Représentation des données numériques

G. Dewaele

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)...

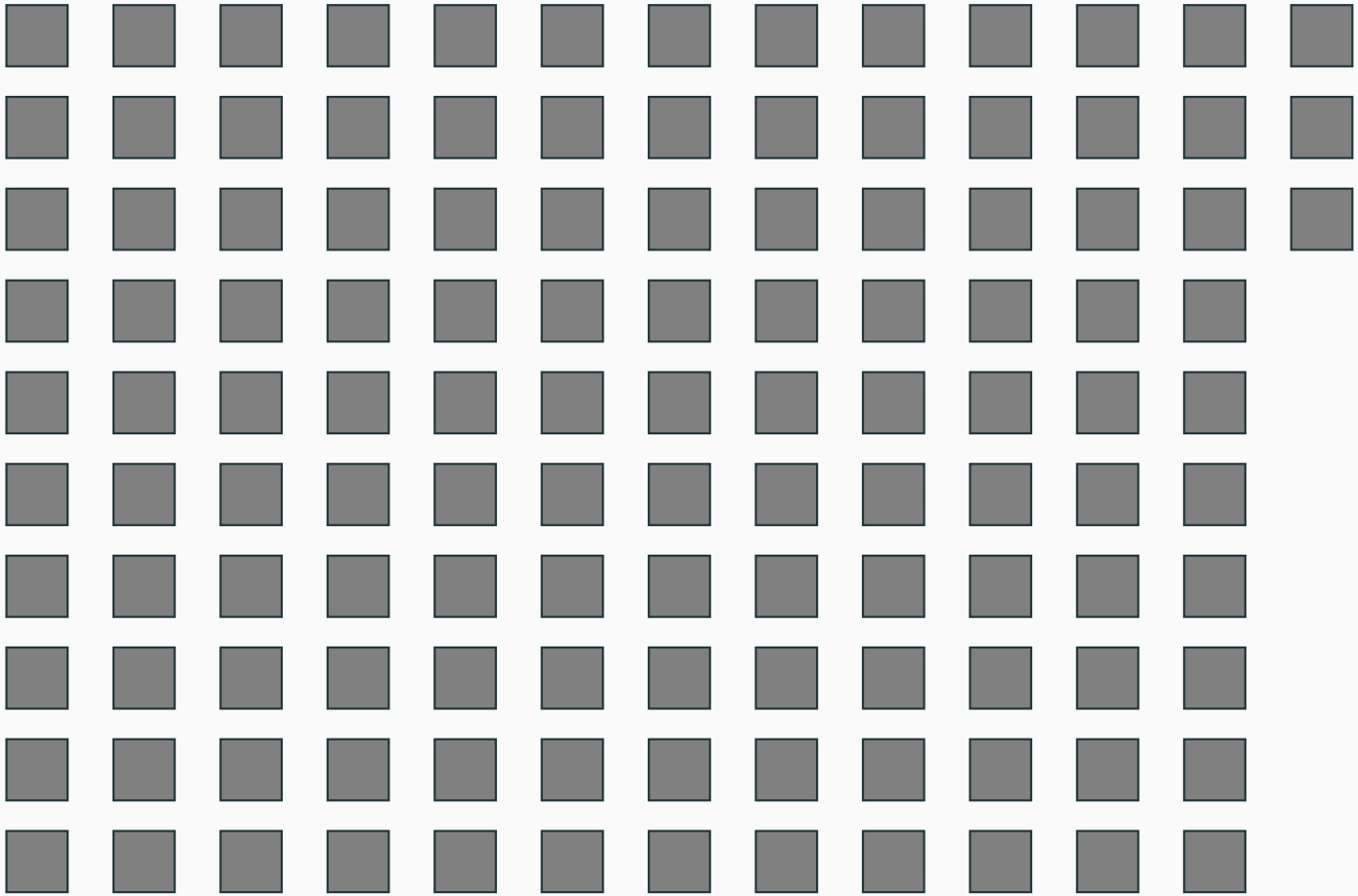
Avoir plus de deux états augmente les risques de confusion.

Conséquences

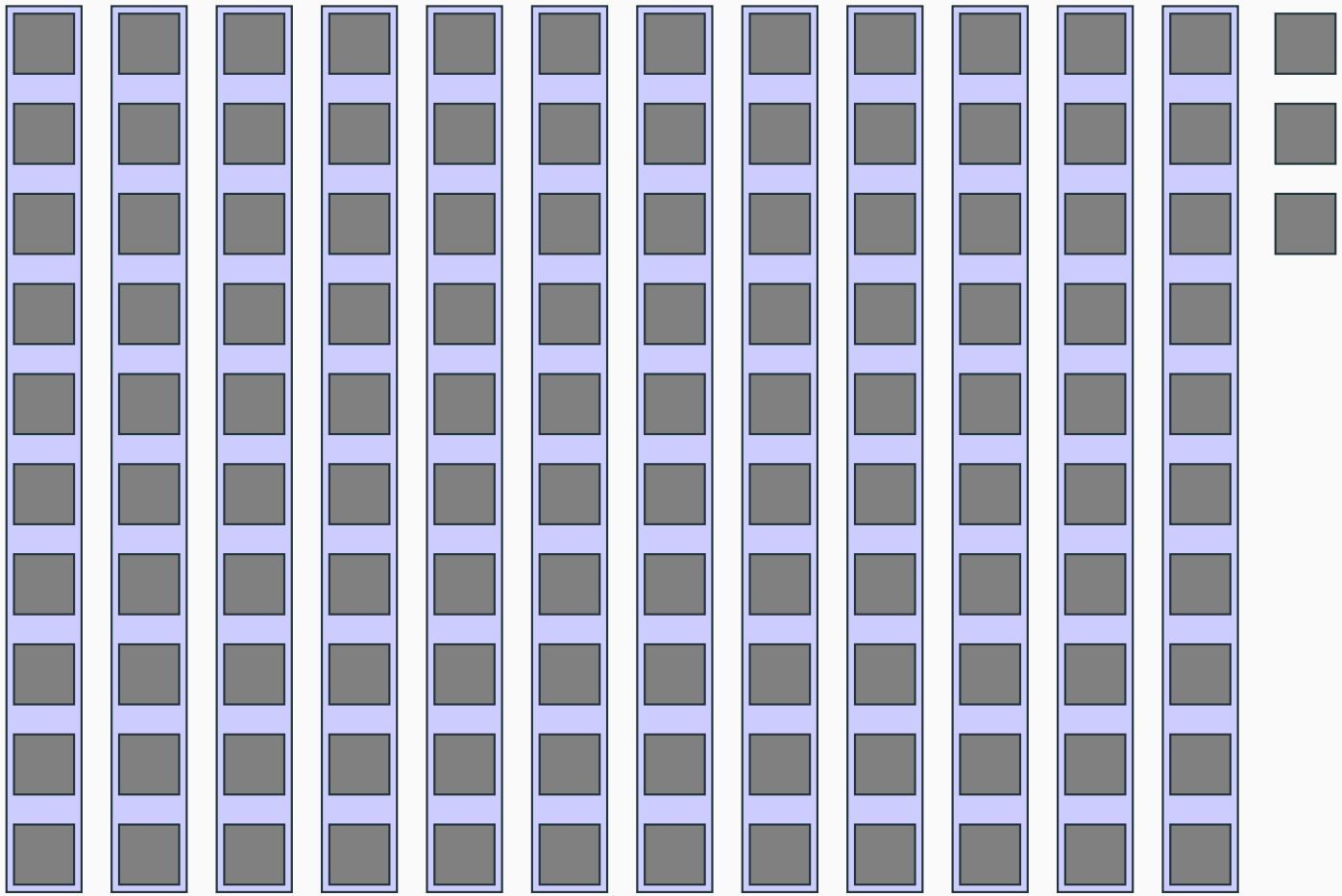
Il faut représenter les objets à manipuler par des suites de 0 et 1 :

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

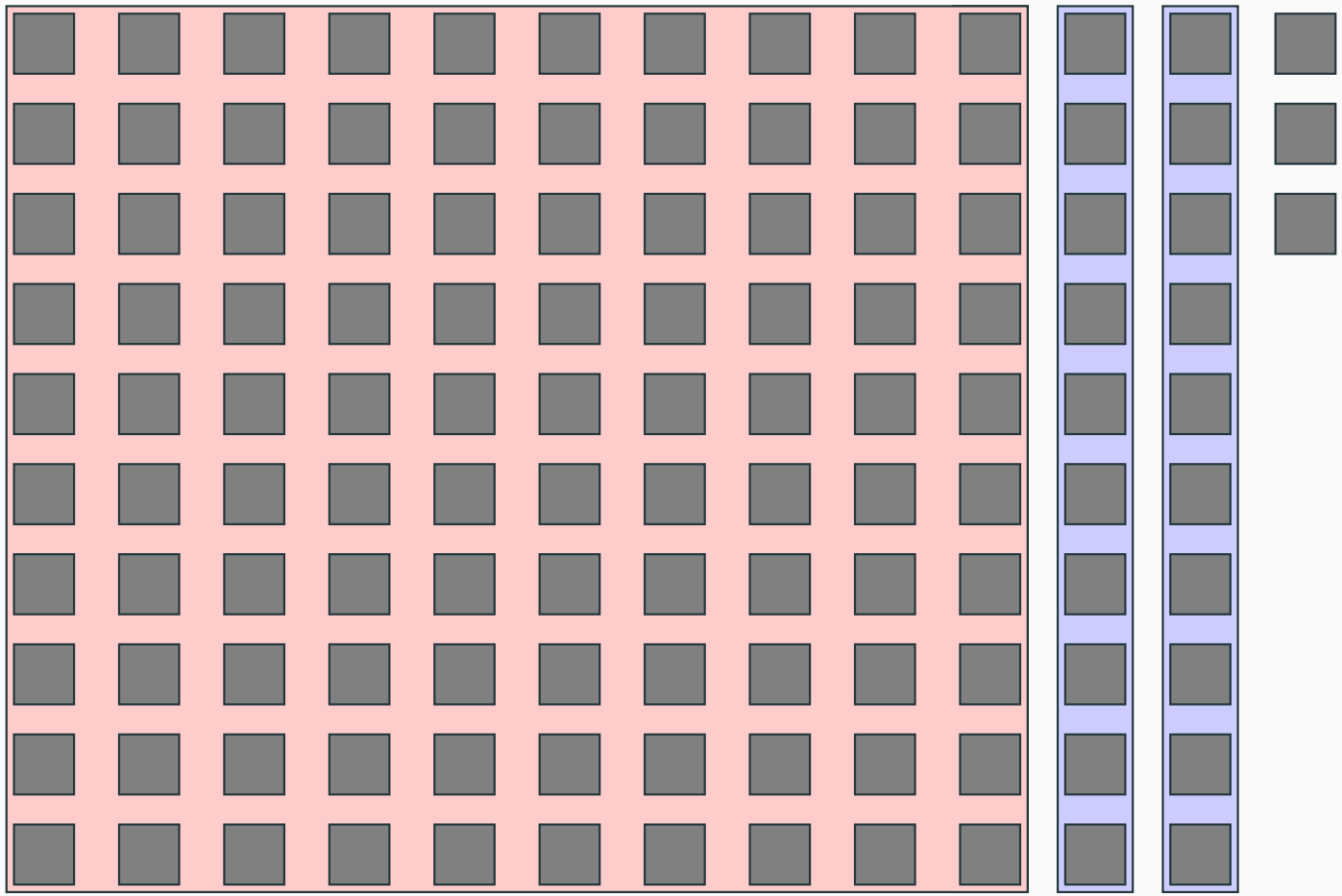
La représentation décimale (base 10)



