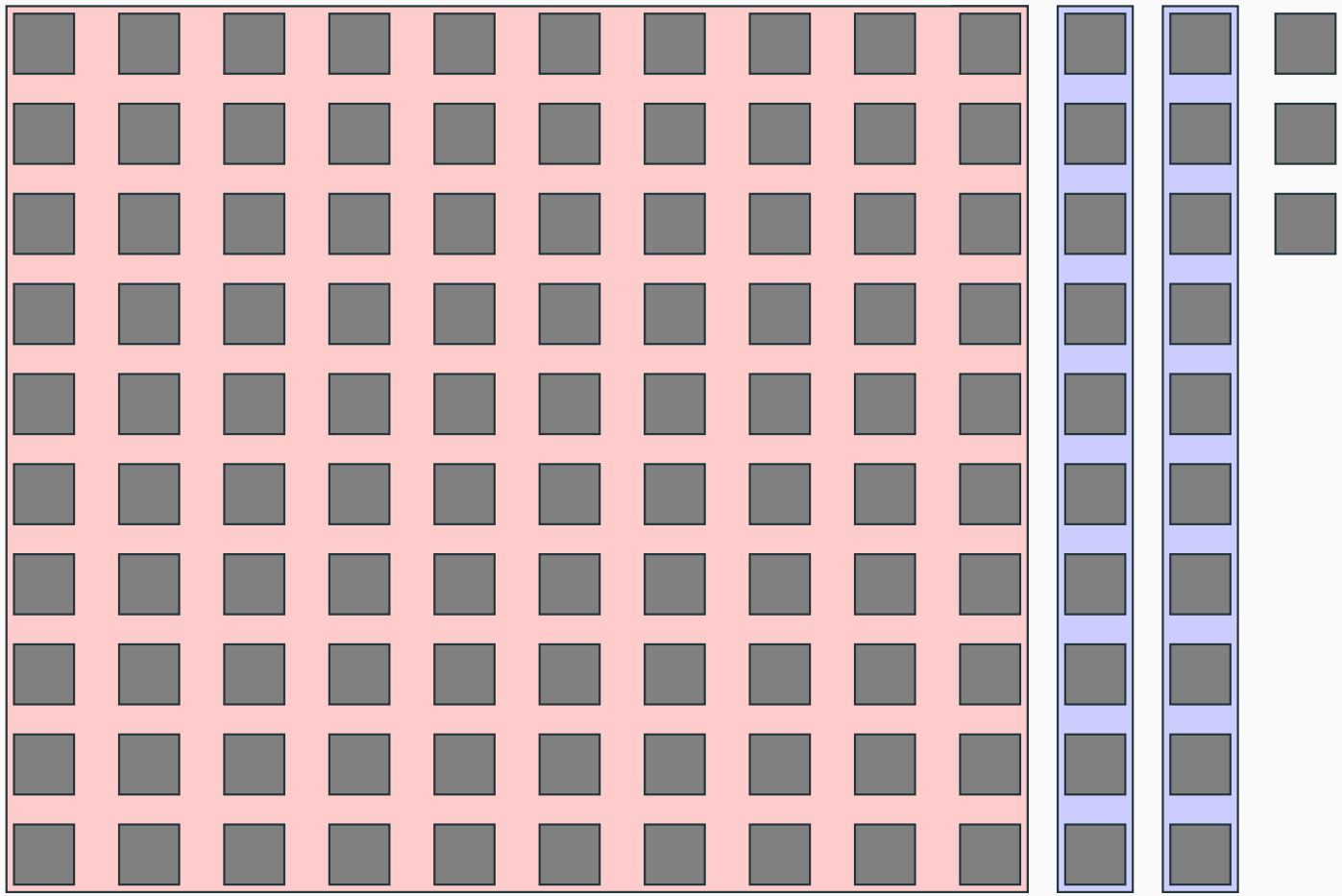
La représentation décimale (base 10)



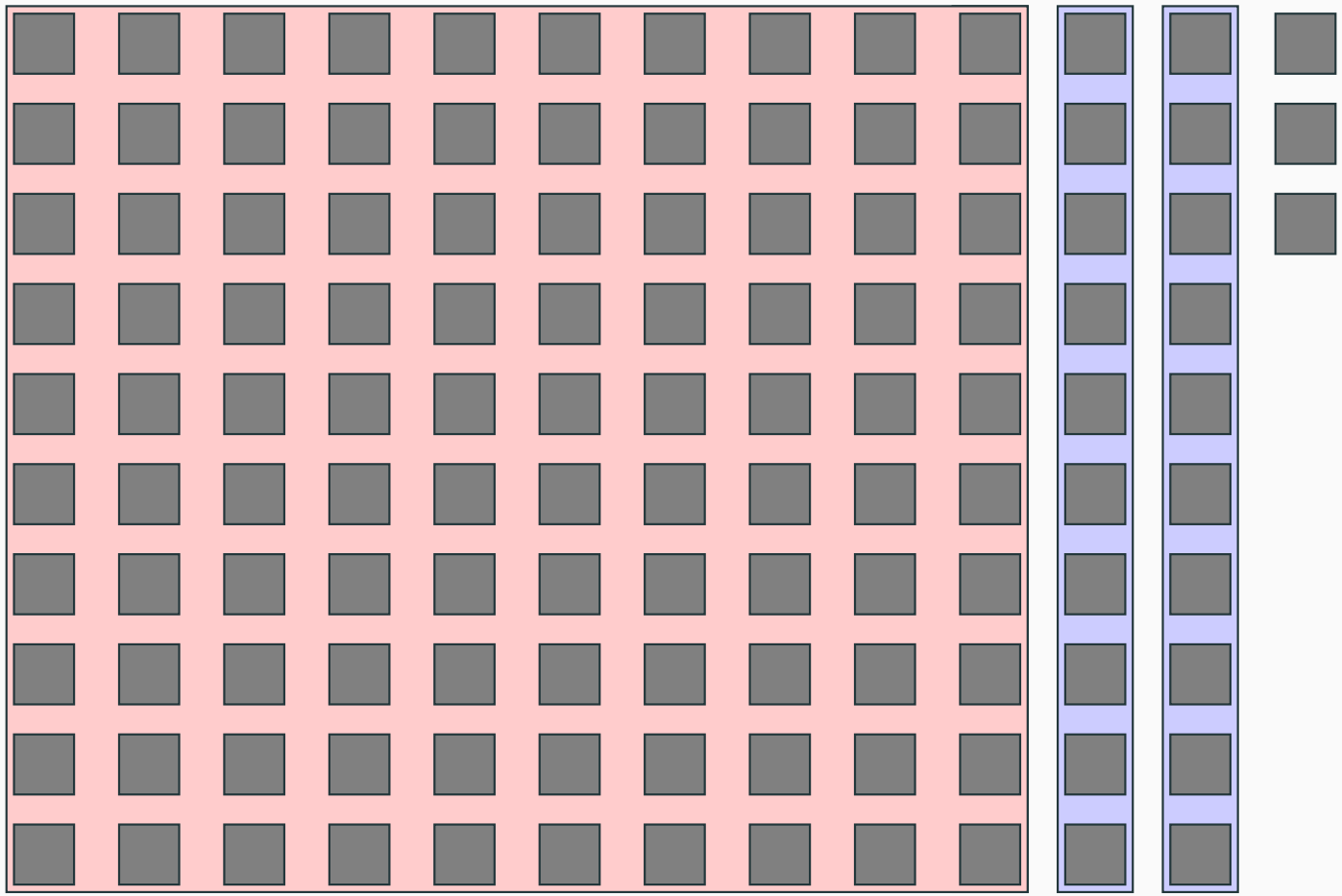
La représentation décimale (base 10)



La représentation décimale (base 10)

