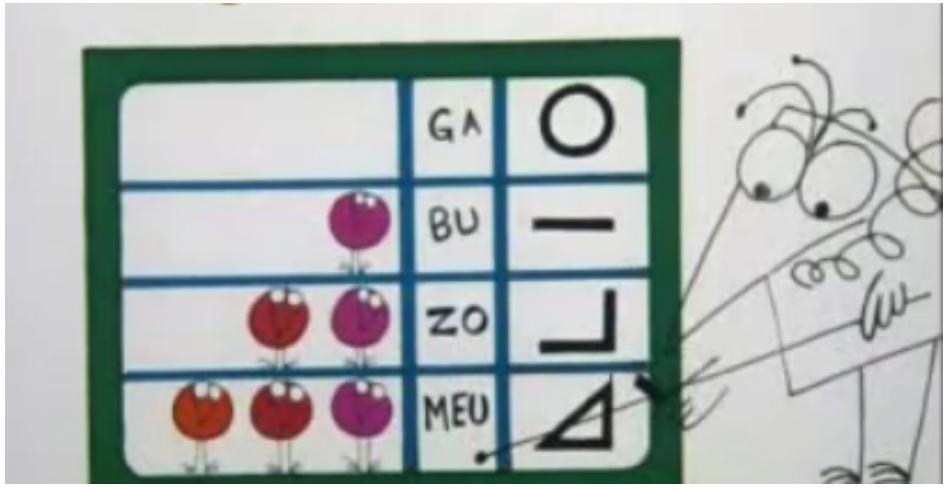
Représentation des entiers

Informatique pour tous



Base de numérotation

Question

Comment représenter un nombre ?

Base de numérotation

1ère méthode, avec des bâtons (système **unaire**) :

|

||

|||

||||

...

Base de numérotation

1ère méthode, avec des bâtons (système **unaire**) :

|

||

|||

||||

...

Prend beaucoup de temps pour écrire un grand nombre : il faut n caractères pour écrire un nombre n.

Base de numérotation

Idée : regrouper par «paquets».

Ainsi, le nombre $1234 = 10^3 + 2 \times 10^2 + 3 \times 10 + 4$ signifie :

1. 1 paquet de 10^3
2. 2 paquets de 10^2
3. 3 paquets de 10
4. 4 paquets de 1

Base de numérotation

Idée : regrouper par «paquets».

Ainsi, le nombre $1234 = 10^3 + 2 \times 10^2 + 3 \times 10 + 4$ signifie :

1. 1 paquet de 10^3
2. 2 paquets de 10^2
3. 3 paquets de 10
4. 4 paquets de 1

L'unique raison d'avoir regroupé par paquets de 10 est que nous avons 10 doigts sur nos mains.

Base de numérotation

Théorème (admis)

Soit $b \in \mathbb{N}^*$. Tout entier $n \in \mathbb{N}^*$ peut s'écrire de façon unique sous la forme :

$$n = n_{p-1} \times b^{p-1} + ... + n_1 \times b + n_0$$

où $0 \leq n_i < b$, $\forall i$.

Base de numérotation

Théorème (admis)

Soit $b \in \mathbb{N}^*$. Tout entier $n \in \mathbb{N}^*$ peut s'écrire de façon unique sous la forme :

$$n = n_{p-1} \times b^{p-1} + ... + n_1 \times b + n_0$$

où $0 \leq n_i < b$, $\forall i$.

Définition

La suite $n_{p-1}, ..., n_1, n_0$ est l'écriture de n en base b, et on écrit :

$$\boldsymbol{n = < n_{p-1} ... n_1 n_0 >_b}$$

Conversion **en** base 10

Il est facile de passer d'une base quelconque à la base décimale :

$$< 121 >_3 = 1 \times 3^2 + 2 \times 3 + 1 = 16 \ (= < 16 >_{10})$$

Conversion **en** base 10

Il est facile de passer d'une base quelconque à la base décimale :

$$< 121 >_3 = 1 \times 3^2 + 2 \times 3 + 1 = 16 \ (= < 16 >_{10})$$

$$< 1011 >_2 =$$

Conversion **en** base 10

Il est facile de passer d'une base quelconque à la base décimale :

$$< 121 >_3 = 1 \times 3^2 + 2 \times 3 + 1 = 16 \ (= < 16 >_{10})$$

$$< 1011 >_2 = 11$$

Exemples

$$< 1\underbrace{00...00}_{\ell} >_2 =$$

$$< \underbrace{11...11}_{\ell} >_2 =$$

Exemples

$$< 1\underbrace{00...00}_{\ell} >_2 = 2^{\ell}$$

$$< \underbrace{11...11}_{\ell} >_2 =$$

Exemples

$$< 1\underbrace{00...00}_{\ell} >_2 = 2^{\ell}$$

$$< \underbrace{11...11}_{\ell} >_2 = \ < 1\underbrace{00...00}_{\ell} >_2 -1 = 2^{\ell} - 1$$