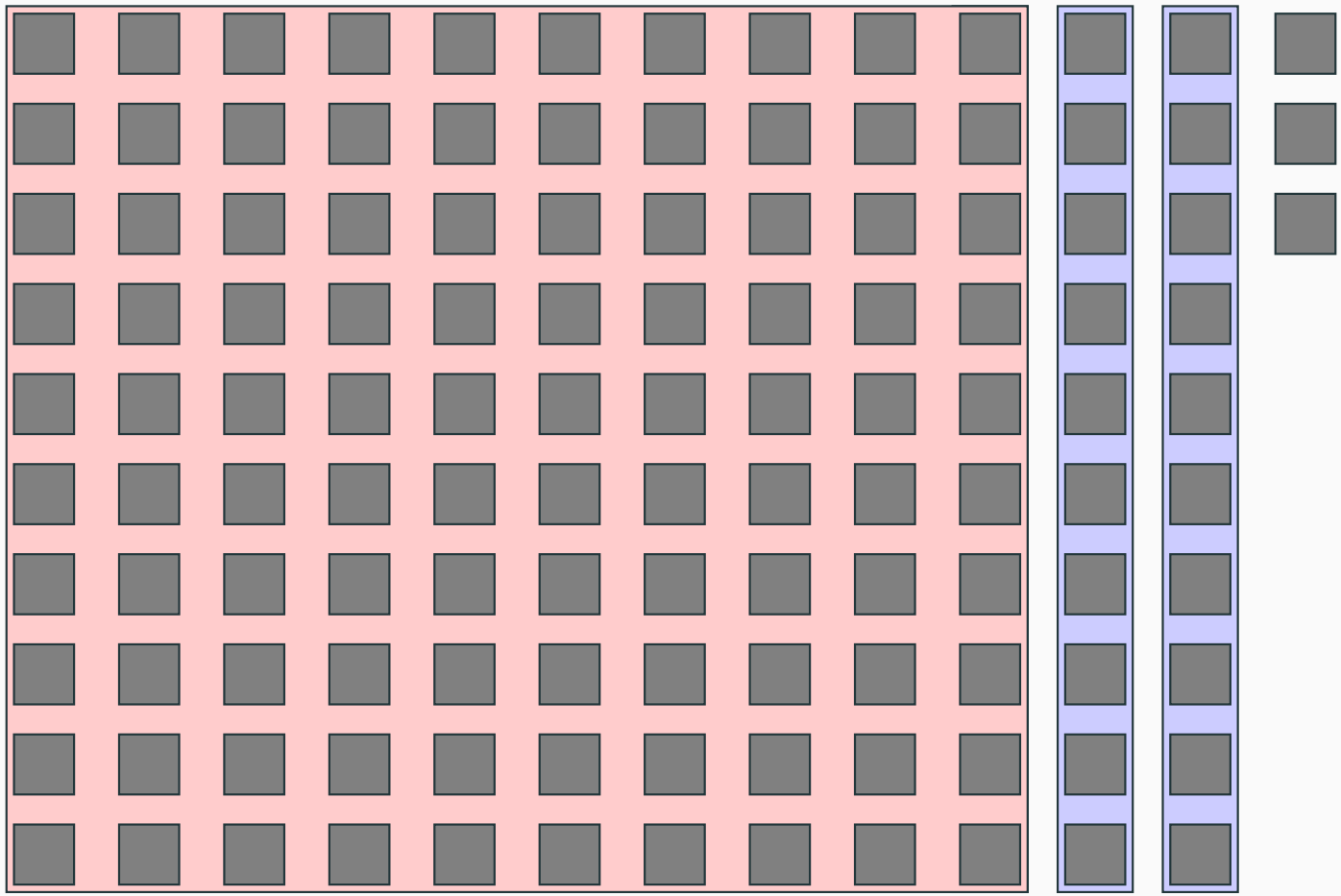
La représentation décimale (base 10)



La représentation décimale (base 10)

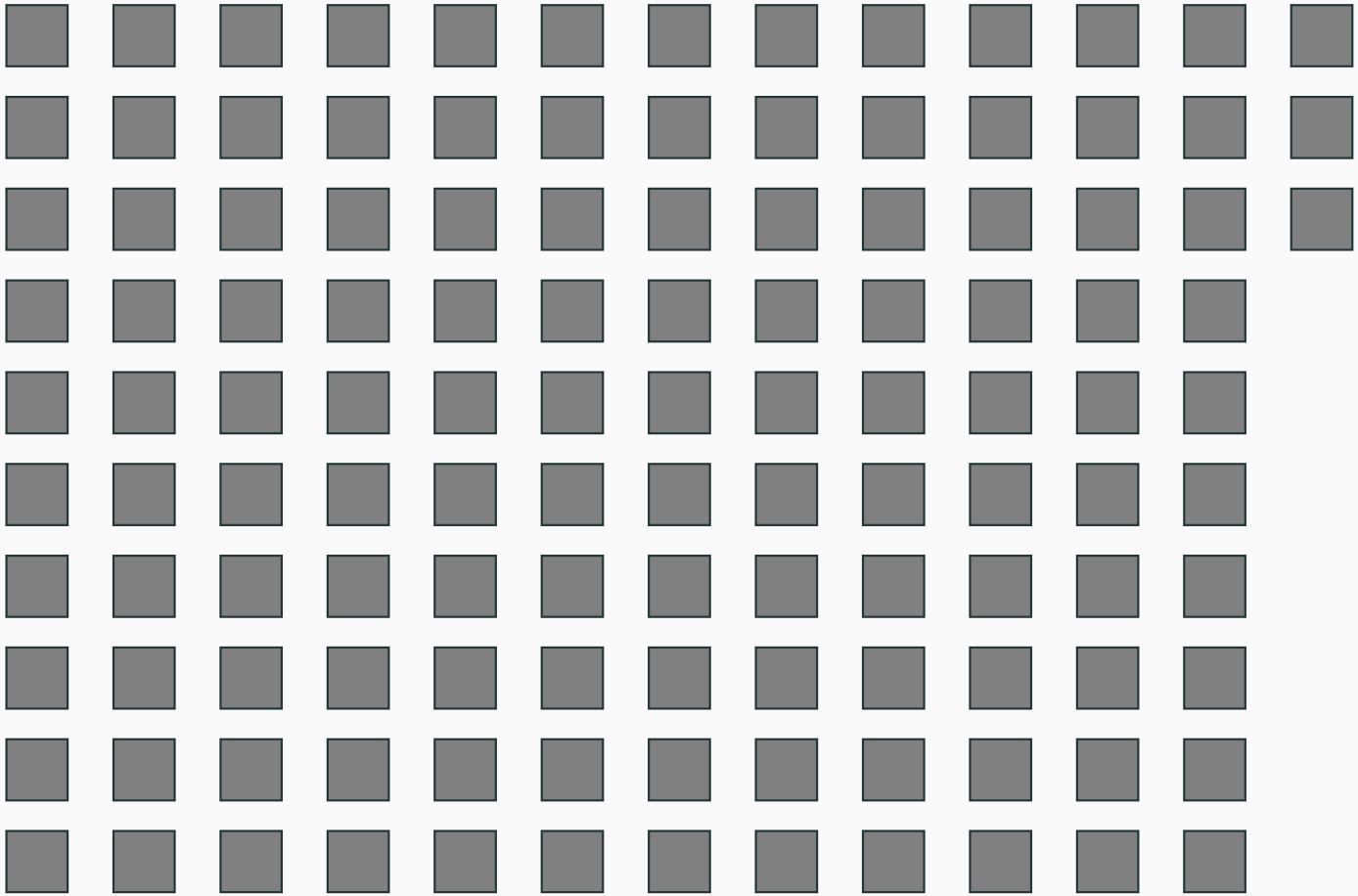


La représentation décimale (base 10)