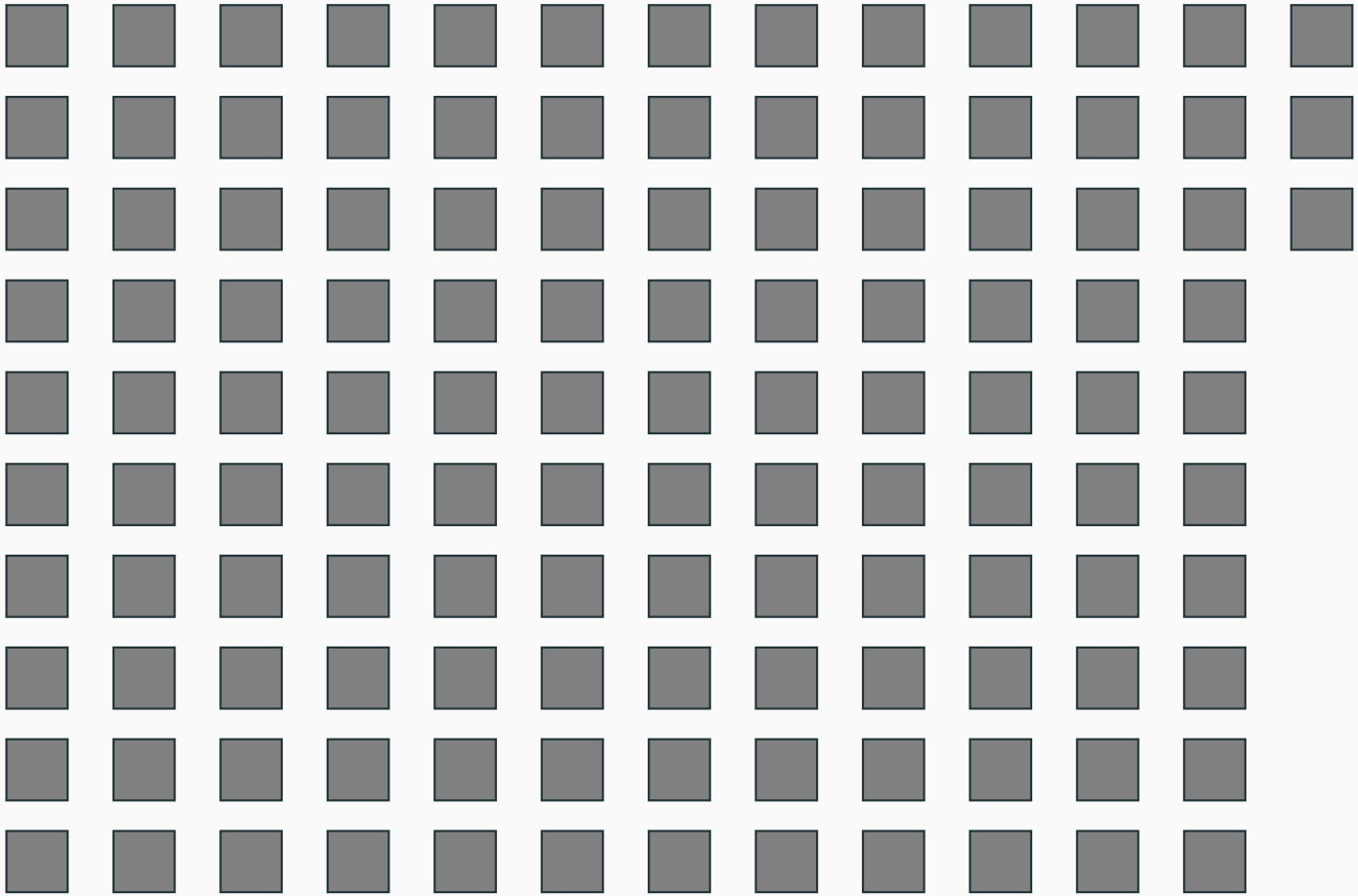


La représentation décimale (base 10)

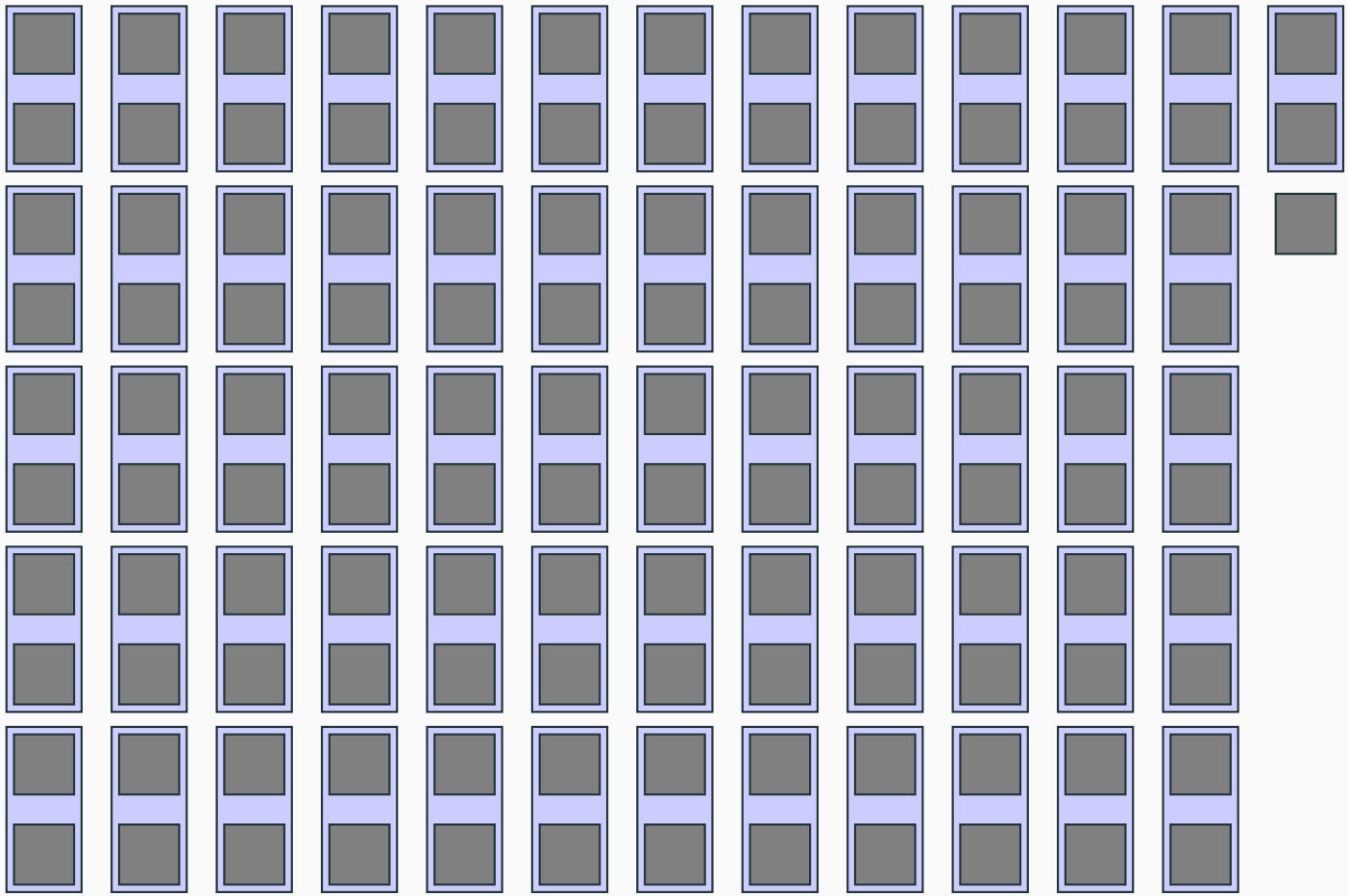


$$1 \times 10^2 + 2 \times 10 + 3 \rightarrow 123$$

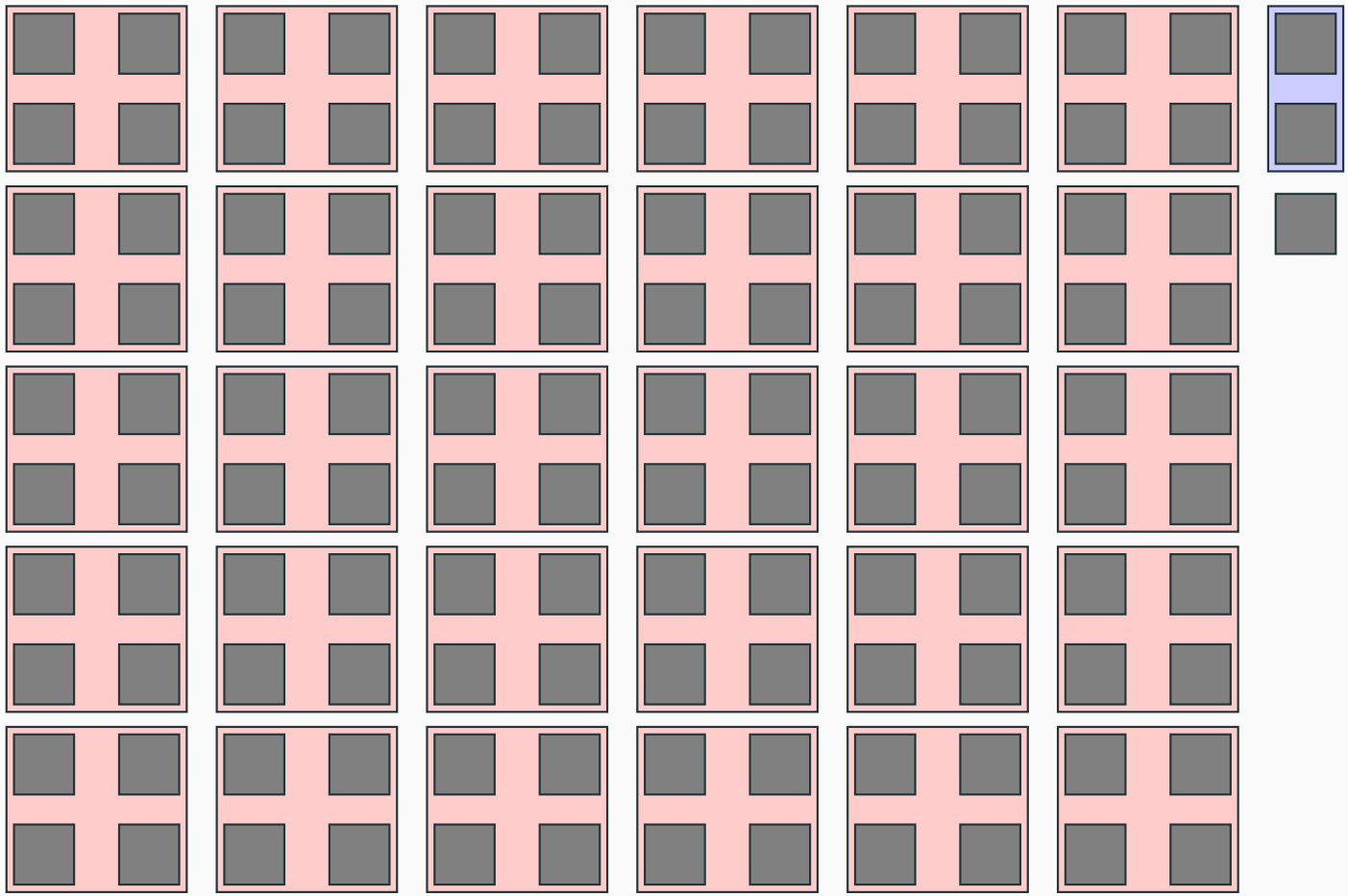
La représentation binaire (base 2)