Bases fréquentes

1. Si la base n'est pas spécifiée, il s'agit de la base 10 (**décimale**).
2. Les ordinateurs utilisent la base 2 (**binaire**) : 1 si il y a passage de courant, 0 sinon.
3. On rencontre aussi la base 16 (**hexadécimale**) en informatique. Comme il n'y a que 10 chiffres, on est obligé d'utiliser des lettres :

$$A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$$

Retour en primaire

Remarque : on peut additionner et multiplier deux nombres en base b comme vous avez l'habitude.

Conversion **en** base 10

Question

Écrire une fonction `to_base10` ayant comme arguments une base `b` et une liste `L`, et renvoyant le nombre $< \texttt{L[len(L) - 1]...L[1]L[0]} >_b$ converti en base 10.

Conversion **en** base 10

Question

Écrire une fonction `to_base10` ayant comme arguments une base `b` et une liste `L`, et renvoyant le nombre $<$ `L[len(L) - 1]...L[1]L[0]` $>_b$ converti en base 10.

```
def to_base10(b, L):
    res = 0
    for i in range(len(L)):
        res += L[i] * (b**i)
    return res
```

Conversion **en** base 10

Question

Écrire une fonction `to_base10` ayant comme arguments une base `b` et une liste `L`, et renvoyant le nombre $<$ `L[len(L) - 1]...L[1]L[0]` $>_b$ converti en base 10.

```
def to_base10(b, L):
    res = 0
    for i in range(len(L)):
        res += L[i] * (b**i)
    return res
```

L'algorithme fait `len(L)` calculs de puissances, qui sont coûteux...

Conversion **en** base 10

On peut se passer du calcul de puissance

Conversion **en** base 10

On peut se passer du calcul de puissance , en la « mémorisant » :

```
def to_base10(b, L):
    res = 0
    puiss = 1
    for i in range(len(L)):
        res += L[i] * puiss
        puiss *= b
    return res
```

Complexité :

Conversion **en** base 10

On peut se passer du calcul de puissance , en la « mémorisant » :

```
def to_base10(b, L):
    res = 0
    puiss = 1
    for i in range(len(L)):
        res += L[i] * puiss
        puiss *= b
    return res
```

Complexité : $O(\texttt{len(L)})$

Conversion **depuis** la base 10

On veut maintenant passer de la base décimale à une autre base.

Pour passer un nombre n de la base 10 à la base 2, il faut trouver les n_i tels que :

$$n = n_{p-1} \times 2^{p-1} + ... + n_1 \times 2 + n_0$$

Conversion **depuis** la base 10

On veut maintenant passer de la base décimale à une autre base.

Pour passer un nombre n de la base 10 à la base 2, il faut trouver les n_i tels que :

$$n = n_{p-1} \times 2^{p-1} + ... + n_1 \times 2 + n_0$$

$$\Leftrightarrow n = \underbrace{2}_{b} \times \underbrace{\left(n_{p-1} \times 2^{p-2} + ... + n_1\right)}_{q} + \underbrace{n_0}_{r}$$

Conversion **depuis** la base 10

On veut maintenant passer de la base décimale à une autre base.

Pour passer un nombre n de la base 10 à la base 2, il faut trouver les n_i tels que :

$$n = n_{p-1} \times 2^{p-1} + ... + n_1 \times 2 + n_0$$

$$\Leftrightarrow n = \underbrace{2}_{b} \times \underbrace{\left(n_{p-1} \times 2^{p-2} + ... + n_1\right)}_{q} + \underbrace{n_0}_{r}$$

Ainsi, n_0 est le reste de la division de n par 2, n_1 est le reste de la division de q par 2, et ainsi de suite...

Exemple

Passons 98 de la base 10 à la base 2 :

Exemple

Passons 98 de la base 10 à la base 2 :

98	|	2							
0	|	49	|	2					
		1	|	24	|	2			
				0	|	12	|	2	
						0	|	6	|	2
								0	|	3	|	2
										1	|	1	|	2
												1	|	0

Exemple

Passons 98 de la base 10 à la base 2 :

98	2	
0	49	2
1	24	2
0	12	2
0	6	2
0	3	2
1	1	2
1	0	

Donc $98 = <1100010>_2$ (on lit les restes « à l'envers »).