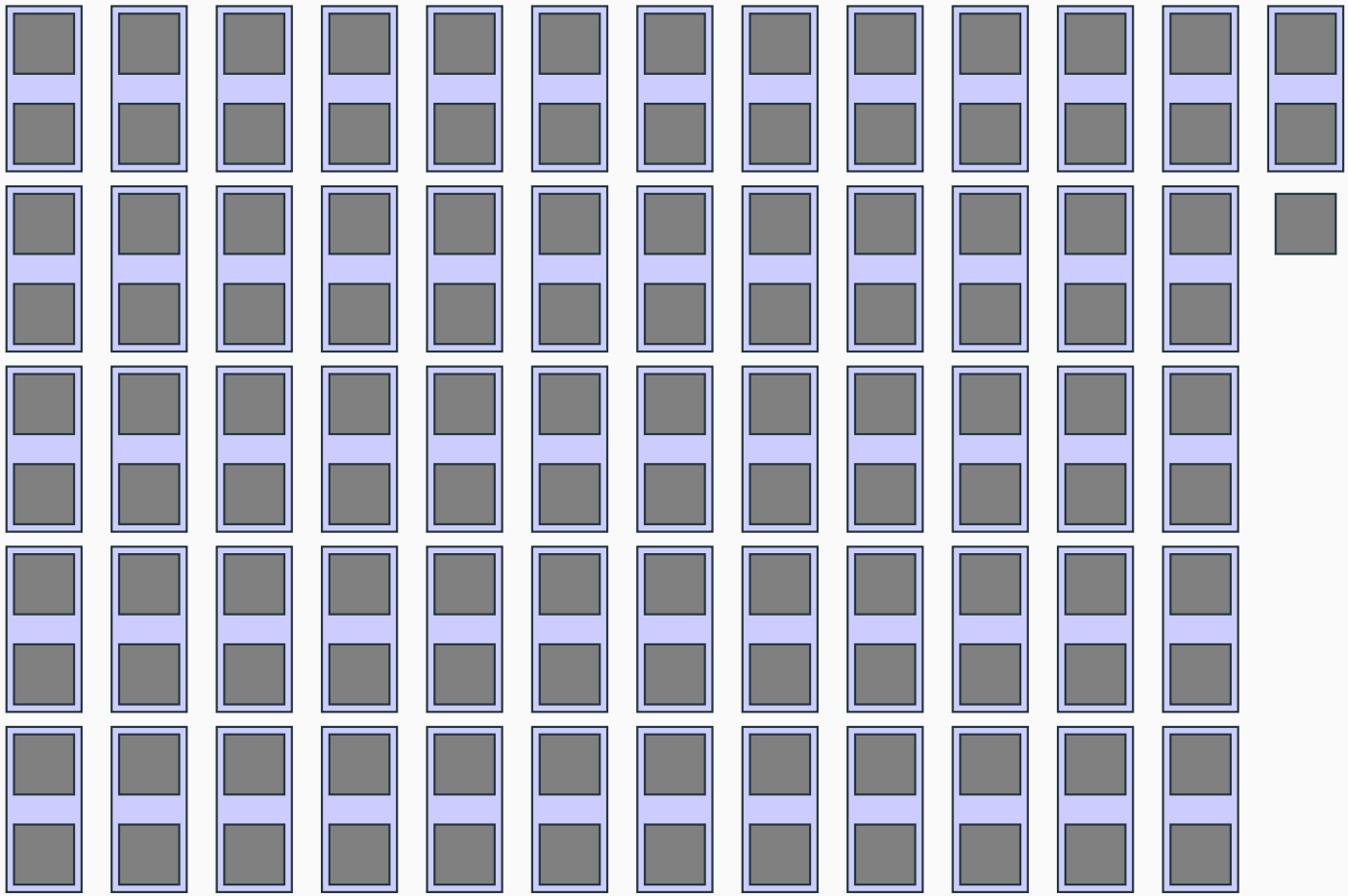


$$1 \times 10^2 + 2 \times 10 + 3 \rightarrow 123$$

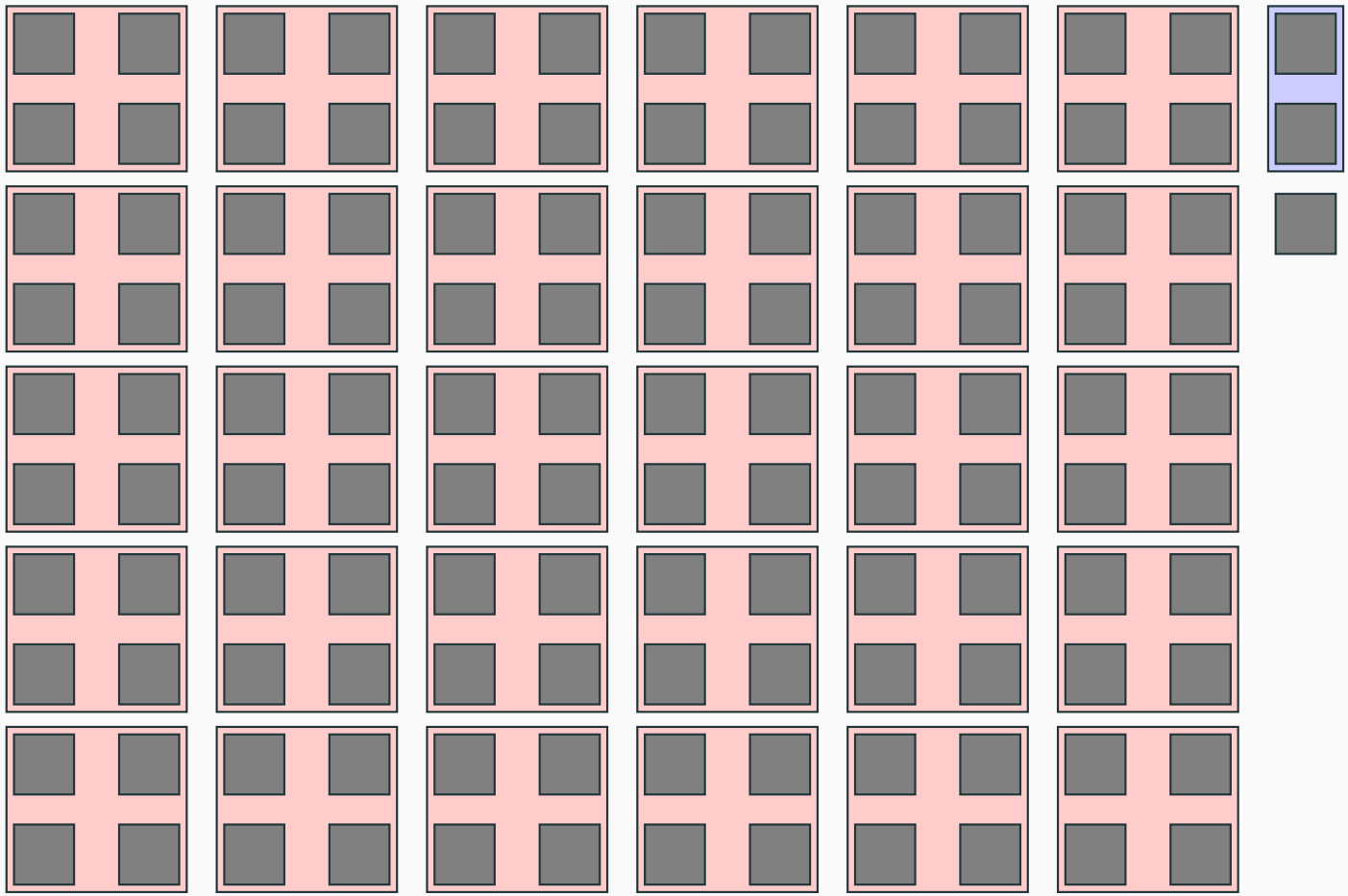
La représentation binaire (base 2)



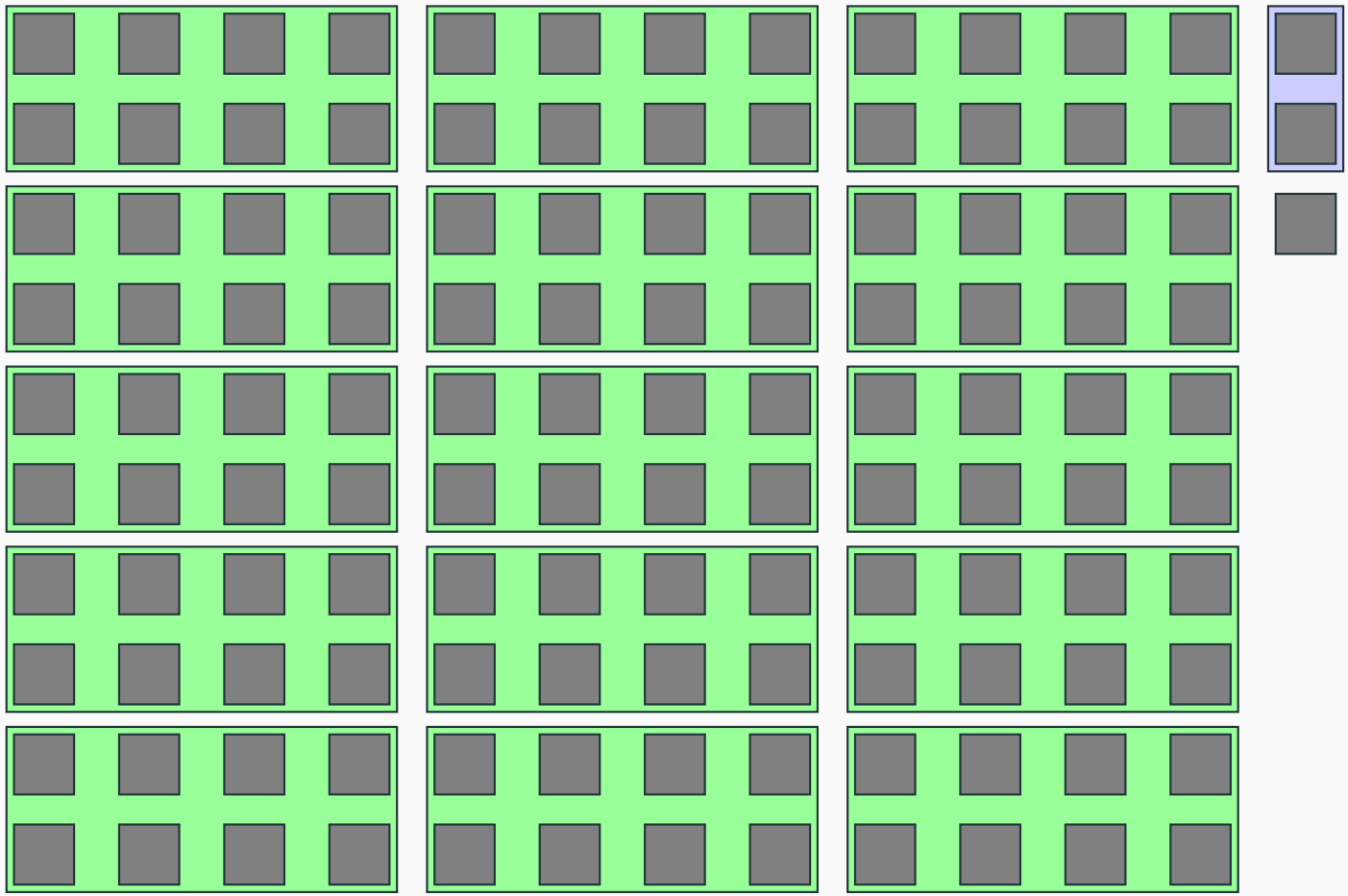
La représentation binaire (base 2)



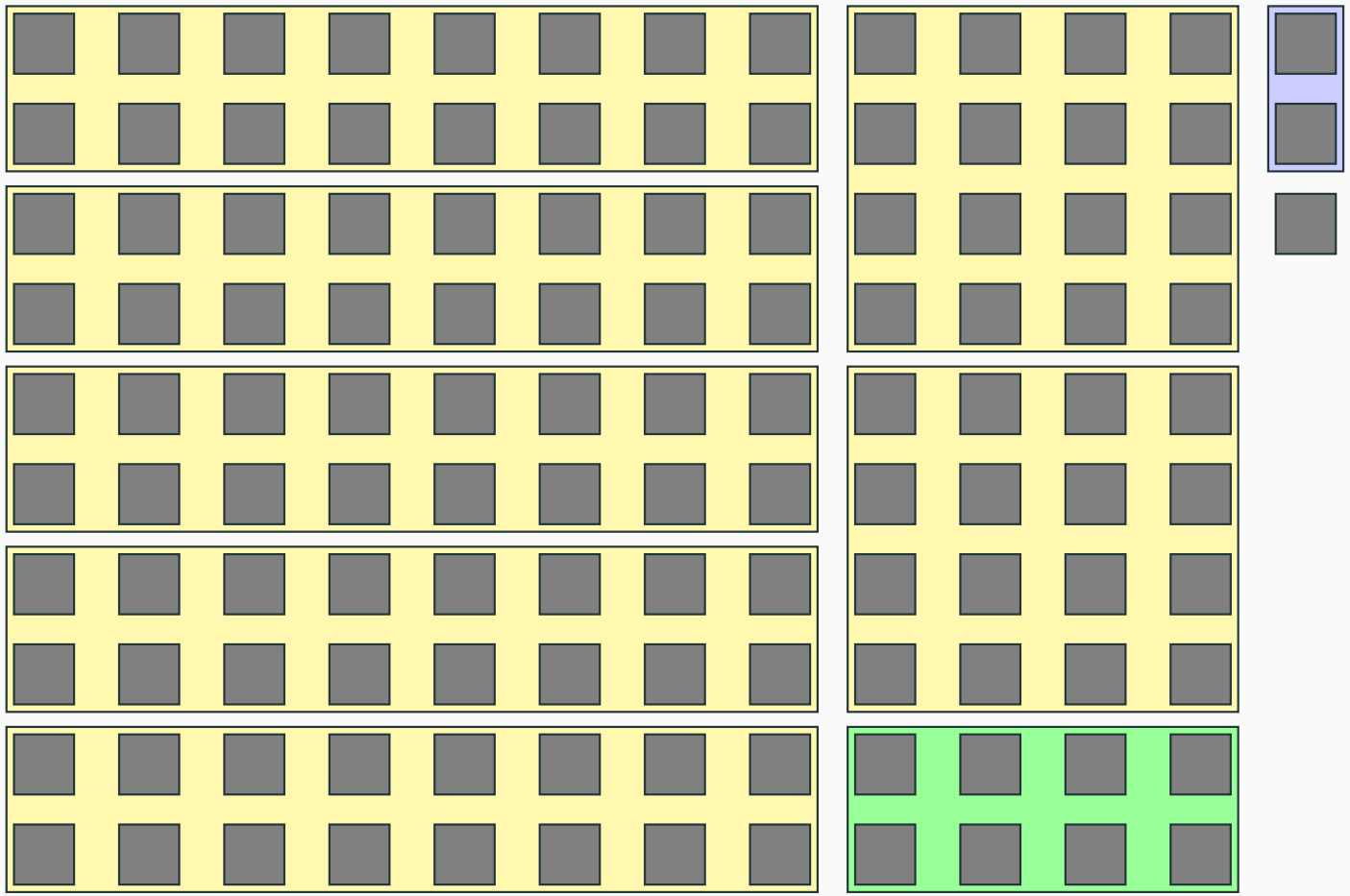
La représentation binaire (base 2)



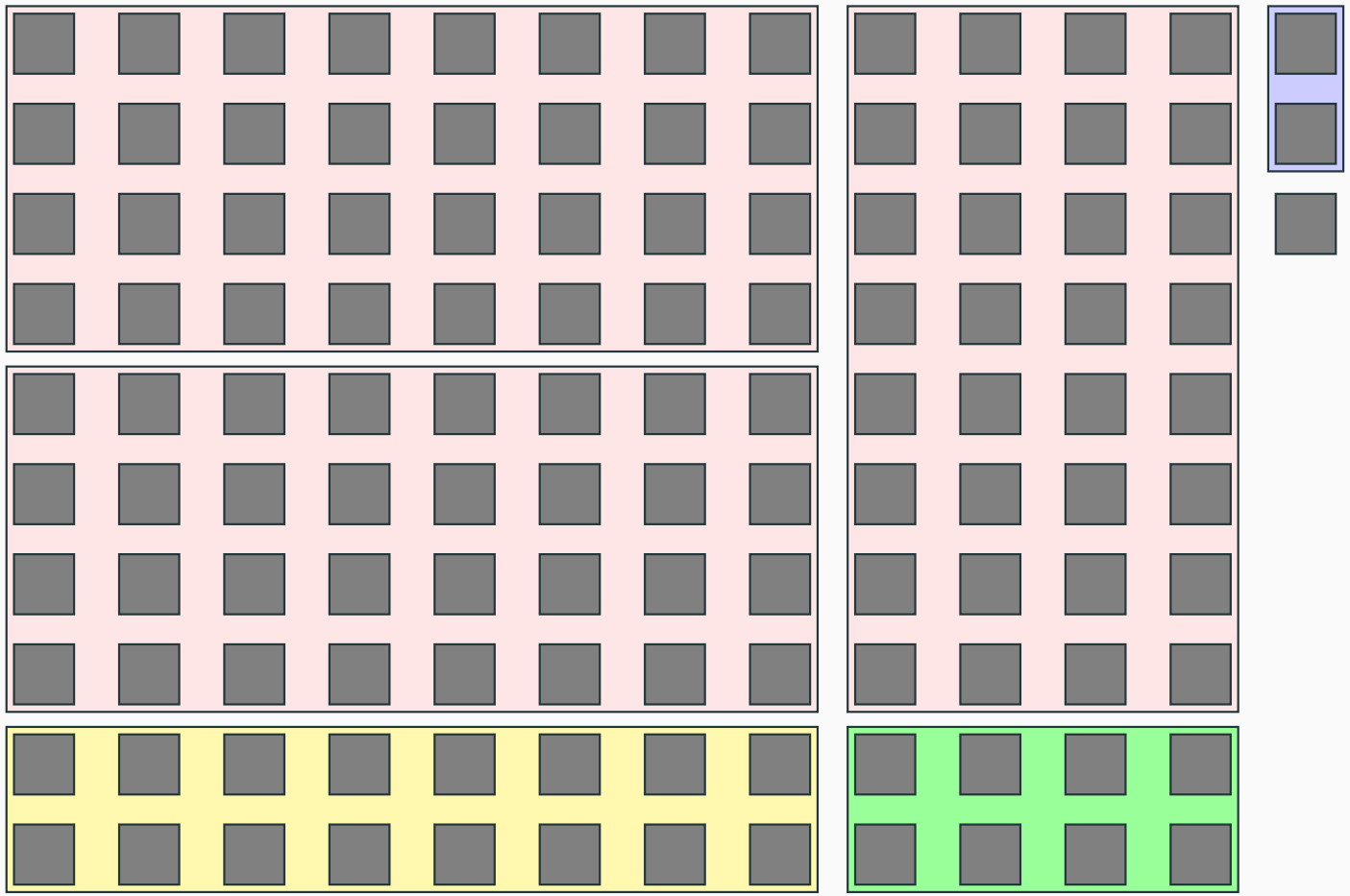
La représentation binaire (base 2)