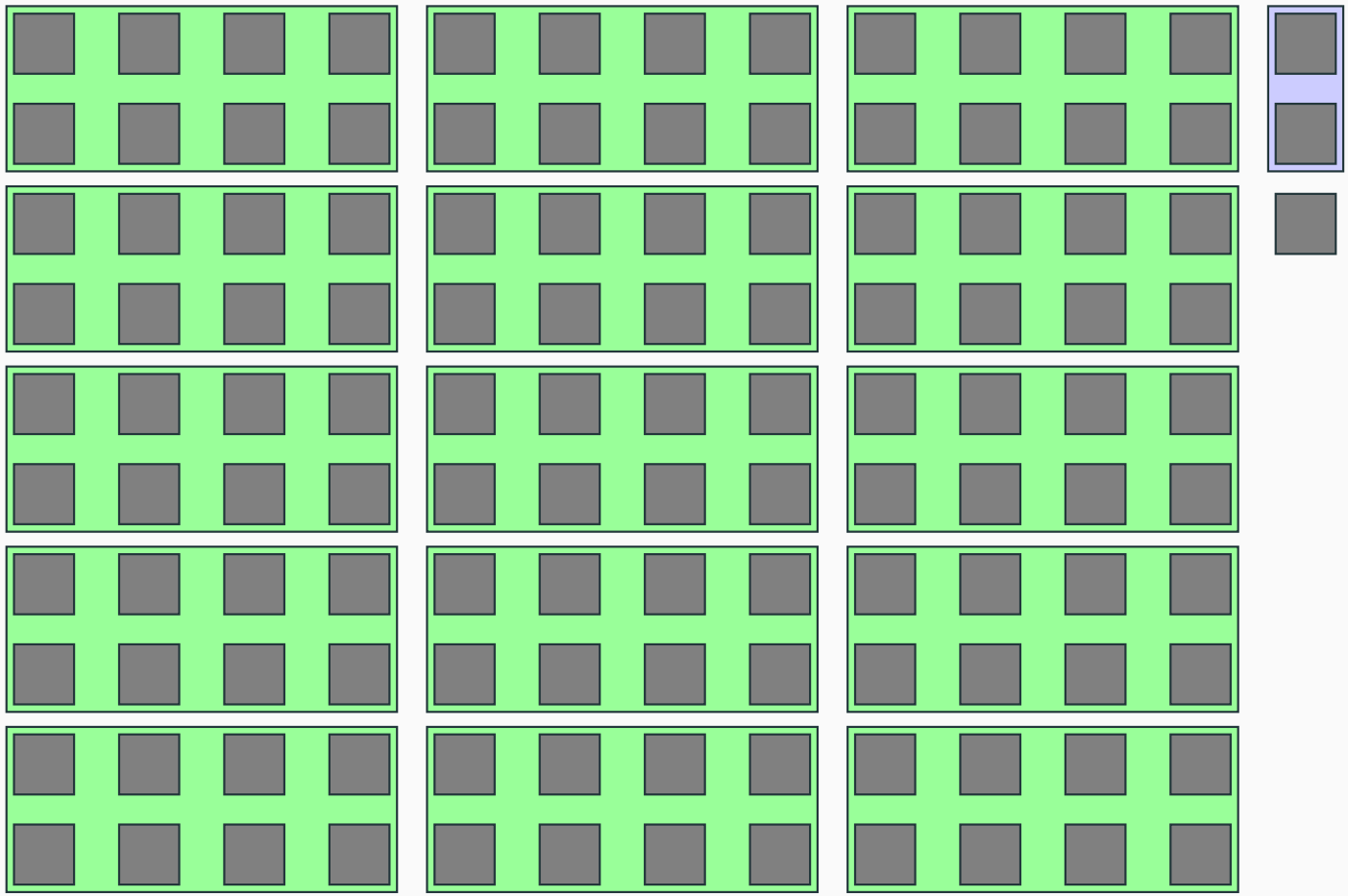
La représentation binaire (base 2)



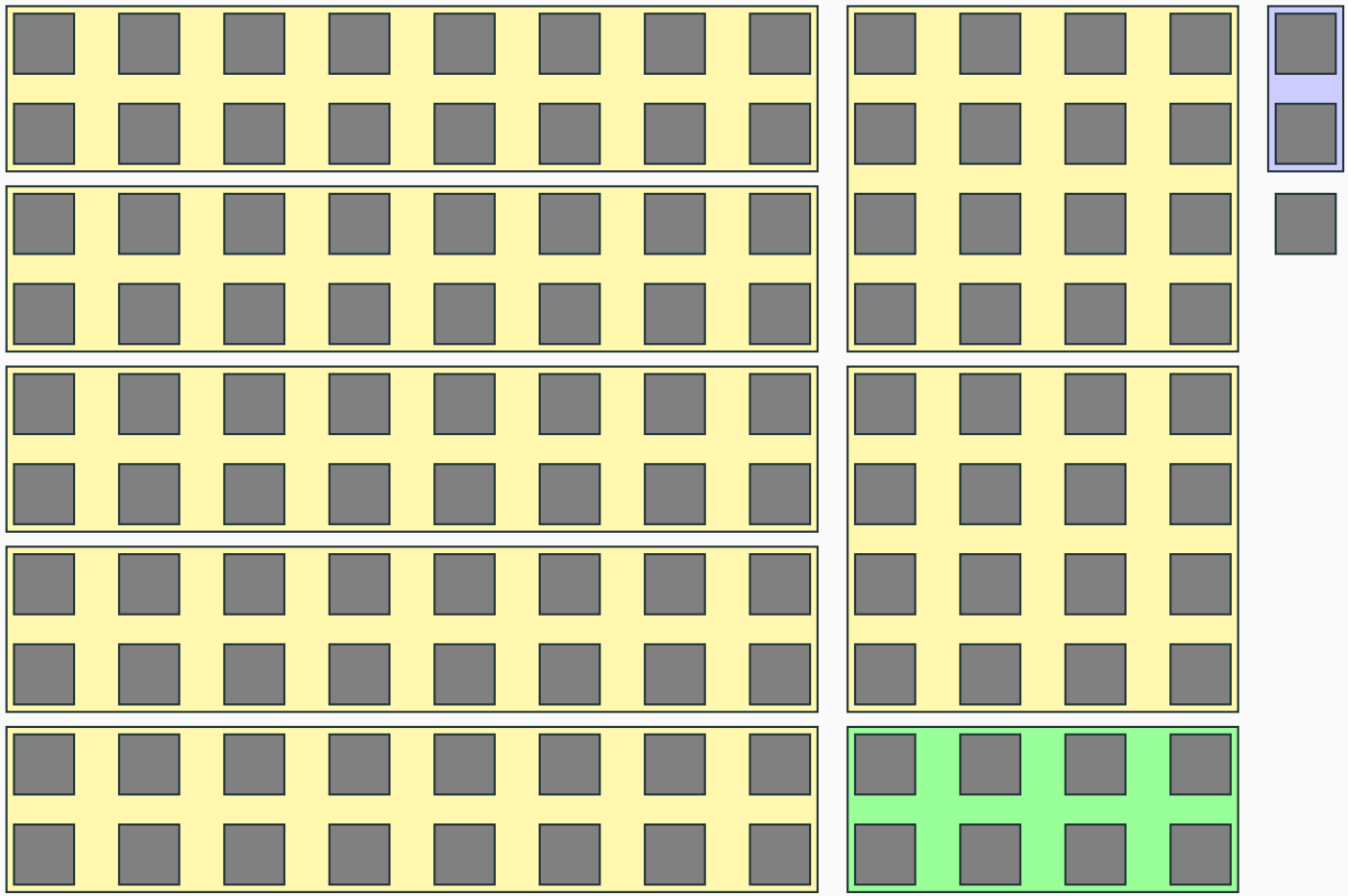
La représentation binaire (base 2)