Représentation machine

Dans un ordinateur, les entiers positifs sont représentés en base 2.

Un ordinateur a une mémoire limitée donc il ne peut pas stocker des entiers arbitrairement grands.

Par exemple, un processeur 64 bits stocke les entiers en utilisant 64 bits, donc peut stocker n'importe quel entier entre 0 et $2^{64} - 1$.

Conversion **depuis** la base 10

Question

Écrire une fonction `from_base10` prenant une base `b` et un nombre `n` et renvoyant l'écriture de `n` en base `b`.

Conversion **depuis** la base 10

Question

Écrire une fonction `from_base10` prenant une base `b` et un nombre `n` et renvoyant l'écriture de `n` en base `b`.

```
def from_base10(b, n):
    L = []
    while n != 0:
        L.append(n % b)
        n = n // b
    return L
```

Conversion **depuis** la base 10

```
In [19]: from_base10(2, 98)
Out[19]: [0, 1, 0, 0, 0, 1, 1]
```

⚠ la liste doit être lue de droite à gauche :

$$98 = < 1100010 >_2$$

Nombre de chiffres

Question

Quel est le nombre p de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Nombre de chiffres

Question

Quel est le nombre p de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Le plus petit nombre à p chiffres en base b est :

Nombre de chiffres

Question

Quel est le nombre p de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Le plus petit nombre à p chiffres en base b est : b^{p-1}.
Donc :

$$b^{p-1} <= n < b^p$$

Nombre de chiffres

Question

Quel est le nombre p de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Le plus petit nombre à p chiffres en base b est : b^{p-1}.
Donc :

$$b^{p-1} <= n < b^p$$

Comme $\log_b$ est $\nearrow$:

$$(p-1)\underbrace{\log_b(b)}_{=1} \leq \log_b(n) < p\underbrace{\log_b(b)}_{=1}$$

D'où :

$$p - 1 = \lfloor \log_b(n) \rfloor$$

Nombre de chiffres

Question

Quel est le nombre de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Réponse :

$$\boxed{\lfloor \log_b(n) \rfloor + 1}$$

Nombre de chiffres

Question

Quel est le nombre de chiffres de l'écriture en base $b \neq 1$ d'un entier n ?

Réponse :

$$\boxed{\lfloor \log_b(n) \rfloor + 1}$$

Comme $\log_b(n) = \frac{\ln(n)}{\ln(b)}$, $\lfloor \log_b(n) \rfloor + 1 = O(\ln(n))$.

Conversion **depuis** la base 10

```python
def from_base10(b, n):
    L = []
    while n != 0:
        L.append(n % b)
        n = n // b
    return L
```

Question

Quelle est la complexité de `from_base10` ?

Conversion **depuis** la base 10

```python
def from_base10(b, n):
    L = []
    while n != 0:
        L.append(n % b)
        n = n // b
    return L
```

Question

Quelle est la complexité de `from_base10` ?

Le `while` s'exécute autant de fois que le nombre de chiffres de n en base b, c'est à dire $\lfloor \log_b(n) \rfloor + 1$ fois.
Chaque passage dans le while effectue un nombre constant d'opération, donc la complexité de `from_base10` est $O(\log(n))$.

Nombre de chiffres

Dans un ordinateur, on ne dispose que d'une mémoire finie. Les entiers positifs sont stockés en base 2, avec un nombre maximum de chiffres (souvent 32 ou 64).

Question

Quels sont les entiers que l'on peut représenter avec p bits ?

Nombre de chiffres

Dans un ordinateur, on ne dispose que d'une mémoire finie. Les entiers positifs sont stockés en base 2, avec un nombre maximum de chiffres (souvent 32 ou 64).

Question

Quels sont les entiers que l'on peut représenter avec p bits ?

Les entiers de $< \underbrace{0...0}_{p} >_2 = 0$ à $< \underbrace{1...1}_{p} >_2 = 2^p - 1$.

Soit un total de 2^p entiers différents (on a deux choix pour chaque bit donc il y a bien 2^p possibilités).

Dépassement

Si les entiers sont stockés sur p bits, les éventuels chiffres qui « dépasseraient » sont supprimés.

Dépassement

Si les entiers sont stockés sur p bits, les éventuels chiffres qui « dépasseraient » sont supprimés.