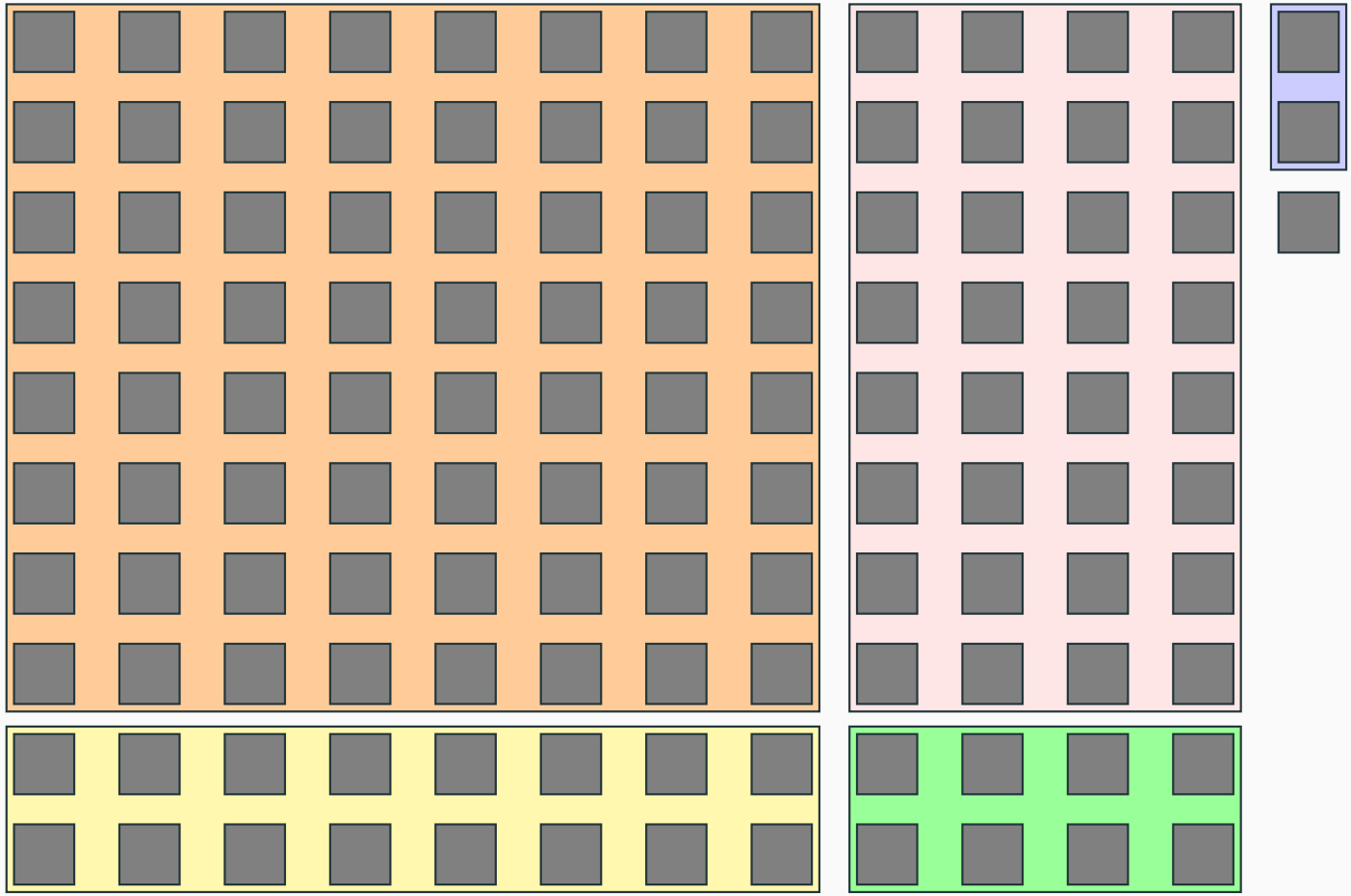
La représentation binaire (base 2)



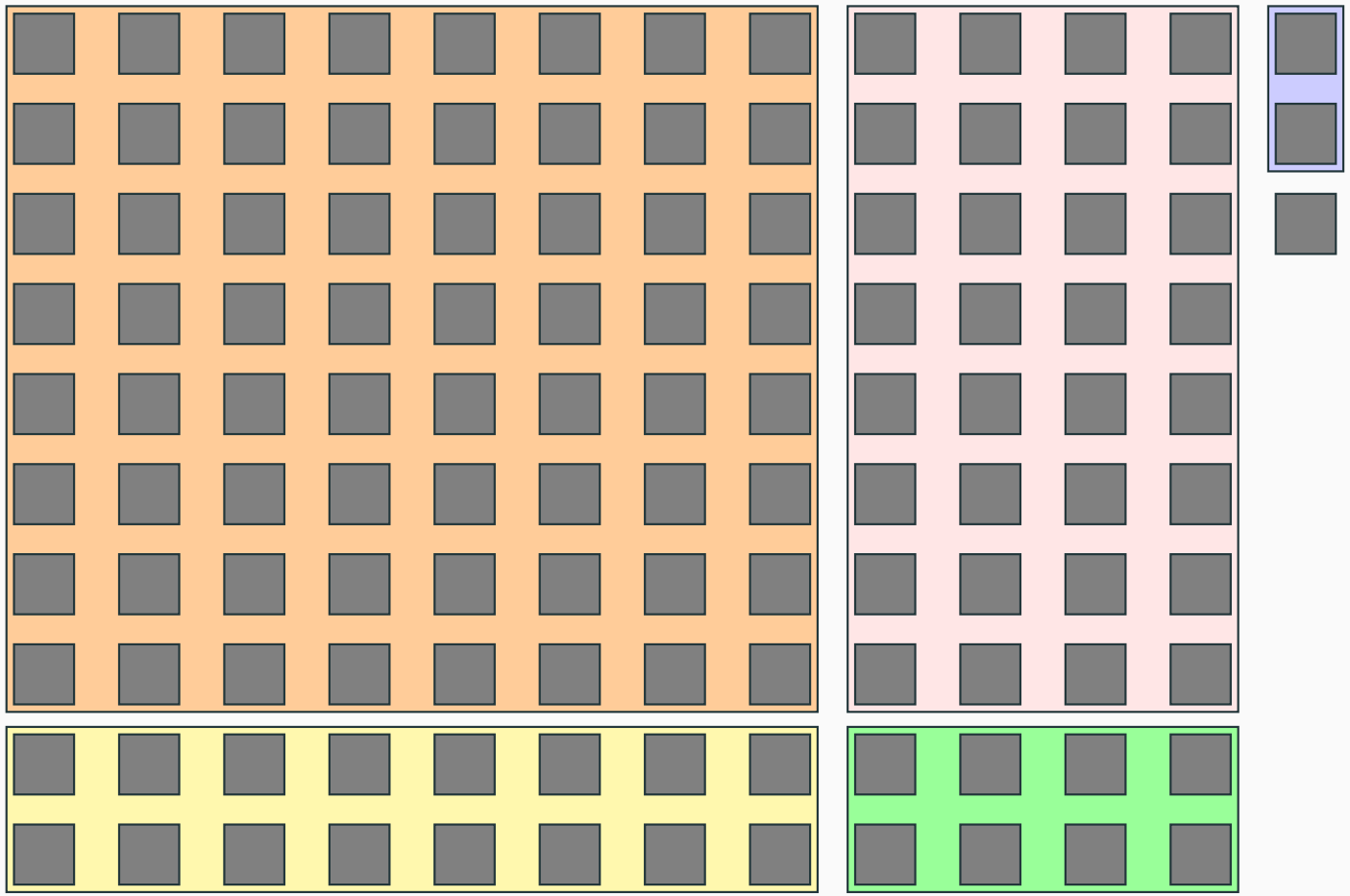
La représentation binaire (base 2)



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)

Un nombre $\mathrm{N} = \sum a_i b^i$ s'écrit en concaténant les chiffres a_i :

a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

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 :

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

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

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

Algorithme de décomposition en base b

123	2		1
1	61	2	

Algorithme de décomposition en base b

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

11

Algorithme de décomposition en base b

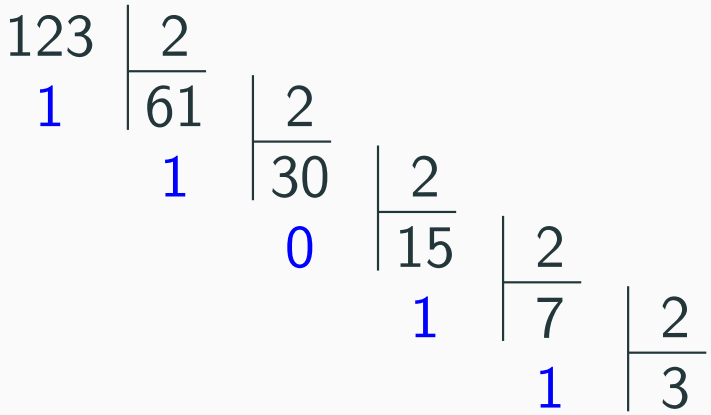
123	2		
1	61	2	
	1	30	2
		0	15

011

Algorithme de décomposition en base b

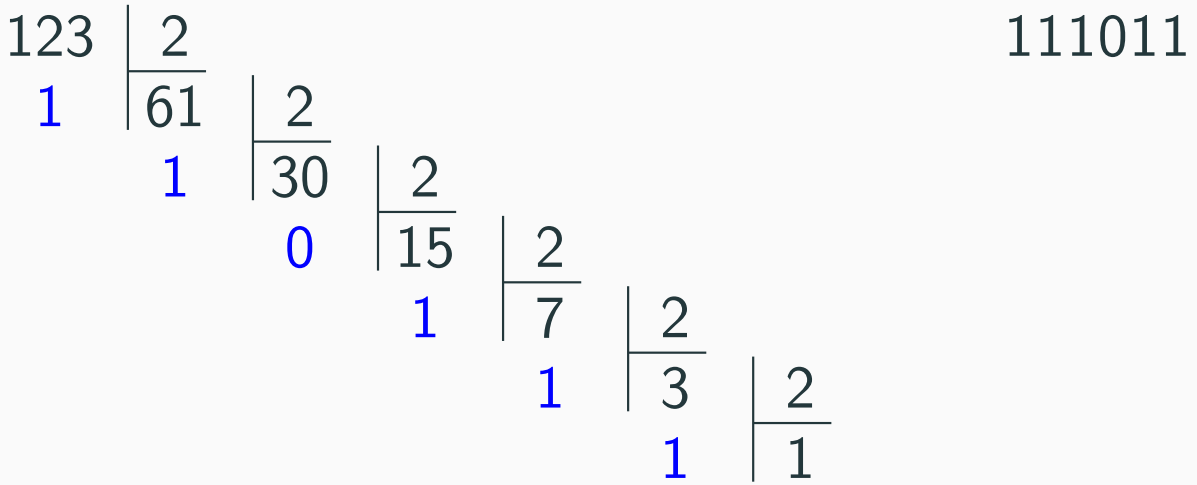
123	2			1011
1	61	2		
	1	30	2	
		0	15	2
			1	7