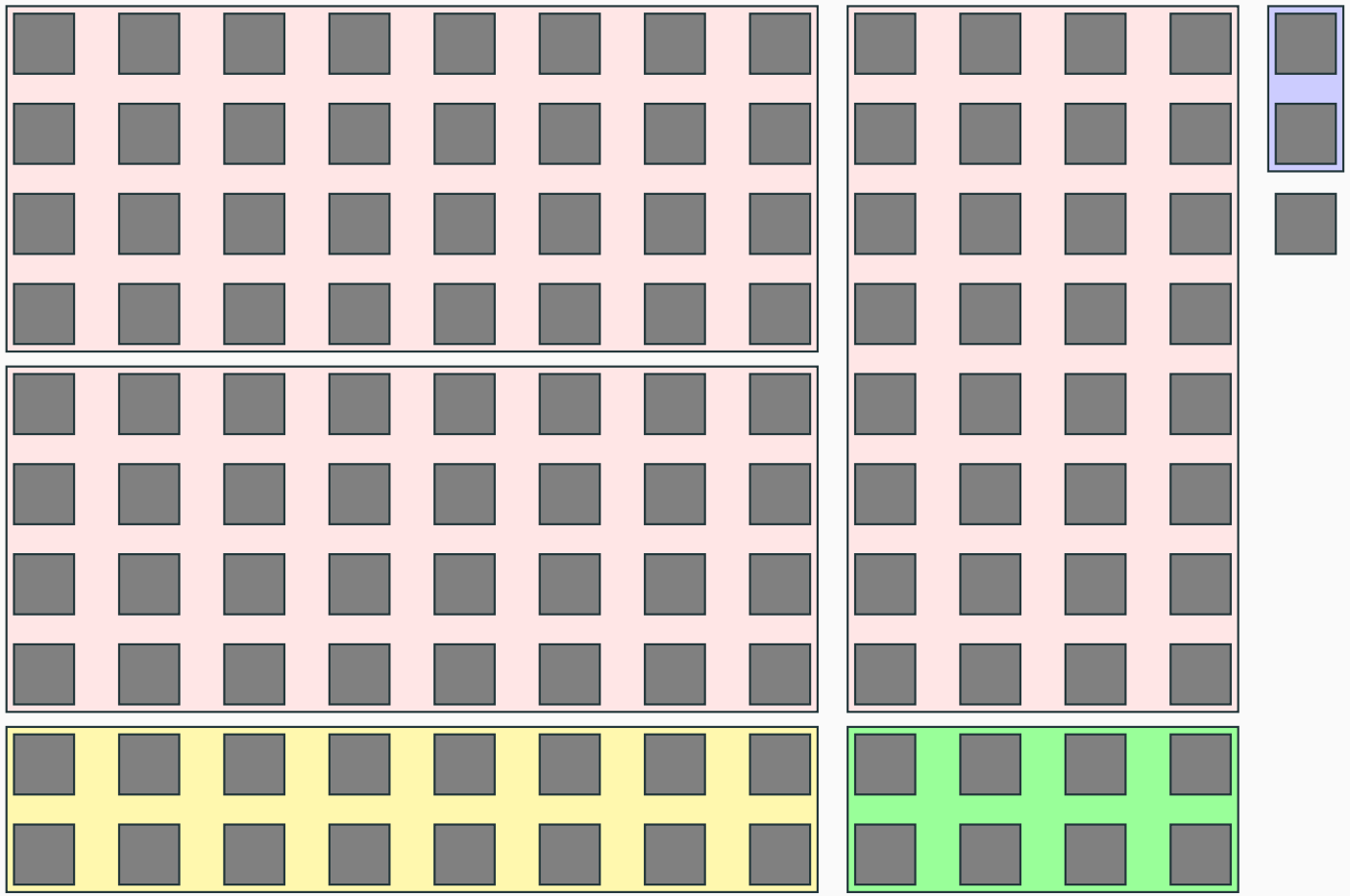
La représentation binaire (base 2)



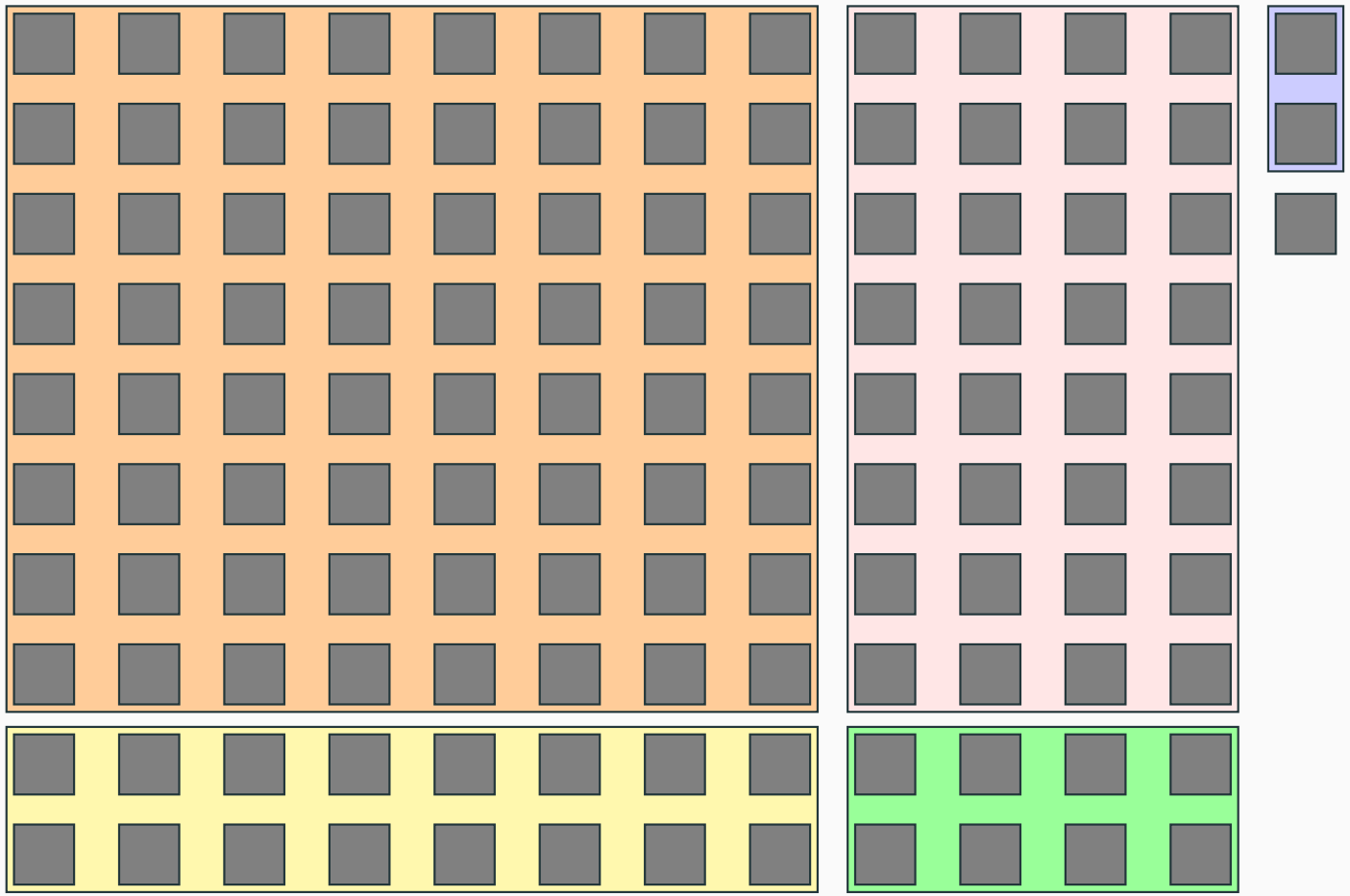
La représentation binaire (base 2)



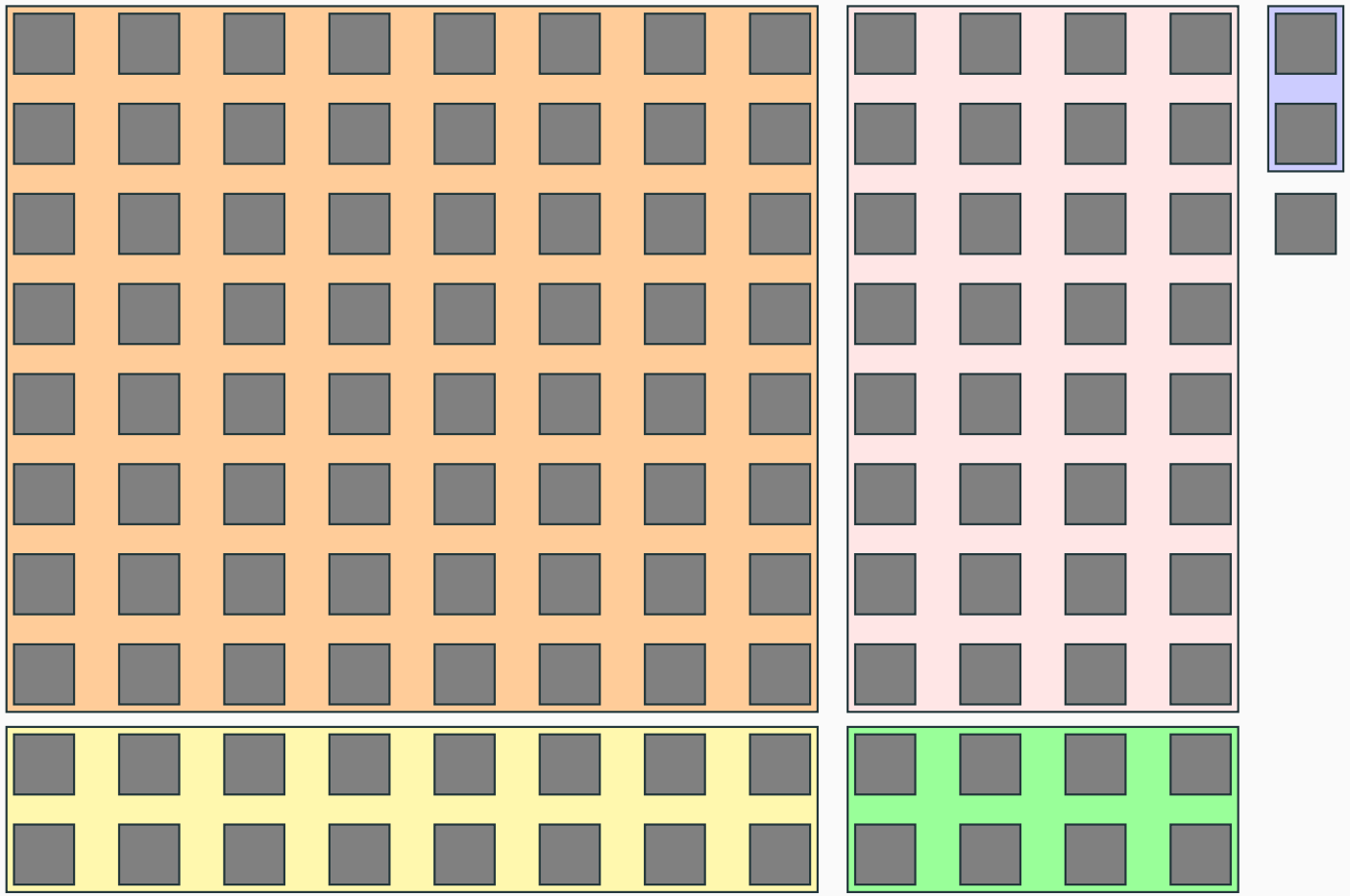
La représentation binaire (base 2)



La représentation binaire (base 2)



La représentation binaire (base 2)



$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 \rightarrow 1111011$$

Représentation en base b

Les nombres s'écrivent avec b chiffres différents :

- 0 et 1 en base 2 (binaire)
- 0, 1, 2, ..., 6, 7 en base 8 (octale)
- 0, 1, 2, ..., 8, 9 en base 10 (décimale)
- 0, 1, 2, ..., 8, 9, A, B, ..., F en base 16 (hexadécimale)

On obtient le nombre N en concaténant les chiffres a_i :

a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

$$\mathrm{N} = \sum_{i=0}^{\mathrm{N}} a_i b^i$$

Il faut $\lfloor \log_b(\mathrm{N}) + 1 \rfloor$ chiffres.