Par exemple, si les entiers sont stockés sur 4 bits alors :

	1	1	1	1
+	0	0	0	1
$= \not{1}$	0	0	0	0

Dépassement

Si les entiers sont stockés sur p bits, les éventuels chiffres qui « dépasseraient » sont supprimés.

Par exemple, si les entiers sont stockés sur 4 bits alors :

$$\begin{array}{cc} & \boxed{1}\boxed{1}\boxed{1}\boxed{1} \\ + & \boxed{0}\boxed{0}\boxed{0}\boxed{1} \\ = \not{1} & \boxed{0}\boxed{0}\boxed{0}\boxed{0} \end{array}$$

Plus généralement, si les entiers sont stockés sur p bits alors en ajoutant 1 au plus grand entier représentable ($< \underbrace{1...1}_{p} >_2 = 2^p - 1$) on obtient 0.

Dépassement

Le nombre de bits utilisés pour stocker un entier dépend :

1. du langage de programmation
2. du processeur (32 bits, 64 bits...)
3. ...

En Python il n'y a pas de problème de dépassement possible : le nombre de bits utilisés augmente automatiquement quand un entier devient trop grand.

Nombres négatifs

Pour l'instant, nous n'avons écrit que des nombres positifs.

Comment coder des nombres négatifs ?

Nombres négatifs : 1ère solution

Si on écrit nos nombres sur p bits, une première solution est de réserver le premier bit pour le signe :

- Un 0 signifie un nombre positif.
- Un 1 signifie un nombre négatif.

Le reste des bits est utilisé pour écrire le nombre en valeur absolue en base 2.

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

0	1	1	1	1	1	1	1

représente :

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

0	1	1	1	1	1	1	1

représente :

$< 1111111 >_2 =$

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

0	1	1	1	1	1	1	1

représente :

$< 1111111 >_2 = < 10000000 >_2 \ -1 = 2^7 - 1.$

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

0	1	1	1	1	1	1	1

représente :

$< 1111111 >_2 = < 10000000 >_2 \ -1 = 2^7 - 1.$

1	1	1	1	1	1	1	1

représente :

Nombres négatifs : 1ère solution

Exemples sur 8 bits :

-3 est représenté par :

1	0	0	0	0	0	1	1

3 est représenté par :

0	0	0	0	0	0	1	1

0	1	1	1	1	1	1	1

représente :
$< 1111111 >_2 = < 10000000 >_2 \ -1 = 2^7 - 1.$

1	1	1	1	1	1	1	1

représente : $-2^7 + 1$.

Avec 8 bits, on peut donc représenter tous les nombres de $-2^7 + 1$ à $2^7 - 1$.

Nombres négatifs : Complément à 2

La solution précédente a un gros désavantage : l'addition et la multiplication ne marchent plus.

	1	0	0	0	0	0	1	1
+	0	0	0	0	0	0	1	1
=								

Nombres négatifs : Complément à 2

La solution précédente a un gros désavantage : l'addition et la multiplication ne marchent plus.

	1	0	0	0	0	0	1	1
+	0	0	0	0	0	0	1	1
=	1	0	0	0	0	1	1	0

Nombres négatifs : Complément à 2

La solution précédente a un gros désavantage : l'addition et la multiplication ne marchent plus.

	1	0	0	0	0	0	1	1	$\longrightarrow$ -3
+	0	0	0	0	0	0	1	1	$\longrightarrow$ 3
=	1	0	0	0	0	1	1	0	$\longrightarrow$ -6

Nombres négatifs : Complément à 2

Le **codage par complément à 2** (sur p bits) représente tous les entiers $n \in \{-2^{p-1}, ..., 2^{p-1} - 1\}$:

1. Si $n \geq 0$, on représente n par son écriture en base 2.
2. Si $n < 0$, on représente n par l'écriture en base 2 de $n + 2^p$ $(= 2^p - |n|)$.

Nombres négatifs : Complément à 2

Le **codage par complément à 2** (sur p bits) représente tous les entiers $n \in \{-2^{p-1}, ..., 2^{p-1} - 1\}$:

1. Si $n \geq 0$, on représente n par son écriture en base 2.
2. Si $n < 0$, on représente n par l'écriture en base 2 de $n + 2^p$ $(= 2^p - |n|)$.

6 est représenté sur 8 bits par :

Nombres négatifs : Complément à 2

Le **codage par complément à 2** (sur p bits) représente tous les entiers $n \in \{-2^{p-1}, ..., 2^{p-1} - 1\}$:

1. Si $n \geq 0$, on représente n par son écriture en base 2.
2. Si $n < 0$, on représente n par l'écriture en base 2 de $n + 2^p$ $(= 2^p - |n|)$.