Algorithme de décomposition en base b

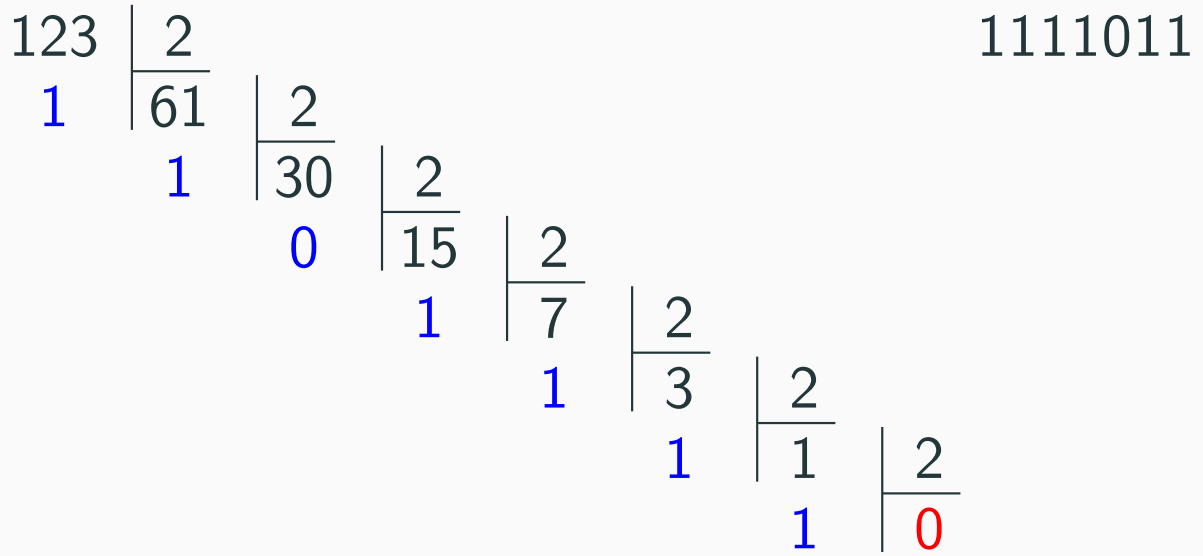


11011

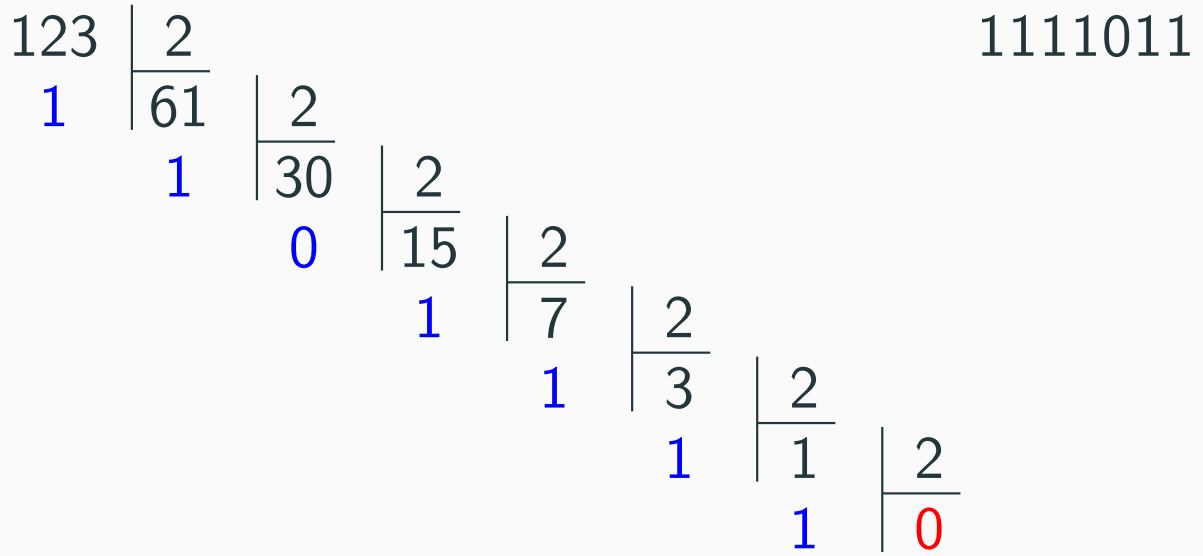
Algorithme de décomposition en base b



Algorithme de décomposition en base b



Algorithme de décomposition en base b




```c
int i = 0;
while (N>0) {
    a[i] = N % b;
    i++;
    N = N / b;
}
```

Recomposition par la méthode de Horner

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

Recomposition par la méthode de Horner

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

```c
int N = 0;
for (int i=p; i>=0; --i) {
    N = b * N + a[i]
}
```

Représentation hexadécimale (base 16)

La notation binaire n'est pas toujours commode.

$$1492_{(10)} \rightarrow 101\,1101\,0100_{(2)}$$

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

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

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

Conversion binaire / hexadécimal

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

$$101\,1101\,0100_{(2)} \rightarrow 5\mathrm{D}4_{(16)}$$

On groupe 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...

Affichages avec C

Pour afficher un **int** en base 10 :

```c
printf("%d", n);
```

En base 16 :

```c
printf("%x", n);
```

Affichages avec C

En binaire, pas de solution immédiate :

```c
void printb(int n) {
    if (n==0) { printf("0"); }
    else if (n==1) { printf("1"); }
    else if (n<0) { printf("-"); printb(-n); }
    else { printb(n/2); printb(n%2); }
}
```

Affichages avec OCaml

On dispose de la fonction `Printf.printf`


```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>

# Printf.printf "En hexadécimal : %x\n" 54321;;
En hexadécimal : d431
- : unit = ()
```

La mémoire d'un ordinateur

La mémoire ne stocke pas les bits isolément mais des *bytes*

En général, les bytes sont des *octets* (groupes de 8 bits)

adresse	mémoire							
	⋮							
4974E	1	0	0	1	1	0	0	1
4974F	0	0	0	1	1	0	1	1
49750	1	1	1	1	1	0	0	0
49751	0	1	1	0	0	1	0	1
49752	1	0	0	0	0	1	1	1
49753	1	1	0	1	0	0	1	0
49754	0	0	1	0	1	0	1	0
	⋮							

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

Note : OCaml réserve un bit pour son propre usage

Boutisme (endianness)

L'ordre des octets en mémoire est délicat.

Par exemple, $51130560_{(10)} = 030C30C0_{(16)}$

Soit en binaire $00000011\ 00001100\ 00110000\ 11000000_{(2)}$

Petit boutiste (« little-endian »)

Par exemple, $51130560_{(10)} = 030C30C0_{(16)}$

Soit en binaire $00000011\ 00001100\ 00110000\ 11000000_{(2)}$

	⋮							
4974E	1	1	0	0	0	0	0	0
4974F	0	0	1	1	0	0	0	0
49750	0	0	0	0	1	1	0	0
49751	0	0	0	0	0	0	1	1
	⋮							

Exemples : Intel x86, x86/64

Grand boutiste (« big-endian »)

Par exemple, $51130560_{(10)} = 030C30C0_{(16)}$

Soit en binaire $00000011\ 00001100\ 00110000\ 11000000_{(2)}$

	⋮							
4974E	0	0	0	0	0	0	1	1
4974F	0	0	0	0	1	1	0	0
49750	0	0	1	1	0	0	0	0
49751	1	1	0	0	0	0	0	0
	⋮							

Exemples : Motorola 68000

Pour le processeur

Chaque solution a ses (légers) avantages :

- petit-boutisme : conversions/arithmétique plus faciles
- gros-boutisme : comparaisons plus faciles

Certaines architectures sont bi-boutistes

Au-delà de la mémoire

Ce problème se retrouve également :

- dans les fichiers « binaires »
- dans les communications

IP → *Network Byte Order* (gros boutiste)

Pour gérer le problème, on dispose :

- de l'instruction « `bswap` » en assembleur x86
- de fonctions « `htonl` » et « `ntohl` »
- ...

Et les entiers négatifs ?

Représentation des entiers naturels sur 4 bits :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Et les entiers négatifs ?

Représentation des entiers naturels sur 4 bits :

On réutilise les 2^{p-1} codes commençant par 1

Le bit de poids fort indiquera le signe ($a_{p-1} = 1$: négatif)

Plusieurs solutions possibles !

Entiers relatifs en signe + magnitude

Entiers relatifs sur 4 bits, signe + magnitude :

0	1	2	3	4	5	6	7	-0	-1	-2	-3	-4	-5	-6	-7
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Hormis le bit de signe, même représentation que les positifs

- ⊕ valeur absolue et négation simples
- ⊖ deux zéros !
- ⊖ opérations arithmétiques complexes

Entiers relatifs en complément à 1

Entiers relatifs sur 4 bits, complément à 1 :

0	1	2	3	4	5	6	7	-7	-6	-5	-4	-3	-2	-1	-0
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

$n < 0$ est représenté par le complément de $-n$

- ⊕ valeur absolue et négation simples
- ⊕ opérations arithmétiques moins problématiques
- ⊖ toujours deux zéros

Entiers relatifs en complément à 2

Entiers relatifs sur 4 bits, complément à 2 :

0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

$n < 0$ est représenté par le complément de $-n-1$

- ⊕ opérations arithmétiques similaires aux naturels
- ⊕ une seule représentation pour zéro
- ⊖ valeur absolue et négation un peu plus complexes

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 $2^{31}-1 \simeq 2{,}1 \times 10^9$
- 64 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).

Interprétations

Tout se passe comme si la seconde moitié des représentations :

0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

se trouvait en fait de l'autre côté :

-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111

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

$$\mathrm{N} = \sum_{i=0}^{p-2} a_i 2^i + a_{p-1}\left(-2^{p-1}\right)$$

Codage et décodage en complément à 2

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

- coder $n + 2^p$ comme un entier positif sur p bits
- coder $|n-1|$ sur p bits puis inverser tous les bits
- coder $|n|$ sur p bits, inverser tous les bits et ajouter 1

Et inversement pour le décodage si $a_{p-1} = 1$!

Exemple de codage en complément à deux

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 !

Le programme doit savoir comment interpréter les données...

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 $17 + 105$:

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

Exemples d'additions de signes différents (résultat négatif)

Pour calculer 17 + (-37) :

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

Pour calculer 42 + (-94) :

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

Exemples d'additions de signes différents (résultat négatif)

Pour calculer 17 + (-37) :

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

Pour calculer 42 + (-94) :

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

Exemples d'additions de signes différents (résultat négatif)

Pour calculer 17 + (-37) :

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

Pour calculer 42 + (-94) :

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

Exemples d'additions de signes différents (résultat négatif)

Pour calculer 17 + (-37) :

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

Pour calculer 42 + (-94) :

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

Exemples d'additions de signes différents (résultat négatif)

Pour calculer 17 + (-37) :

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

Pour calculer 42 + (-94) :

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

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

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

Pour calculer 105 + (-37) :

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

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 1\,00001100 \end{array}$$

Pour calculer 105 + (-37) :

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

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = 1 & 00001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

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

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 1\,00001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & 1\,01000100 \end{array}$$

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 1\,00001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & 1\,01000100 \end{array} \rightarrow 68$$

Exemples d'additions de signes différents (résultat positif)

Pour calculer 17 + (-5) :

$$\begin{array}{rr} & 00010001 \\ + & 11111011 \\ \hline = & 1\,00001100 \end{array} \rightarrow 12$$

Pour calculer 105 + (-37) :

$$\begin{array}{rr} & 01101001 \\ + & 11011011 \\ \hline = & 1\,01000100 \end{array} \rightarrow 68$$

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

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = & \end{array}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = 1 & 11010110 \end{array}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = 1 & 11010110 \end{array} \rightarrow \text{-42}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = & \end{array}$$

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = 1 & 11010110 \end{array} \rightarrow \text{-42}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = 1 & 10010001 \end{array}$$

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = 1 & 11010110 \end{array} \rightarrow \text{-42}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = 1 & 10010001 \end{array} \rightarrow \text{-111}$$

Exemples d'additions d'entiers négatifs

Pour calculer (-37) + (-5) :

$$\begin{array}{rr} & 11011011 \\ + & 11111011 \\ \hline = 1 & 11010110 \end{array} \rightarrow \text{-42}$$

Pour calculer (-17) + (-94) :

$$\begin{array}{rr} & 11101111 \\ + & 10100010 \\ \hline = 1 & 10010001 \end{array} \rightarrow \text{-111}$$

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

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & \end{array}$$

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

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

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & 10010011 \end{array}$$

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

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

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

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

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

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

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

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = 1 & 01111101 \end{array}$$

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

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

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = 1 & 01111101 \end{array} \rightarrow 125$$