Conversion en base b

Décomposition :

$$a_i = \left\lfloor \frac{\mathrm{N}}{b^i} \right\rfloor \% b$$

Reconstruction :

$$\mathrm{N} = \sum_{i=0}^{\mathrm{N}} a_i \, b^i$$

On peut trouver des méthodes plus efficaces.

Algorithme de décomposition en base b

123

Algorithme de décomposition en base b

$$\begin{array}{r|l} 123 & 2 \\ \hline & \\ & \end{array}$$

Algorithme de décomposition en base b

$$\begin{array}{r|l} 123 & 2 \\ \hline 1 & 61 \end{array}$$

1

Algorithme de décomposition en base b

123	2		1
1	61	2	

Algorithme de décomposition en base b

$$
\begin{array}{r|l}
123 & 2 \\
\hline
1 & 61 \\
\end{array}
\qquad
\begin{array}{r|l}
61 & 2 \\
\hline
1 & 30 \\
\end{array}
$$

11

Algorithme de décomposition en base b

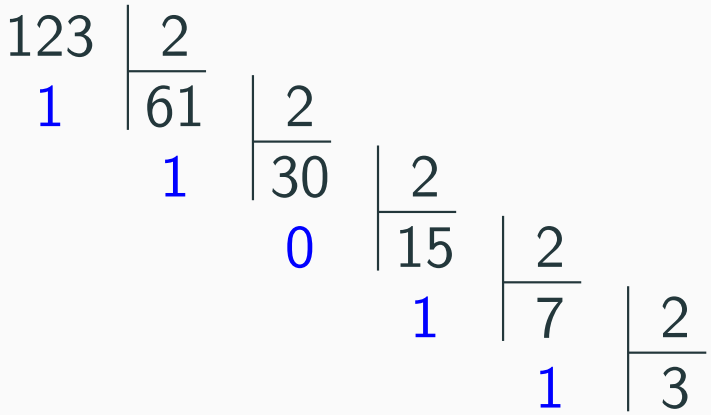
123	2		
1	61	2	
	1	30	2
		0	15

011

Algorithme de décomposition en base b

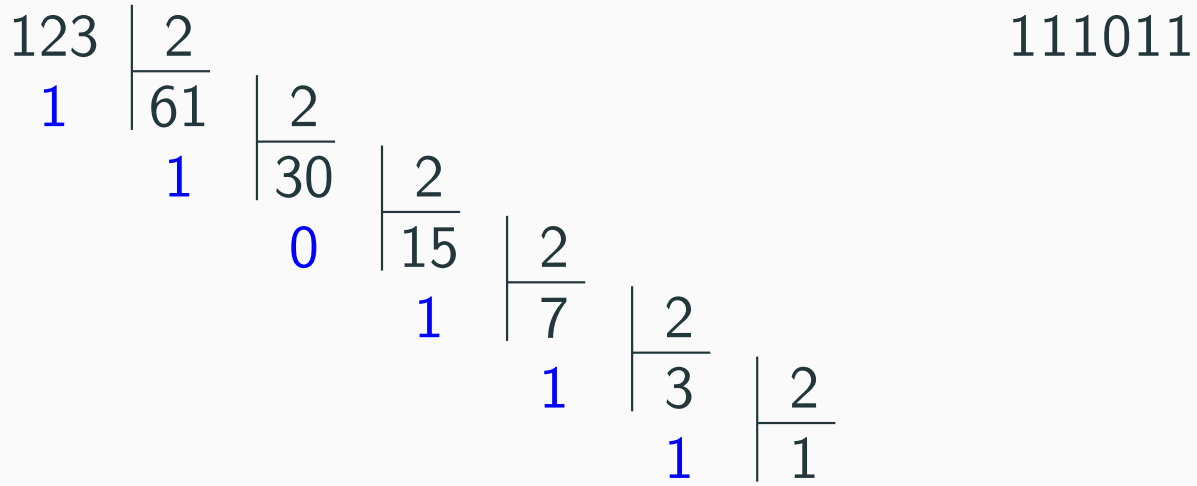
123	2				1011
1	61	2			
	1	30	2		
		0	15	2	
			1	7	

Algorithme de décomposition en base b

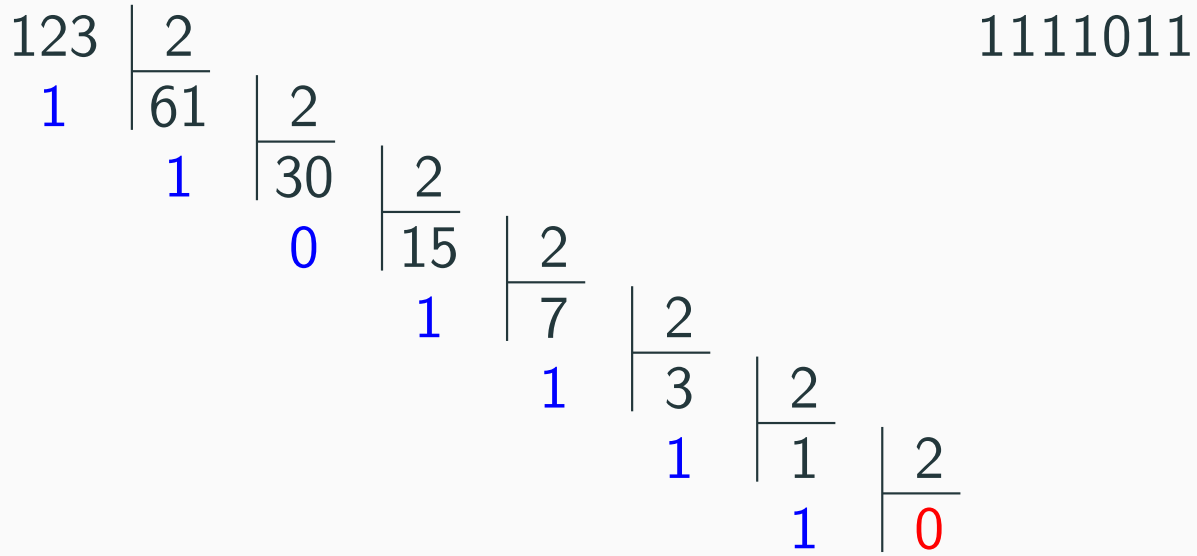


11011

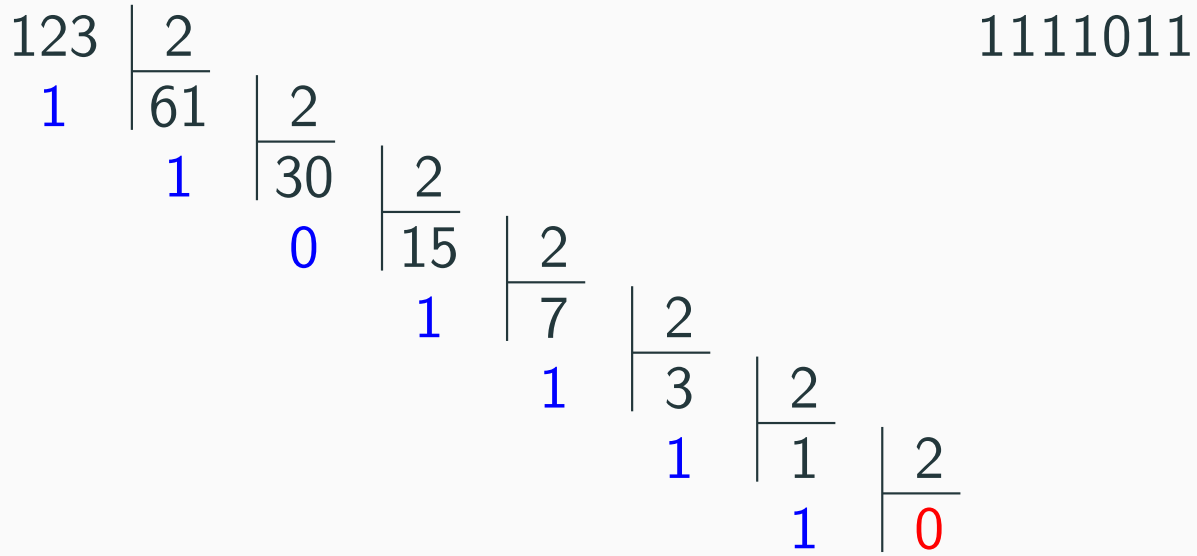
Algorithme de décomposition en base b



Algorithme de décomposition en base b



Algorithme de décomposition en base b




```
i = 0
while N>0 :
        a[i] = N % b
        i = i + 1
        N = N // b
```

Recomposition par la méthode de Horner

$$\mathrm{N} = \sum_{i=0}^{\mathrm{N}} a_i b^i = (a_0 + b \times (a_1 + b \times (\ldots \times (a_{n-2} + b \times (a_{n-1} + b \times a_n)) \ldots)$$

Recomposition par la méthode de Horner

$$\mathrm{N} = \sum_{i=0}^{\mathrm{N}} a_i b^i = (a_0 + b \times (a_1 + b \times (\ldots \times (a_{n-2} + b \times (a_{n-1} + b \times a_n)) \ldots)$$

```
N = 0
for ai in reversed(a) :
        N = b * N + ai
```

Représentation hexadécimale (base 16)

La notation binaire n'est pas toujours commode.

$$1492_{(10)} \rightarrow 10111010100_{(2)}$$

La conversion du binaire au décimal n'est pas immédiate.