6 est représenté sur 8 bits par :

0	0	0	0	0	1	1	0

-6 est représenté sur 8 bits par l'écriture en base 2 de $-6 + 2^8 = 250$ $= <? >_2$

Nombres négatifs : Complément à 2

Soit $n \geq 0$ un entier écrit en base 2 sur p bits et $\tilde{n}$ son complémentaire (où les bits 0 et 1 sont inversés).

Alors : $n + \tilde{n} =$

Nombres négatifs : Complément à 2

Soit $n \geq 0$ un entier écrit en base 2 sur p bits et $\tilde{n}$ son complémentaire (où les bits 0 et 1 sont inversés).

Alors : $n + \tilde{n} = <\underbrace{1...1}_{p}>_2$

Nombres négatifs : Complément à 2

Soit $n \geq 0$ un entier écrit en base 2 sur p bits et $\tilde{n}$ son complémentaire (où les bits 0 et 1 sont inversés).

Alors : $n + \tilde{n} = <\underbrace{1...1}_{p}>_2 = 2^p - 1$

Donc la représentation par complément à 2 de $-n$ est $2^p - n = \tilde{n} + 1$.

Nombres négatifs : Complément à 2

Si $n < 0$, le codage par complément à 2 sur p bits de n peut donc s'obtenir de la façon suivante :

1. Écrire $|n|$ en base 2.
2. Inverser les 0 et les 1.
3. Ajouter 1.

Nombres négatifs : Complément à 2

Si $n < 0$, le codage par complément à 2 sur p bits de n peut donc s'obtenir de la façon suivante :

1. Écrire $|n|$ en base 2.
2. Inverser les 0 et les 1.
3. Ajouter 1.

Exemple avec $n = -6$, $p = 8$:

1. $-n$ est codé par

Nombres négatifs : Complément à 2

Si $n < 0$, le codage par complément à 2 sur p bits de n peut donc s'obtenir de la façon suivante :

1. Écrire $|n|$ en base 2.
2. Inverser les 0 et les 1.
3. Ajouter 1.

Exemple avec $n = -6$, $p = 8$:

1. $-n$ est codé par

0	0	0	0	0	1	1	0

Nombres négatifs : Complément à 2

Si $n < 0$, le codage par complément à 2 sur p bits de n peut donc s'obtenir de la façon suivante :

1. Écrire $|n|$ en base 2.
2. Inverser les 0 et les 1.
3. Ajouter 1.

Exemple avec $n = -6$, $p = 8$:

1. $-n$ est codé par

0	0	0	0	0	1	1	0

2. On inverse :

1	1	1	1	1	0	0	1

Nombres négatifs : Complément à 2

Si $n < 0$, le codage par complément à 2 sur p bits de n peut donc s'obtenir de la façon suivante :

1. Écrire $|n|$ en base 2.
2. Inverser les 0 et les 1.
3. Ajouter 1.

Exemple avec $n = -6$, $p = 8$:

1. $-n$ est codé par

0	0	0	0	0	1	1	0

2. On inverse :

1	1	1	1	1	0	0	1

3. On ajoute 1 :

1	1	1	1	1	0	1	0

Nombres négatifs : Complément à 2

Avec le codage par complément à 2, on peut additionner et multiplier des nombres, en «oubliant» les dépassements :

	0	0	0	0	0	1	1	0
+	1	1	1	1	1	0	1	0
=								

Nombres négatifs : Complément à 2

Avec le codage par complément à 2, on peut additionner et multiplier des nombres, en «oubliant» les dépassements :

		0	0	0	0	0	1	1	0	$\longrightarrow$	6
+		1	1	1	1	1	0	1	0	$\longrightarrow$	-6
=	$\not{1}$	0	0	0	0	0	0	0	0	$\longrightarrow$	0

Nombres négatifs : Complément à 2

Question

Écrire une fonction `complement2(n, p)` renvoyant le codage en complément à 2 sur p bits de n.

Question 3 :

Parmi les affirmations suivantes lesquelles sont vraies :

A) Un nombre entier naturel qui est représenté en binaire par une suite de 0 et de 1 et qui se termine par 1 est pair.

B) Le nombre décimal 0,1 possède une représentation binaire finie.

C) Sur un octet de mémoire le plus grand entier naturel représentable par une suite de 0 ou de 1 est 255.

D) 110 en binaire représente 10 en base dix.