Exemples d'additions d'entiers (résultat non représentable)

Pour calculer 42 + 105 :

$$\begin{array}{rr} & 00101010 \\ + & 01101001 \\ \hline = & 10010011 \end{array} \rightarrow \text{-109}$$

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

$$\begin{array}{rr} & 11011011 \\ + & 10100010 \\ \hline = 1 & 01111101 \end{array} \rightarrow 125$$

Les résultats sont incorrects (dont mauvais signe)

Débordements

Il y a **débordement** lorsque le résultat d'un calcul n'est pas représentable

Même une division entière peut déborder :

Débordements

Il y a **débordement** lorsque le résultat d'un calcul n'est pas représentable

Même une division entière peut déborder :

`INT_MIN` / -1 si en complément à 2 sans trap

Processeur et débordements

En l'absence de débordement,

on peut utiliser les mêmes circuits pour

- les additions d'entiers naturels
- les additions d'enters relatifs en complément à 2

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

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

$$0+0=0$$

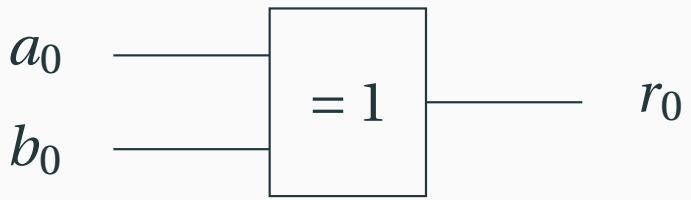
$$0+1=1$$

$$1+0=1$$

$$1+1=10$$

Additions binaires et électronique

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



$$0+0=0$$

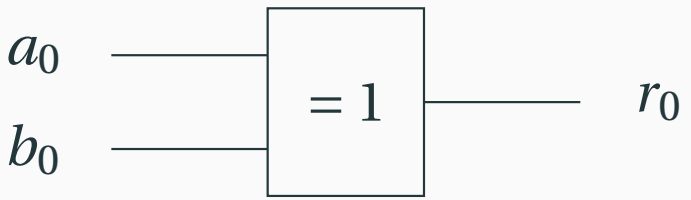
$$0+1=1$$

$$1+0=1$$

$$1+1=10$$

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

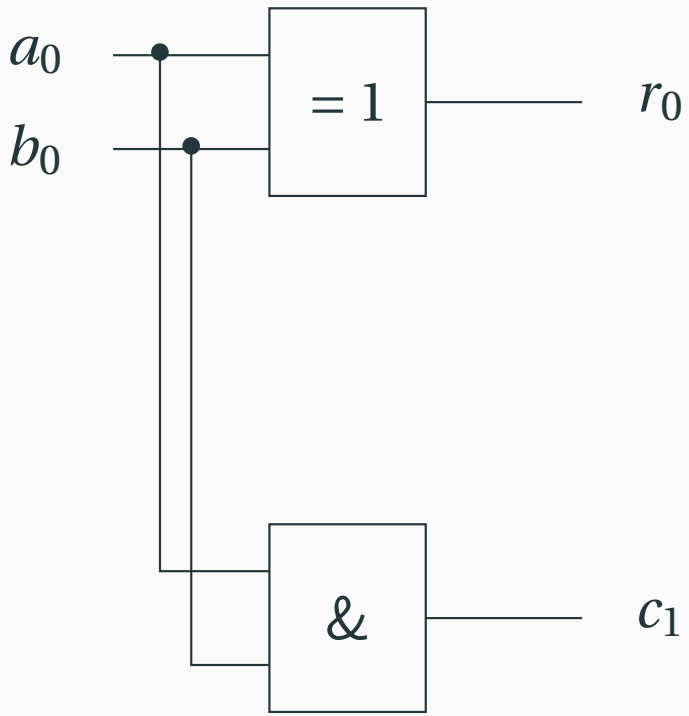
$$
\begin{array}{cccc}
 & & c_1 & \\
 & & a_1 & a_0 \\
+ & & b_1 & b_0 \\
\hline
= & r_2 & r_1 & r_0
\end{array}
$$

$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1=10$$



Additions binaires et électronique

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

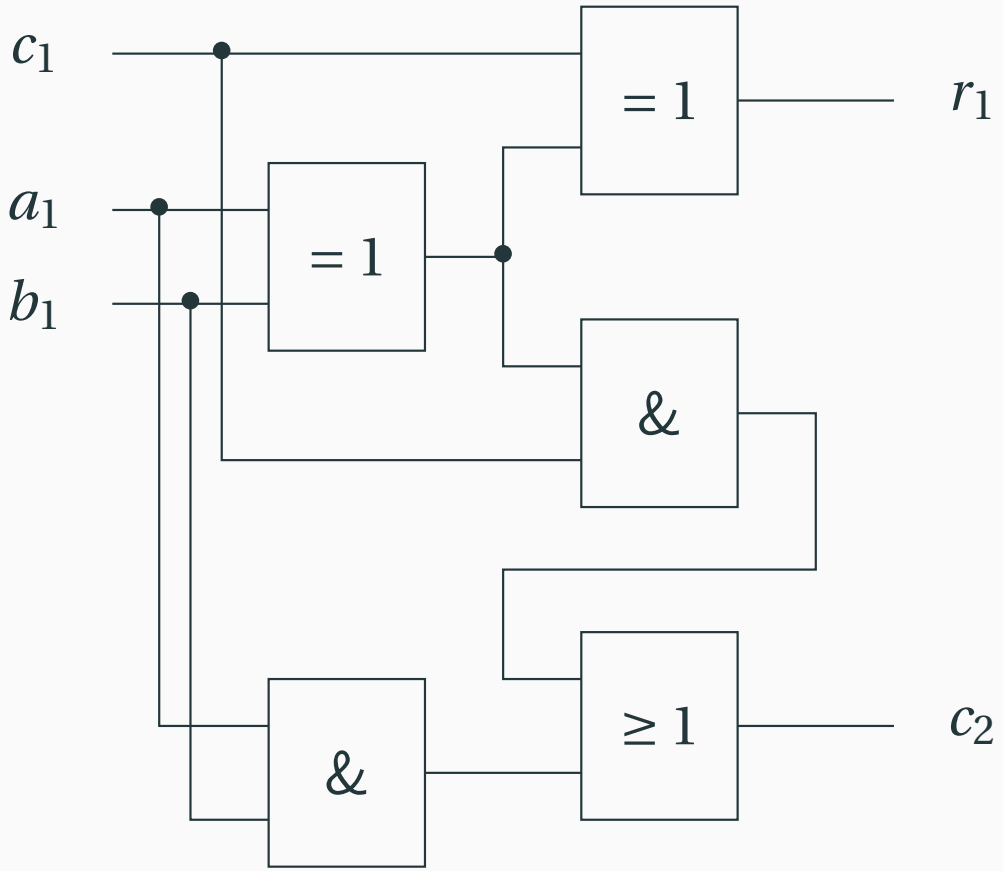
$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1=10$$

$$1+1+1=11$$



Processeur et débordements

S'il y a débordement, cela dépend du processeur :

- arithmétique modulaire (on ignore les bits supplémentaires)

  Sommer de deux entiers positifs peut donner un négatif

  Sommer de deux entiers négatifs peut donner un positif

- arithmétique à saturation...

Dans le processeur

Le processeur peut détecter ces débordements

Une *même* instruction peut servir à additionner

- deux entiers naturels
- deux entiers relatifs

Le débordement dépend de l'interprétation des données !

Du côté des langages

OCaml n'utilise qu'un type d'entiers (relatifs en complément à 2)

Il ignore les débordements (arithmétique modulaire)

Le langage C possède des dizaines de types entiers

Pour gérer les architectures qui n'utilisent pas le complément à 2 ou l'arithmétique modulaire, les débordements sont UB pour les `int` !

Python 3 utilise des entiers de taille arbitraire

Il ajoute des bits lorsque c'est nécessaire

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.

Le cas d'Ariane 5

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...

Le cas d'Ariane 5

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 l'inclinaison, ingérable, nécessite une autodestruction du lanceur.

L'arbre qui cache la forêt

L'exemple d'Ariane 5 laisse penser que c'est exceptionnel

Il n'en est rien !

C'est un problème incroyablement répandu

Cela touche *la plupart* des programmes !

Plus récent, MS Exchange

Format YYMMDDHHMM pour stocker des dates dans un entier

22 novembre 2017, 10h23 → 1711221023

Plus récent, MS Exchange

Format YYMMDDHHMM pour stocker des dates dans un entier

22 novembre 2017, 10h23 → 1711221023

Dans un entier signé 32 bits...

Plus récent, MS Exchange

Format YYMMDDHHMM pour stocker des dates dans un entier

22 novembre 2017, 10h23 → 1711221023

Dans un entier signé 32 bits...

... tout est cassé le 1er janvier 2022 à minuit !

Le bug de l'an 2000

De l'intérêt de réfléchir aux solutions choisies :

La délicate question des dates

Temps selon POSIX : secondes depuis 1er janvier 1970

entier 32 bits signé : 19 janvier 2038 à 3h14 et 8 secondes

La gestion des dates, heures et durées est très complexe

Petit tour d'horizon de la jungle des entiers en C

Important : **très peu de ce qui suit est exigible en concours !**

Les entiers en C

Le module `stdint.h` fournit des types entiers de taille fixée

On peut typiquement trouver les types signés suivants :

`int8_t` `int16_t` `int32_t` `int64_t`

On peut également trouver les types non-signés suivants :

`uint8_t` `uint16_t` `uint32_t` `uint64_t`

Les entiers en C

Il peut également fournir des types assurant une taille *minimale*

Les plus rapides possibles :

`int_fast8_t` `uint_fast8_t` ...

Les plus petits (en taille mémoire) possibles :

`int_least8_t` `uint_least8_t` ...

Les entiers en C

Spécifier la taille ne permet pas au compilateur de choisir le type le plus efficace pour l'architecture considérée

Il existe des types génériques

Le type char

Le type **char** désigne un entier de taille 1 *byte*

byte = plus petit entier adressable (**sizeof**(**char**)=1)

Souvent un octet, mais peut être davantage

Le type **char** peut être signé ou non

Le type char

On dispose aussi de deux types de même taille que **char**

l'un signé, **signed char**, contient au moins $[\![0\,..\,255]\!]$

l'autre non signé, **unsigned char**, contient au moins $[\![-127\,..\,127]\!]$

L'implémentation associe librement **char** à l'un ou l'autre

Les types signés

On dispose ensuite des types signés suivants :

- **short**, qui contient au moins $[\![-32767\,..\,32767]\!]$
- **int**, qui contient au moins $[\![-32767\,..\,32767]\!]$
- **long**, qui contient au moins $[\![-2^{31}+1\,..\,2^{31}-1]\!]$
- **long long**, qui contient au moins $[\![-2^{63}+1\,..\,2^{63}-1]\!]$

Chaque représentation contient au moins la précédente

int est *a priori* la représentation la plus rapide

Les types signés

Il est possible de préciser « `signed` » avant et « `int` » après

Cela correspond au même type

Par exemple, « `signed short` », « `short int` » et
« `signed short int` » sont des alias de `short`

Les types signés

Pour les signés :

- la représentation binaire est non spécifiée (certains codes peuvent être illégaux, « *traps* »)
- les débordements sont UB

On veut pouvoir utiliser les opérations du processeur !

Les types non signés

On dispose de même des types non signés suivants :

- **`unsigned short`**, qui contient au moins $[\![0\,..\,65535]\!]$
- **`unsigned int`**, qui contient au moins $[\![0\,..\,65535]\!]$
- **`unsigned long`**, qui contient au moins $[\![0\,..\,2^{32}-1]\!]$
- **`unsigned long long`**, qui contient au moins $[\![0\,..\,2^{64}-1]\!]$

Chaque représentation contient au moins la précédente

On peut également ajouter « **`int`** » derrière

Les types non signés

Pour les non signés sur p bits :

- entiers entre 0 et $2^p - 1$
- arithmétique modulo 2^p

Principal but : pouvoir utiliser la représentation binaire

Opérations sur les types non signés

On dispose d'opérateurs unaires :

- `~` effectue un complément à 1 ($n \mapsto 2^p - 1 - n$)
- `-` effectue un complément à 2 ($n \mapsto 2^p - n$)

D'opérateurs binaires logiques :

- `&` effectue un « et » binaire bit à bit
- `|` effectue un « ou » binaire bit à bit
- `^` effectue un « ou exclusif » binaire bit à bit

Opérations sur les types non signés

On dispose aussi d'opérateurs binaires arithmétiques :

a<<b donne $a \times 2^b \pmod{2^p}$

a>>b donne $\left\lfloor \frac{a}{2^b} \right\rfloor$

Revient à décaler de b rangs vers la gauche (resp. la droite) la représentation binaire de a (en complétant avec des zéros)

Remarque : a<<b est UB si b< 0 ou b$\geqslant$p

Et sur les signés ?