Pour représenter des valeurs binaires, on utilise donc souvent la base hexadécimale (base 16).

Elle utilise les chiffres de 0 à 9 et de A à F :

$$1492_{(10)} = 5 \times 16^2 + 13 \times 16 + 4 \rightarrow 5\mathrm{D}4_{(16)}$$

Conversion binaire / hexadécimal

Le passage entre les deux bases est simple car $16 = 2^4$.

$$10111010100_{(2)} \rightarrow 5\mathrm{D}4_{(16)}$$

En effet, il suffit de grouper les chiffres par 4, en partant de la droite :

$$0100_{(2)} \rightarrow 4_{(10)} \rightarrow 4_{(16)}$$

$$1101_{(2)} \rightarrow 13_{(10)} \rightarrow \mathrm{D}_{(16)}$$

$$101_{(2)} \rightarrow 5_{(10)} \rightarrow 5_{(16)}$$

La conversion en sens inverse est tout aussi simple...

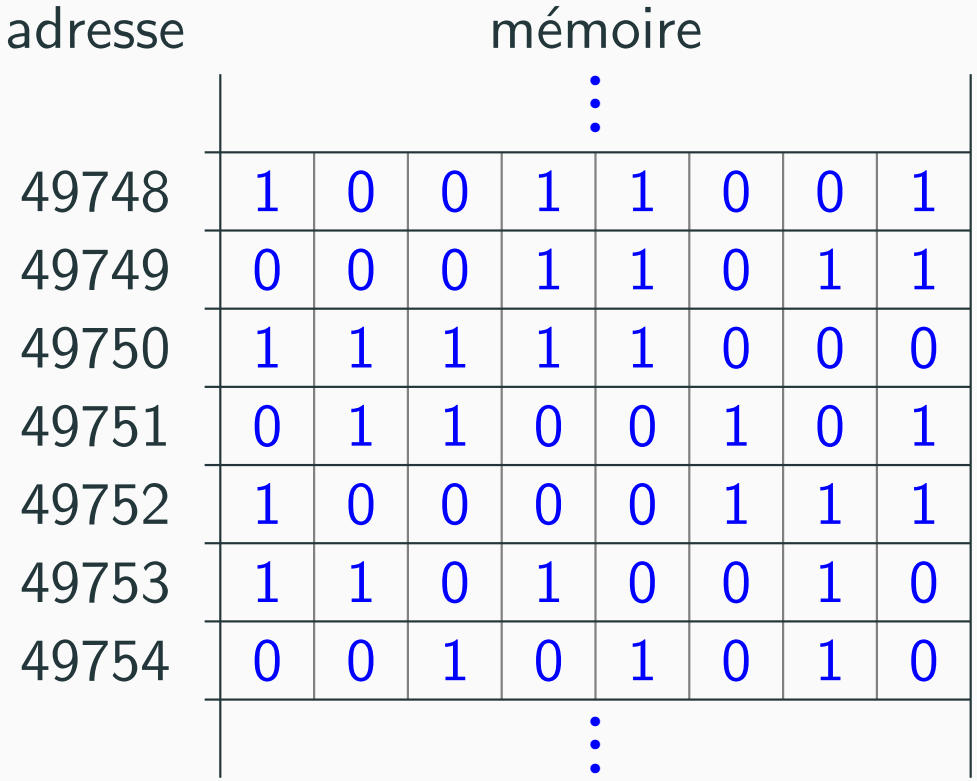
En Python

Python dispose de fonctions pour ces conversions :

```
In [1]: bin(1492)
Out[1]: '0b10111010100'

In [2]: hex(1492)
Out[2]: '0x5D4'

In [3]: int('10111010100', 2)
Out[3]: 1492

In [4]: int('5D4', 16)
Out[4]: 1492
```

La mémoire d'un ordinateur

Dans la mémoire de la plupart des ordinateurs, chaque « case » contient huit bits. Un groupe de huit bits est appelé un « octet ».



Entiers sur 8, 16, 32 et 64 bits

Un octet peut contenir un entier entre 0 et $\sum_{i=0}^{7} 2^i = 2^8 - 1 = 255$.

C'est fréquemment insuffisant, donc on utilise plusieurs octets :

- 16 bits → entre 0 et $2^{16} - 1 = 65535$;
- 32 bits → entre 0 et $2^{32} - 1 \simeq 4{,}3 \times 10^9$;
- 64 bits → entre 0 et $2^{64} - 1 \simeq 1{,}8 \times 10^{19}$...

L'ordre de rangement des octets en mémoire est potentiellement délicat.

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17 42 105

Et déterminer les entiers correspondant à ces représentions :

00001011 00100101 01011010

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001		

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001	00101010	

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001	00101010	01101001

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001	00101010	01101001

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010
11		

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001	00101010	01101001

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010
11	37	

Quelques exemples...

On souhaite coder sur 8 bits les entiers suivants :

17	42	105
00010001	00101010	01101001

Et déterminer les entiers correspondant à ces représentions :

00001011	00100101	01011010
11	37	90

Et les entiers négatifs ?

Dans une représentation sur p bits (ici 8 bits) :

On récupère la moitié des codes (ceux des plus grands entiers) :

0	1	2	...	127	128	...	254	255

pour représenter les entiers négatifs (jusqu'à -2^{p-1}) :

0	1	2	...	127	-128	...	-2	-1

Tout se passe comme si le bit a_{p-1} « valait » -2^{p-1} et non 2^{p-1}.

Ce codage est appelé **complément à deux**.

Entiers relatifs sur 8, 16, 32 et 64 bits

On peut ainsi coder sur p bits les entiers de -2^{p-1} à $2^{p-1}-1$.

- 8 bits → entre -128 et $+127$;
- 16 bits → entre $-2^{15} = -32768$ et $2^{15}-1 = 32767$;
- 32 bits → entre $-2^{31} \simeq -2{,}1 \times 10^9$ et $2^{31}-1 \simeq 2{,}1 \times 10^9$
- 32 bits → entre $-2^{63} \simeq -9{,}2 \times 10^{18}$ et $2^{63}-1 \simeq 9{,}2 \times 10^{18}$...

Le bit de poids fort (a_{p-1}) indique le signe ($0 \equiv$ positif, $1 \equiv$ négatif).

Codage d'un entier N négatif

Pour encoder sur p bits un entier $2^{p-1} \leq \mathrm{N} < 0$, possibilités :

- Coder $\mathrm{N} + 2^p$
- Coder $|\mathrm{N}|$, inverser tous les bits et ajouter 1.

Exemple du codage de -27 sur $p = 8$ bits :

$$27 \;=\; 16+8+2+1 \;\rightarrow\; 00011011_{(2)}$$

$$-27 \;\rightarrow\; 11100100_{(2)} + 1 \;=\; 11100101_{(2)}$$

L'octet $11100101_{(2)}$ peut représenter également l'entier 229, il faut donc se souvenir si l'on a stocké un entier relatif ou un entier naturel !

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94

Et inversement :

11111001	10011001	11001100

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011		

Et inversement :

11111001	10011001	11001100

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011	11011011	

Et inversement :

11111001	10011001	11001100

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011	11011011	10100010

Et inversement :

11111001	10011001	11001100

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011	11011011	10100010

Et inversement :

11111001	10011001	11001100
-7		

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011	11011011	10100010

Et inversement :

11111001	10011001	11001100
-7	-103	

Quelques exemples pour les entiers négatifs...

Toujours sur 8 bits, à coder en utilisant le complément à deux :

-5	-37	-94
11111011	11011011	10100010

Et inversement :

11111001	10011001	11001100
-7	-103	-52

Exemples d'additions d'entiers positifs

Pour calculer $17 + 42$:

$$\begin{array}{rr} & 00010001 \\ + & 00101010 \\ \hline = & \end{array}$$

Pour calculer $17 + 105$:

$$\begin{array}{rr} & 00010001 \\ + & 01101001 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers positifs

Pour calculer 17 + 42 :

$$\begin{array}{rr} & 00010001 \\ + & 00101010 \\ \hline = & 00111011 \end{array}$$

Pour calculer 17 + 105 :

$$\begin{array}{rr} & 00010001 \\ + & 01101001 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers positifs

Pour calculer 17 + 42 :

$$\begin{array}{rr} & 00010001 \\ + & 00101010 \\ \hline = & 00111011 \end{array} \rightarrow 59$$

Pour calculer 17 + 105 :

$$\begin{array}{rr} & 00010001 \\ + & 01101001 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers positifs

Pour calculer $17 + 42$:

$$\begin{array}{rr} & 00010001 \\ + & 00101010 \\ \hline = & 00111011 \end{array} \rightarrow 59$$

Pour calculer $17 + 105$:

$$\begin{array}{rr} & 00010001 \\ + & 01101001 \\ \hline = & 01111010 \end{array}$$

Exemples d'additions d'entiers positifs

Pour calculer $17 + 42$:

$$\begin{array}{rr} & 00010001 \\ + & 00101010 \\ \hline = & 00111011 \end{array} \rightarrow 59$$

Pour calculer $17 + 105$:

$$\begin{array}{rr} & 00010001 \\ + & 01101001 \\ \hline = & 01111010 \end{array} \rightarrow 122$$

Additions d'entiers de signe différent

Pour calculer 17 + (-37) :

$$\begin{array}{rr} & 00010001 \\ + & 11011011 \\ \hline = & \end{array}$$

Pour calculer 42 + (-94) :

$$\begin{array}{rr} & 00101010 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-37) :

$$\begin{array}{rr} & 00010001 \\ + & 11011011 \\ \hline = & 11101100 \end{array}$$

Pour calculer 42 + (-94) :

$$\begin{array}{rr} & 00101010 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-37) :

$$\begin{array}{rr} & 00010001 \\ + & 11011011 \\ \hline = & 11101100 \end{array} \rightarrow \text{-20}$$

Pour calculer 42 + (-94) :

$$\begin{array}{rr} & 00101010 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-37) :

$$\begin{array}{rr} & 00010001 \\ + & 11011011 \\ \hline = & 11101100 \end{array} \rightarrow \text{-20}$$

Pour calculer 42 + (-94) :

$$\begin{array}{rr} & 00101010 \\ + & 10100010 \\ \hline = & 11001100 \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-37) :

$$\begin{array}{rr} & 00010001 \\ + & 11011011 \\ \hline = & 11101100 \end{array} \rightarrow \text{-20}$$

Pour calculer 42 + (-94) :

$$\begin{array}{rr} & 00101010 \\ + & 10100010 \\ \hline = & 11001100 \end{array} \rightarrow \text{-52}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & \end{array}$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{r} 00010001 \\ +\ 11111011 \\ \hline = 100001100 \end{array}$$

Pour calculer 105 + (-37) :

$$\begin{array}{r} 01101001 \\ +\ 11011011 \\ \hline = \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 100001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{r} 00010001 \\ +\ 11111011 \\ \hline = 100001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{r} 01101001 \\ +\ 11011011 \\ \hline = 101000100 \end{array}$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 100001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & 101000100 \end{array} \rightarrow 68$$

Additions d'entiers de signe différent

Pour calculer 17 + (-5) :

$$\begin{array}{r} 00010001 \\ +\ 11111011 \\ \hline = 100001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{r} 01101001 \\ +\ 11011011 \\ \hline = 101000100 \end{array} \rightarrow 68$$

Les résultats sont corrects si l'on ignore le 1 supplémentaire !

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & \end{array}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & 10010011 \end{array}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & \end{array}$$

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & 101111101 \end{array}$$

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & 101111101 \end{array} \rightarrow 125$$

Additions d'entiers de même signe

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 10010011 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

Pour calculer (-37) + (-94) :

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = & 101111101 \end{array} \rightarrow 125$$

Les résultats sont incorrects sans le bit supplémentaire.

Débordement

On dit qu'il y a **débordement** lorsque le résultat est trop grand (positif ou négatif) pour tenir dans le nombre de bits fixés.

Une addition sur des entiers en complément à 2 de signe différents ne déborde jamais.

Une somme de deux entiers positifs trop grands donne un négatif.

Une somme de deux entiers négatifs trop grands donne un positif.

Le processeur détecte ces débordements.

Conséquences d'un débordement

La plupart des langages ne font rien de particulier !

Les conséquences peuvent être graves...

Premier vol d'Ariane 5 en 1996 : autodestruction.

Pourquoi ?

Un capteur mesure l'inclinaison (en fait, la vitesse horizontale)

La valeur est stockée comme un entier relatif sur 22 bits.

Après quelques secondes, la fusée s'incline un peu vers la gauche,

mais la valeur déborde...

Pourquoi ?

Un capteur mesure l'inclinaison (en fait, la vitesse horizontale)

La valeur est stockée comme un entier relatif sur 22 bits.

Après quelques secondes, la fusée s'incline un peu vers la gauche,

mais la valeur déborde...

... c'est interprété comme une inclinaison à droite.

La correction de trajectoire amplifie le problème, jusqu'à ce que

Additions binaires et électronique

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$

Additions binaires et électronique

$$\begin{array}{cccc} & & a_1 & a_0 \\ + & & b_1 & b_0 \\ \hline = & r_2 & r_1 & r_0 \end{array}$$

$$0+0=0$$

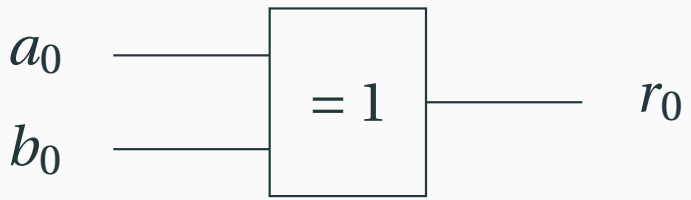
$$0+1=1$$

$$1+0=1$$

$$1+1=10$$

Additions binaires et électronique

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$



$$0 + 0 = 0$$

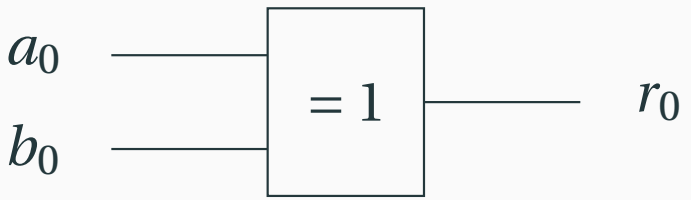
$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

Additions binaires et électronique

$$
\begin{array}{cccc}
 & & c_1 & \\
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$



$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

Additions binaires et électronique

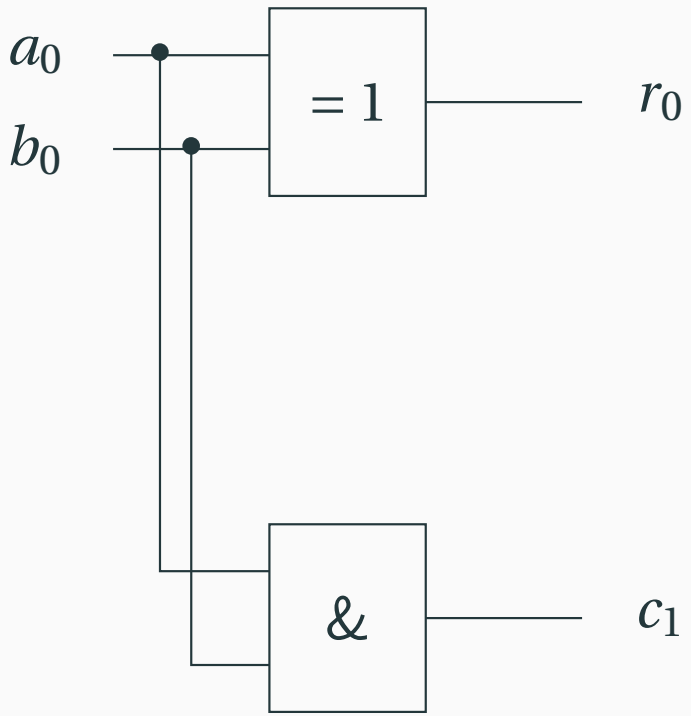
$$
\begin{array}{cccc}
 & & c_1 & \\
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$

$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1=10$$



Additions binaires et électronique

$$
\begin{array}{cccc}
 & c_2 & c_1 & \\
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$

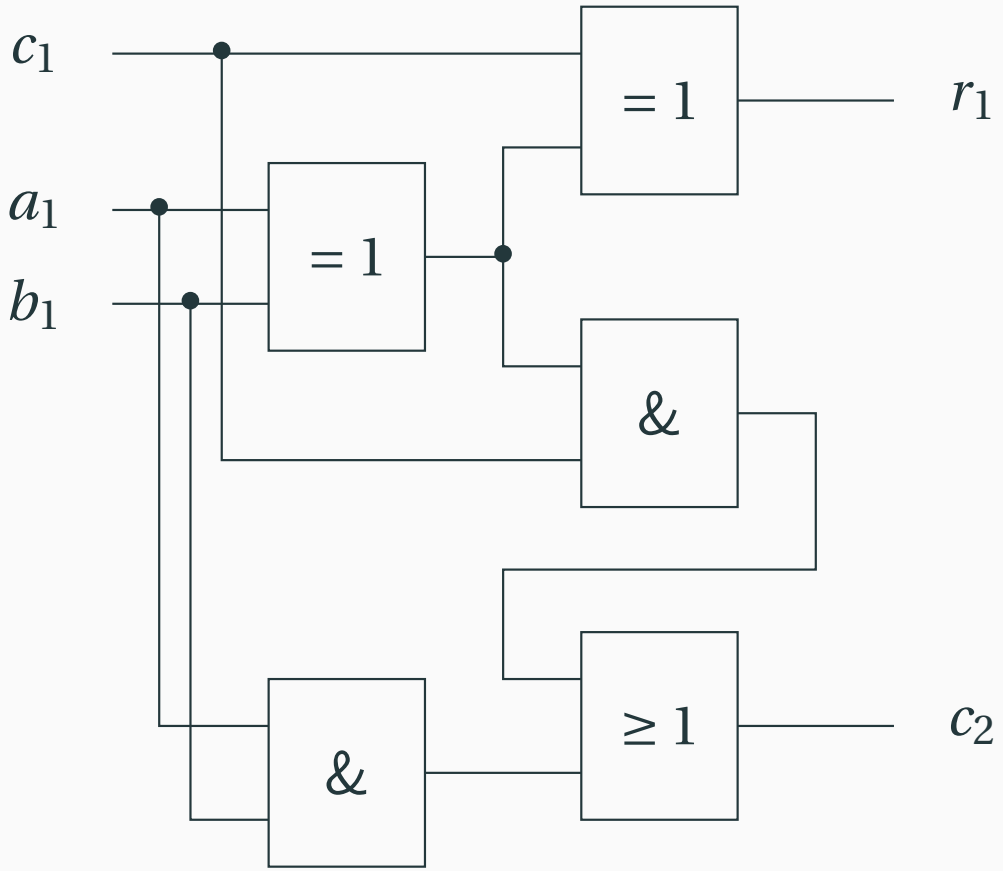
$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1=10$$