À fuir !

a<<b et a>>b sont UB si b est négatif ou b$\geqslant p$

a<<b est UB si a est négatif ou a<<b déborde

a>>b est laissé au choix de l'implémentation si a négatif

a&b, a|b et a^b opèrent *a priori* sur les bits...
... mais comme la représentation n'est pas imposée

- le résultat est incertain
- UB si débordement (écritures binaires illégales)
- le cas de -0 n'est pas limpide

Pour les calculs en C

Considérons à présent un calcul tel que :

```c
c = a + b;
```

Pour les calculs en C

Considérons à présent un calcul tel que :

```c
c = a + b;
```

a et b doivent être mis dans un format commun

Pour les calculs en C

Considérons à présent un calcul tel que :

```c
c = a + b;
```

a et b doivent être mis dans un format commun

Le résultat sera converti dans le type de c

Pour les calculs en C

Considérons à présent un calcul tel que :

```c
c = a + b;
```

a et b doivent être mis dans un format commun

- a et b subissent possiblement une promotion
- puis une conversion dans un format commun

Le résultat sera converti dans le type de c

Rangs

On définit des « rangs » pour les types numériques

`_Bool`

- $\prec$ `char` / `unsigned char` / `signed char`
  - $\prec$ `unsigned short` / `short`
    - $\prec$ `unsigned int` / `int`
      - $\prec$ `unsigned long` / `long`
        - $\prec$ `unsigned long long` / `long long`

Règles de promotion

Les arguments de rang strictement inférieur à `int`

(soit `_Bool`, `char`, `short`, signés ou non) sont promus

- en `int` si c'est possible sans perte
- en `unsigned int` sinon

Les valeurs ne peuvent pas changer lors de la promotion

Règles de choix de la représentation commune

Après promotion

- même type → pas de conversion nécessaire
- deux signés → type de rang le plus élevé
- deux non-signés → type de rang le plus élevé
- non-signé de rang supérieur ou égal au rang du signé

  → conversion vers le type non-signé
- le type signé inclue le type non-signé

  → conversion vers le type signé
- sinon conversion des *deux arguments* vers le type non-signé « correspondant » au type de l'argument signé

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	
`unsigned int`	`unsigned int`	
`short`	`short`	
`unsigned short`	`unsigned short`	
`char`	`char`	
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	
`short`	`short`	
`unsigned short`	`unsigned short`	
`char`	`char`	
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	
`unsigned short`	`unsigned short`	
`char`	`char`	
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	
`char`	`char`	
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	`int`
`unsigned int`	`int`	
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	`int`
`unsigned int`	`int`	`unsigned int`
`unsigned long`	`int`	
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	`int`
`unsigned int`	`int`	`unsigned int`
`unsigned long`	`int`	`unsigned long`
`unsigned short`	`int`	
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	`int`
`unsigned int`	`int`	`unsigned int`
`unsigned long`	`int`	`unsigned long`
`unsigned short`	`int`	`int` / `unsigned int` ?
`unsigned int`	`long`	

Exemples

opérande 1	opérande 2	convertis en
`int`	`int`	`int`
`unsigned int`	`unsigned int`	`unsigned int`
`short`	`short`	`int`
`unsigned short`	`unsigned short`	`int` / `unsigned int` ?
`char`	`char`	`int` / `unsigned int` ?
`signed char`	`signed char`	`int`
`unsigned int`	`int`	`unsigned int`
`unsigned long`	`int`	`unsigned long`
`unsigned short`	`int`	`int` / `unsigned int` ?
`unsigned int`	`long`	`long` / `unsigned long` ?

Conversion finale

Le résultat est enfin converti :

Vers `_Bool`, `0` → `false`, sinon `true`

Aucun problème possible

Vers un **`unsigned`**, on rajoute/retire `MAX+1` jusqu'à ce que ça rentre

Arithmétique modulaire, toujours valable

Vers un **`signed`**, si c'est représentable, super !

Sinon, le résultat dépend des choix de l'implémentation

Et les constantes (littéraux) ?

Préfixes : `0` (octal), `0x` (hexadécimal)

Suffixes : `U` (unsigned), `L` (long), `LL` (long long)

Constante	Type choisi parmi :
`1234`	**`int`**, **`long`**, **`long long`**
`0x5D4`	**`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**, ...
`1234U`	**`unsigned int`**, **`unsigned long`**, **`unsigned long long`**
`0x5D4U`	**`unsigned int`**, **`unsigned long`**, **`unsigned long long`**
`1234L`	**`long`**, **`long long`**
`1234UL`	**`unsigned long`**, **`unsigned long long`**

Note : le signe « - » ne fait pas partie de la constante !

Autres entiers

`size_t` : non signé, capable de contenir une taille

C'est le type retourné par `sizeof`

Rétrospectivement, c'était une mauvaise idée

`ptrdiff_t` : signé, capable de contenir le nombre de « cases » entre deux pointeurs de même type

Autres entiers

On est tenté d'écrire :

```c
int foo(int* tab, size_t length) {
    for (size_t i = length; i>=0; --i) {
        // faire quelque chose avec tab[i]
    }
    return ...;
}
```

Autres entiers

On est tenté d'écrire :

```c
int foo(int* tab, size_t length) {
    for (size_t i = length; i>=0; --i) {
        // faire quelque chose avec tab[i]
    }
    return ...;
}
```

La boucle ne s'arrête jamais !

Entiers dans le cadre du programme

`int8_t`, `int32_t`, `int64_t` et `int`

`uint8_t`, `uint32_t`, `uint64_t` et `unsigned int`

Littéraux *sans suffixes* et sans souci de typage

`+` `-` `*` `/` et `%` (opérandes positifs pour ce dernier)

Savoir qu'il faut éviter les débordements

Entiers dans le cadre du programme

`bool`, uniquement pour des booléens, sans conversion

`!` `&&` `||`

`char`, utilisé uniquement dans le cadre de chaînes

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, même chose, avec des 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 :

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

$$21.9 = 10101.11100011001100\ldots_{(2)}$$

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 :

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

$$21.9 = 10101.11100011001100..._{(2)}$$

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 cela dans 32 ou 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 :

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

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

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

Par exemple,

$$\begin{aligned} 21.625 &= 10101.101_{(2)} \\ &= +1.0101101_{(2)} \times 2^4 \\ &= +1.0101101_{(2)} \times 10_{(2)}^{100_{(2)}} \end{aligned}$$

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
0	10000011	01011010000000000000000

Et zéro ?

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

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 corrects pour $+$, $-$, $\times$, $\div$, $\sqrt{}$) ;
- différentes règles d'arrondis.

Pourquoi une norme ?

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

  ($x+y$ différent de $y+x$, $0.5\times x$ différent de $x/2.0$, etc.)

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 $\simeq$ 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, $0.3 = 1.\overline{0011}_{(2)} \times 2^{-2} \simeq 1.0011001100110011... \times 2^{-2}$

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'intervalle $[0.10229999945, 0.1023000069]$ 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 flottante

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$+(2^{11-1}-1)$) ;
- 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 $2{,}23 \times 10^{-308}$).

≡ généralement au type `double` en C, au type `float` en OCaml

Calculs en arithmétique flottante

Les résultats des calculs sont mémorisés comme des flottants.

Ils vont donc être arrondis à la plus proche valeur représentable !

Les valeurs décimales fournies par l'utilisateur le sont aussi.

En général, l'affichage rend les calculs apparamment corrects :

```
# 0.1 +. 0.1;;
- : float = 0.2
```

Conséquences sur les calculs

On peut cependant remarquer le mécanisme d'arrondi :


```
# 0.10229999999999997;;
- : float = 0.10229999999999997

# 0.10229999999999998;;
- : float = 0.10229999999999997

# 0.102299999999999951;;
- : float = 0.10229999999999999

# 0.10229999999999996;;
- : float = 0.1023
```

Conséquences sur les calculs

Certains calculs peuvent sembler imprécis, voire incorrects :


```
# 0.1 +. 0.2;;
- : float = 0.30000000000000004

# 0.1 *. 3.0;;
- : float = 0.30000000000000004

# 0.1 +. 0.2 = 0.3;;
- : bool = false

# 0.1 *. 3.0 = 0.3;;
- : bool = false
```

S'explique en considérant les arrondis successifs

0.09999999999999917

0.10000000000000056 $\leftarrow 0.1$

0.10000000000000194

...

0.19999999999999983

0.20000000000000011 $\leftarrow 0.2$

0.20000000000000039

...

0.29999999999999934

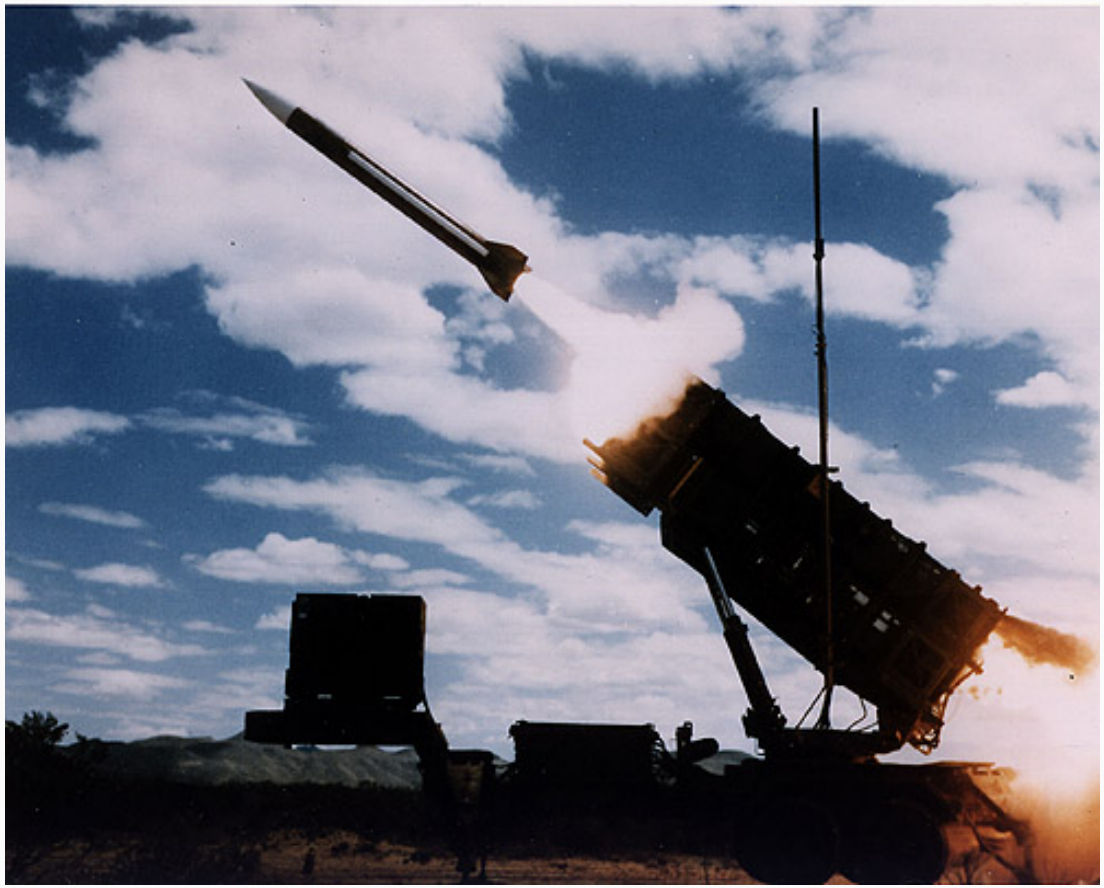
0.29999999999999989 $\leftarrow 0.3$

0.30000000000000044 $\leftarrow 0.1 + 0.2$ ou 0.1×3

0.30000000000000099

Dans la vie réelle

Durant la première guerre du golfe, des missiles américains *Patriots* associés à un système de suivi radar performant ont eu pour tâche d'intercepter les missiles irakiens *Scud*.