$$1+1+1=11$$



Déroulement de la présentation

Nombres réels et binaire

Nous avons vu comment décomposer un entier en binaire :

$$21 = 2^4 + 2^2 + 2^0$$

$$21 = 10101_{(2)}$$

Nombres réels et binaire

Nous avons vu comment décomposer un entier en binaire :

$$21 = 2^4 + 2^2 + 2^0$$

$$21 = 10101_{(2)}$$

Pour un réel, c'est la même chose, en ajoutant les puissances négatives de 2 :

$$21.625 = 2^4 + 2^2 + 2^0 + 2^{-1} + 2^{-3}$$

$$21.625 = 10101.101_{(2)}$$

Nombres réels en binaire

Un nombre avec un nombre fini de décimales en base 10 peut avoir une infinité de chiffres après la virgule en base 2 :

$$21.9 = 2^4 + 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-10} + 2^{-11} ...$$

$$21.9 = 10101.11100110011100..._{(2)}$$

Toutefois, on peut montrer qu'il y a nécessairement une périodicité (comme pour les décimales d'un rationnel).

On note parfois la périodicité par une barre supérieure :

$$21.9 = 10101.1\overline{11100}_{(2)}$$

Celle-ci peut être très longue.

Comment stocker un réel en mémoire ?

On souhaite ranger cette représentation dans une bloc de 32 bits ou de 64 bits.

Problèmes :

- on ne peut pas représenter la virgule ;
- il peut y avoir un nombre quelconque de chiffres de *chaque* côté.

Solution :

La représentation *scientifique*

La représentation scientifique pour les décimaux

On peut mettre n'importe quel décimal non nul sous la forme :

$$\pm a \times 10^b \quad \text{où} \quad 1 \leq a < 10 \quad \text{et} \quad b \quad \text{entier.}$$

Cette représentation est unique !

Par exemple, $102349.2211 = +1.023492211 \times 10^5$.

La représentation scientifique en base 2

Même chose, mais en écrivant le nombre sous la forme

$$\pm a \times 2^b \quad \text{où} \quad 1 \leq a < 2 \quad \text{et} \quad b \quad \text{entier.}$$

Par exemple, $21.625 = 10101.101_{(2)} = +1.0101101_{(2)} \times 2^4 = +1.0101101_{(2)} \times 10_{(2)}^{100_{(2)}}$.

Quels morceaux stocker ?

Les parties en rouge sont toujours présentes, seules les parties en bleu sont importantes

$$21.625 = +1.0101101_{(2)} \times 2^{4}$$

Soit :

- le signe (+ ou −) ;
- les chiffres *après* la virgule ;
- l'exposant.

La représentation IEEE sur 32 bits

$$21.625 = +1.0101101_{(2)} \times 2^4$$

Pour représenter un réel sur 32 bits, la norme IEEE propose :

- 1 bit pour le signe (0 pour $+$ et 1 pour $-$) ;
- 8 bits pour l'exposant ;
- 23 bits pour la *mantisse* (chiffres *après* la virgule).

Pour l'exposant, on ne stocke pas un entier signé en complément à 2

mais l'entier non signé $b + \left(2^{8-1} - 1\right)$ soit $b + 127$.

signe	exposant+127	mantisse

Et zéro ?

Zéro ne peut pas être représenté sous la forme $a \times 2^b$ avec $1 \leq a$!

On utilise alors le codage spécifique suivant :

signe	exposant+127	mantisse
0	00000000	00000000000000000000000

Pour ne pas confondre avec 1.0×2^{-127}, les valeurs admises, pour un nombre normalisé, pour exposant+127 sont celles comprises entre

- $00000001_{(2)}$, soit un exposant égal à -126 ;
- $11111110_{(2)}$, soit un exposant égal à $+127$;

Quels nombres positifs peut-on représenter ?

Plus petit nombre positif normalisé :

signe	exposant+127	mantisse
0	00000001	00000000000000000000000

$$1.00000000000000000000000_{(2)} \times 2^{-126} \simeq 1.17549435 \times 10^{-38}$$

Plus grand nombre positif normalisé :

signe	exposant+127	mantisse
0	11111110	11111111111111111111111

$$1.11111111111111111111111_{(2)} \times 2^{+127} \simeq 3.40282346 \times 10^{+38}$$

Autres codages particuliers

En fait, il existe d'autres codages :

- pour représenter *moins zéro* ;
- pour représenter $+\infty$ et $-\infty$;
- pour représenter *not-a-number* (NaN) ;
- pour représenter des valeurs très petites, avec cependant moins de chiffres significatifs (représentation dite *dénormalisée*)
  - entre 1.4×10^{-45} et $1.17549421 \times 10^{-38}$
  - entre $-1.17549421 \times 10^{-38}$ et -1.4×10^{-45}

Pourquoi une norme ?

La norme IEEE-754 a été introduite en 1985.

Elle définit notamment :

- comment représenter des réels en binaire (les *flottants*) ;
- la façon d'effectuer les calculs sur ces flottants

  (tous les bits du résultat sont corrects pour $+$, $-$, $\times$, $\div$, $\sqrt{}$) ;
- différentes règles d'arrondis.

Avant cette norme :

- le même calcul donnait des résultats différents selon le processeur et le langage (manque de portabilité) ;
- certains choix étaient curieux ou surprenants

Précision sur une valeur normalisée

Le dernier bit significatif d'une valeur normalisée vaut 2^{-23+b}.

On a donc une précision *relative* $\frac{\Delta x}{x}$ d'environ $2^{-23} \simeq 1.2 \times 10^{-7}$.

Les calculs se font donc avec (au mieux) sept chiffres significatifs.

Valeurs représentables proches de 1 :

0 01111110 11111111111111111111110	0.99999988
0 01111110 11111111111111111111111	0.99999994
0 01111111 00000000000000000000000	1.00000000
0 01111111 00000000000000000000001	1.00000012
0 01111111 00000000000000000000010	1.00000024...

La plupart des valeurs ne sont pas représentables !

Par exemple,
$0.3 = 1.\overline{0011}_{(2)} \times 2^{-2} \simeq 1.00110011001100110011... \times 2^{-2}$

Il est nécessaire d'arrondir (ou de tronquer) la mantisse.

Valeurs représentables proches de 0.3 :

0 01111101 00110011001100110011000	0.299999952
0 01111101 00110011001100110011001	0.299999982
0 01111101 00110011001100110011010	0.300000012 ←
0 01111101 00110011001100110011011	0.300000042
0 01111101 00110011001100110011100	0.300000072...

Et l'affichage ?

Une infinité de réels ont donc la même approximation flottante.

0 01111011 10100011000001010101001	0.1022999957...
0 01111011 10100011000001010101010	0.1023000032... (A)
0 01111011 10100011000001010101011	0.1023000106...

Tous les réels dans l'intervale $[0.10229999945, 0.1023000069]$ sont représentés par le même flottant (A).

À l'affichage, on choisit celui *avec la plus courte mantisse décimale*.

On affiche donc ici 0.1023.

En cas d'ambiguité...

Parfois, il y a plusieurs « candidats » possibles :

0 01111011 10100011000001010101000	0.1022999882...
0 01111011 10100011000001010101001	0.1022999957... (B)
0 01111011 10100011000001010101010	0.1023000032...

8 candidats avec 9 décimales dans $[0.10229999195, 0.10229999945]$.

On choisit le plus proche de la valeur correspondant à la représentation, c'est-à-dire dans le cas présent 0.102299996.

Bilan sur l'affichage

Finalement, voici quelques flottants 32 bits et l'affichage associé :

Représentation flottante 32 bits	Affichage
0 01111011 10100011000001010101000	0.10229999
0 01111011 10100011000001010101001	0.102299996
0 01111011 10100011000001010101010	0.1023
0 01111011 10100011000001010101011	0.10230001
0 01111011 10100011000001010101100	0.10230002

On « cache » l'approximation autant que possible.

Limitations des flottants 32 bits

Les flottants sur 32 bits sont fréquemment insuffisants :

- seulement 7 chiffres significatifs ;
- pas de nombres entre 0 et $1{,}40 \times 10^{-45}$;
- pas de nombres supérieurs à $3{,}40 \times 10^{38}$.

→ Flottants IEEE sur 64 bits (double précision) :

- 1 bit pour le signe (0 pour + et 1 pour $-$) ;
- 11 bits pour l'exposant (on range exposant$+\left(2^{11-1}-1\right)$) ;
- 52 bits pour la mantisse (chiffres *après* la virgule).

Flottants sur 64 bits (double précision)

Précision *relative* $\frac{\Delta x}{x}$ d'environ $2^{-52} \simeq 2.2 \times 10^{-16}$,

soit quinze à seize chiffres significatifs.

Peut représenter des nombres $4{,}94 \times 10^{-324}$ à $1{,}80 \times 10^{308}$

(avec précision réduite pour ceux inférieurs à $2{,}23 \times 10^{-308}$).

Utilisé en Python par défaut.