Bug du système

Le 25 février 1991, le système radar ne parvient pas à lancer convenablement des Patriots contre un Scud visant des barraquements américains

Le missile Scud fait 28 morts et 98 blessés

L'analyse de l'incident a montré qu'une imprécision de calcul a provoqué l'incapacité du radar à suivre le Scud

(cela étant, en pratique, l'efficacité des missiles anti-missiles, même sans bug, était et reste extrêmement limité)

Origine de l'erreur de calcul

Le radar compte les dixièmes de seconde écoulés (n, entier)

Origine de l'erreur de calcul

Le radar compte les dixièmes de seconde écoulés (n, entier)

Pour le suivi, n $\curvearrowright 0.1 \times$n, flottant sur 24 bits

Origine de l'erreur de calcul

Le radar compte les dixièmes de seconde écoulés (n, entier)

Pour le suivi, n $\curvearrowright 0.1 \times$n, flottant sur 24 bits

0.1 non représentable $\longmapsto$ 95 ns d'erreur pour chaque 0,1 s

Origine de l'erreur de calcul

Le radar compte les dixièmes de seconde écoulés (n, entier)

Pour le suivi, n $\curvearrowright 0.1 \times$n, flottant sur 24 bits

0.1 non représentable $\longmapsto$ 95 ns d'erreur pour chaque 0,1 s

Après 100 h, erreur de 0,34 s

Origine de l'erreur de calcul

Le radar compte les dixièmes de seconde écoulés (n, entier)

Pour le suivi, n $\curvearrowright 0.1 \times$n, flottant sur 24 bits

0.1 non représentable $\longmapsto$ 95 ns d'erreur pour chaque 0,1 s

Après 100 h, erreur de 0,34 s

Pour $v = 1676$ m/s, 573 m d'erreur !

Correction du bug

Une modification de l'algorithme de conversion entier→flottant permet de s'affranchir de cette erreur

Malheureusement, le correctif est arrivé un jour trop tard

Les israéliens avaient noté et signalé une imprécision du radar de 20% après 8 h de fonctionnement en continu

Les américains pensaient que le problème n'était pas critique car le système devait être réinitialisé régulièrement (cela prend moins d'une minute), remettant l'erreur à zéro

Absorption

Les petites valeurs peuvent être absorbées par les grandes :


```
# let a = 6.022e23 and b = 123456.789;;
val a : float = 6.022e+023
val b : float = 123456.789

# a +. b;;
- : float = 6.022e+023

# a +. b = a;;
- : bool = true
```

Absorption

Un exemple d'absorption partielle en simple précision :

```
  10101010110010101 0.01011 0000000000000000
+                         11.00101011101001101010 11
= 10101010110010110 00.10000 1110100110101011
```

Un exemple d'absorption totale en simple précision :

```
  10101010110010101 0.01011 0000000000000000000000000
+                           0.00000011101001101010110110 1001
= 10101010110010101 0.01011 0111010011010101101101001
```

Cancellation

On peut avoir une perte de précision lors de la soustraction de valeurs proches :


```
# let a = 30000000000.888889
  and b = 30000000000.111111;;
val a : float = 30000000000.888889
val b : float = 30000000000.111111

# a +. b;;
- : float = 60000000001.

# a -. b;;
- : float = 0.7777862548828125
```

Cancellation

Un exemple de cancellation en simple précision :

```
  10101010110010101 0.01011
-  10101010110010100 11.00001
=                   10.01010 00000000000000000
```

On ajoute des bits supplémentaires (des zéros) qui ne sont pas nécessairement les bons !

Cancellation catastrophique

Le résultat peut être catastrophique si les deux valeurs ont déjà subi des arrondis (par exemple par absorption) :


```
# let a = 6.022e23 and b = 123456.789;;
val a : float = 6.022e+023
val b : float = 123456.789

# a +. b -. a;;
- : float = 0.

# 1. /. (a +. b -. a);;
- : float = infinity
```

Calcul d'un terme d'une suite

Considérons la suite récurrente suivante :

$$u_0 = \mathrm{e}\,;$$

$$u_n = (u_{n-1} - 1) \times n \text{ pour } n \geq 1.$$

Que vaut u_{50} ?

Calcul itératif

On peut calculer itérativement les termes de la liste :


```
# let u = ref (exp 1.0);;
val u : float ref = {contents = 2.7182818284590451}

# for i = 1 to 50 do
      u := (!u -. 1.0) *. float_of_int i
  done;;
- : unit = ()

# !u;;
- : float = -4.3968039301820685e+048
```

Calcul direct

On peut cependant montrer que

$$u_{50} = 50! \times \left(u_0 - \sum_{k=0}^{49} \frac{1}{k!} \right)$$

Essayons...

```
# let rec fact n =
    if n = 0 then 1.0 else
      float_of_int n *. fact (n - 1);;
val fact : int -> float = <fun>
```

Calcul direct

$$u_{50} = 50! \times \left(u_0 - \sum_{k=0}^{49} \frac{1}{k!} \right)$$

```
# let s = ref 0.0;
- : unit = ()

# for i = 0 to 49 do s := !s +. 1.0 /. fact i done;;
val s : float ref = {contents = 0.}

# fact 50 *. (exp 1.0 -. !s);;
- : float = -1.3506570618255054e+049
```

Ce n'est pas le même résultat...

Creusons encore...

On sait, par ailleurs, que

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots = \sum_{k=0}^{\infty} \frac{1}{k!} \qquad \text{donc} \qquad u_{50} = 50! \times \sum_{k=50}^{\infty} \frac{1}{k!}$$

Cela donne...


```
# let s = ref 0.0;;
val s : float ref = {contents = 0.}

# for i = 50 to 1000 do s := !s +. 1.0 /. fact i done;;
- : unit = ()

# fact 50 *. !s;;
- : float = 1.0199921658366668
```

C'est cette fois-ci complètement différent !

Creusons encore...

Ce n'est pas parce qu'on a arrêté la somme à 1000 :

```
# 1. /. fact 200;;
- : float = 0.
```

Au bilan

On a donc trois résultats différents pour le même calcul :

- -4.3968039301820685e+48
- -1.3506570618255054e+49
- 1.019992165836668

Le dernier résultat est quasiment la bonne valeur.

La différence entre les deux premiers est causée par des erreurs d'arrondis successifs.

La différence entre le troisième et les deux autres par le fait que « exp `1.0` » n'est pas tout à fait égal à e !

Influence de u_0

Pour s'en convaincre :


```
# let s = ref 0.0;;
val s : float ref = {contents = 0.}

# for i = 0 to 49 do s := !s +. 1.0 /. fact i done;;
- : unit = ()

# fact 50 *. (exp 1.0 -. !s);;
- : float = -1.3506570618255054e+049

# fact 50 *. (exp 1.0 +. 2.** -52. -. !s);;
- : float = 0.

# fact 50 *. (exp 1.0 +. 2.** -50. -. !s);;
- : float = 1.3506570618255054e+049
```

La série harmonique

La série harmonique (u_n) est définie par

$$u_n = \frac{1}{n}$$

On définit classiquement la n-ième somme partielle H_n

$$\mathrm{H}_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

$$\mathrm{H}_n = \ln(n) + \gamma + \mathrm{O}(1) \xrightarrow[n\to\infty]{} +\infty$$

Calcul des sommes partielles

On peut vouloir calculer H_n de la façon suivante :


```
# let h n =
    let s = ref 0. in
    for k = 1 to n do
      s := !s +. 1.0 /. float_of_int k
    done;
    !s;;
val h : int -> float = <fun>
```

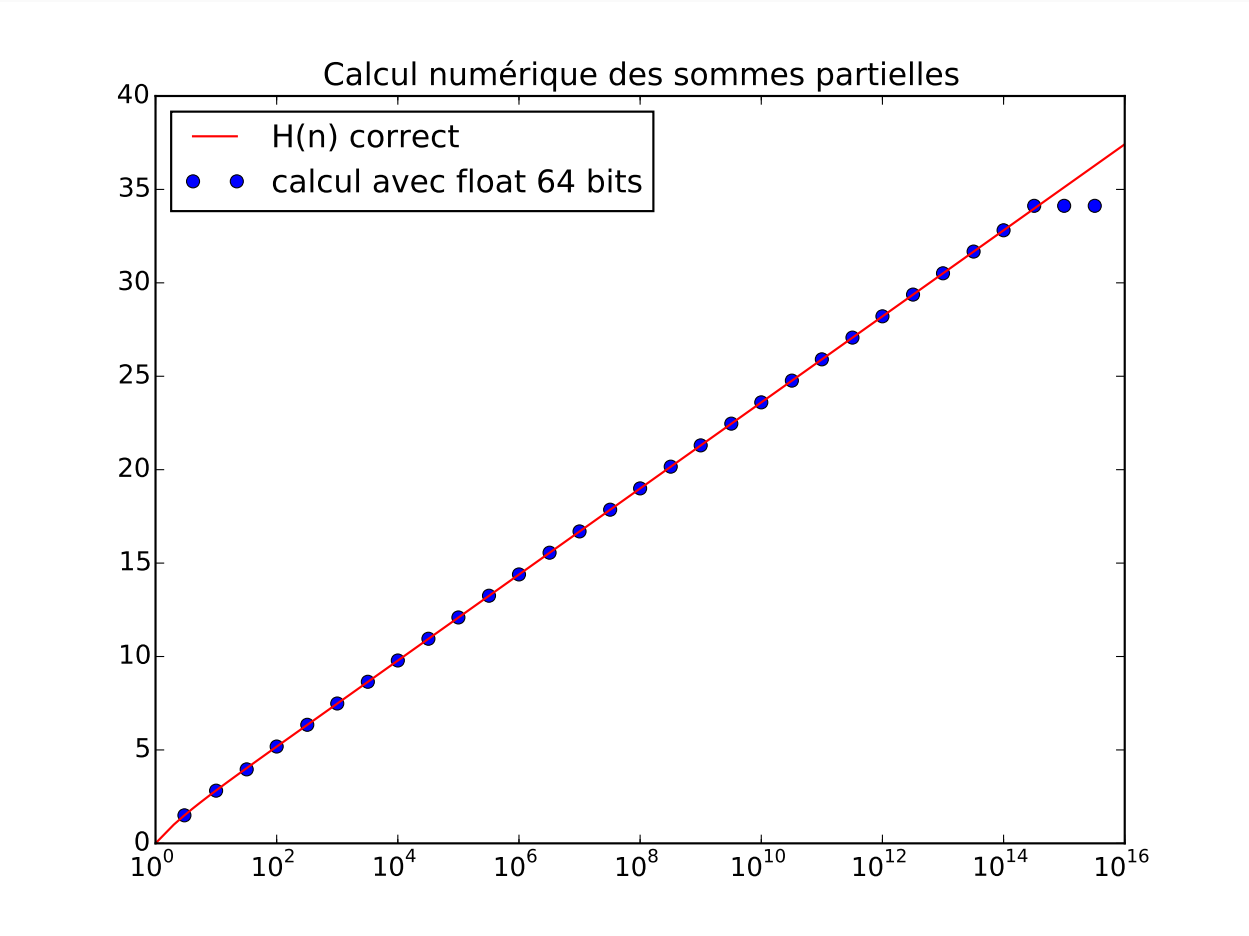
Problème : h n ne tend pas vers $+\infty$

En effet, $1/k$ finit par être absorbé par s !

En double précision, lorsque $\frac{s}{1/k}$ atteint 2^{52} soit $k \simeq 10^{14}$.

La série harmonique

La série, calculée de cette façon, finit par converger :



Calcul des sommes partielles

Changer l'ordre de sommation change le résultat...


```
# let hd n =
    let s = ref 0. in for k = n downto 1 do
      s := !s +. 1.0 /. float_of_int k
    done; !s;;
val hd : int -> float = <fun>

# hd 10000000000;;
- : float = 23.603066594888269

# h  10000000000;;
- : float = 23.6030665949975
```

Il est possible (quoique délicat) de calculer correctement H_n.

Précautions à prendre sur les flottants

Quelques points à garder en tête :

- préférer les entiers aux flottants aussi souvent que possible ;
- garder à l'esprit que les résultats peuvent avoir des problèmes de précision importants ;
- parfois, $(a+b)+c \neq a+(b+c)$;
- parfois $(a+b)-a \neq b$;
- sommer des flottants nécessite des précautions !

Test de nullité correct d'un réel

On n'écrira en général pas :

```
if x = 0 then
    ...
```

Mais de préférence :

```
if fabs x < 1e-10 then
    ...
```

où

```
let fabs x =
  if x < 0.0 then -.x else x;;
```

Test d'« égalité » correct de réels

De même, on n'écrira en général pas :

```
if x = y then
    ...
```


Mais de préférence :

```
if fabs (x-y) < max (fabs x) (fabs y) *. 1e-10 then
    ...
```