

Comment sont stockées vos données dans un ordinateur

Yann Strozecki
yann.strozecki@uvsq.fr

Université de Versailles St-Quentin-en-Yvelines

Année universitaire 2022-2023

Programme

Sur 12 cours et 12 tds.

Chargé de TD : Loric Duhazé, Coline Gianfrotta, Laurie Guenin et Yann Strozecki.

Notation prévue : 3 QCMs, 1 contrôle et un projet.

- ▶ Représenter les nombres, le texte et les images
- ▶ Compresser des données, les crypter et corriger les erreurs
- ▶ Python : utiliser Tkinter, Pillow et Numpy
- ▶ Calcul des propositions et calcul des prédicats (Christina Boura)

Plan

Représentations numériques

La mémoire de l'ordinateur

Dans un ordinateur, la mémoire est séparée du calcul (processeur).
Qu'est-ce qui sert de mémoire à un ordinateur ?

La mémoire de l'ordinateur

Dans un ordinateur, la mémoire est séparée du calcul (processeur).
Qu'est-ce qui sert de mémoire à un ordinateur ?

Qu'y a-t-il dans la mémoire d'un ordinateur ?

La mémoire de l'ordinateur

Dans un ordinateur, la mémoire est séparée du calcul (processeur).
Qu'est-ce qui sert de mémoire à un ordinateur ?

Qu'y a-t-il dans la mémoire d'un ordinateur ?

0	1	1	0	1	0	0	0
0	1	1	0	0	0	1	1
1	0	1	1	0	1	1	0
1	1	1	1	1	1	0	1

La mémoire de l'ordinateur

Dans un ordinateur, la mémoire est séparée du calcul (processeur).
Qu'est-ce qui sert de mémoire à un ordinateur ?

Qu'y a-t-il dans la mémoire d'un ordinateur ?

0	1	1	0	1	0	0	0
0	1	1	0	0	0	1	1
1	0	1	1	0	1	1	0
1	1	1	1	1	1	0	1

On va expliquer comment sont représentées toutes les informations sur votre ordinateur comme des suites de 0 et de 1.

Où l'on apprend à compter

Notre système de numération compte 10 symboles (chiffres), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. On fabrique des mots avec ces symboles, ce sont les **nombre**s. C'est le système **décimal**.

Ou l'on apprend à compter

Notre système de numération compte 10 symboles (chiffres), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. On fabrique des mots avec ces symboles, ce sont les **nombre**s. C'est le système **décimal**.

D'autres systèmes existent, les chiffres romains par exemple : *I*, *V*, *X*, *L*, *C*, *D*, *M*. Ainsi *CDLXXIV* signifie 474.

Où l'on apprend à compter

Notre système de numération compte 10 symboles (chiffres), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. On fabrique des mots avec ces symboles, ce sont les **nombre**s. C'est le système **décimal**.

D'autres systèmes existent, les chiffres romains par exemple : *I, V, X, L, C, D, M*. Ainsi *CDLXXIV* signifie 474.

L'homme des cavernes sait compter : pourtant les chiffres et même l'écriture ne sont pas inventés.

Où l'on apprend à compter

Notre système de numération compte 10 symboles (chiffres), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. On fabrique des mots avec ces symboles, ce sont les **nombre**s. C'est le système **décimal**.





D'autres systèmes existent, les chiffres romains par exemple : *I, V, X, L, C, D, M*. Ainsi *CDLXXIV* signifie 474.

L'homme des cavernes sait compter : pourtant les chiffres et même l'écriture ne sont pas inventés.

On utilise des *cailloux*, on trace des batons, on compte sur ses doigts. C'est un système **unaire**. Problème, c'est **encombrant**!!!

Des nombres plus grand plus petit

Une solution égyptienne :

1	
10	
100	
1000	

Des nombres plus grand plus petit

Une solution égyptienne :

1	
10	∩
100	⤿
1000	⊕

Système **additif** : on ajoute les valeurs des symboles pour obtenir le nombre représenté. La position des chiffres n'est pas importante. Le système est plus efficace que le unaire.

Des nombres plus grand plus petit

Une solution égyptienne :

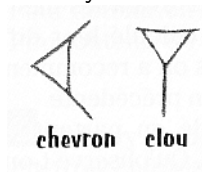
1	
10	∩
100	∩
1000	∩ ↓

Système **additif** : on ajoute les valeurs des symboles pour obtenir le nombre représenté. La position des chiffres n'est pas importante. Le système est plus efficace que le unaire.

Combien de symboles pour représenter 474 ?

Encore plus grand et encore plus petit

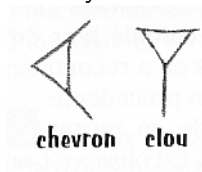
Les babyloniens avait un vrai système à base 60 avec seulement deux symboles :



Le clou pour 1 et le chevron pour 10. Leur position est **importante** ainsi que les espaces de séparation : c'est un système **positionnel**.

Encore plus grand et encore plus petit

Les babyloniens avait un vrai système à base 60 avec seulement deux symboles :

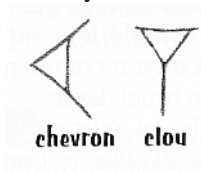


Le clou pour 1 et le chevron pour 10. Leur position est **importante** ainsi que les espaces de séparation : c'est un système **positionnel**.

Représenter 474 dans ce système.

Encore plus grand et encore plus petit

Les babyloniens avait un vrai système à base 60 avec seulement deux symboles :



Le clou pour 1 et le chevron pour 10. Leur position est **importante** ainsi que les espaces de séparation : c'est un système **positionnel**.

Représenter 474 dans ce système.

On utilise encore la base 60 pour les degrés et les heures, pratique car 60 est très divisible.

C'est la base

Un nombre en *base* b , s'écrit

$$a_k a_{k-1} \dots a_1 a_0$$

où $a_i \in [|0, b - 1|]$, i.e. les a_i sont des chiffres entre 0 et $b - 1$.

Pour préciser qu'on est en base b , on note le nombre

$$(a_k a_{k-1} \dots a_1 a_0)_b.$$

Par exemple $(2304)_5$ et $(329)_{10}$ sont deux manières de noter le même nombre.

Valeur d'un nombre

Si un nombre s'écrit $(a_k a_{k-1} \dots a_1 a_0)_b$, alors sa valeur est

$$a_k * b^k + a_{k-1} * b^{k-1} \dots + a_1 * b^1 + a_0 * b^0$$

Sur l'exemple précédant : $(2304)_5 = 2 * 5^3 + 3 * 5^2 + 4 * 5^0 = 329$.

Calculer la valeur de $(1212)_3$.

...avec des 0 et des 1

Le plus simple : le **système binaire** avec deux symboles, 0 et 1.
Dans un ordinateur courant/absence de courant (processeur),
condensateur chargé ou non (RAM) ou spin positif/spin négatif
(disque dur) ...

Et dans un **cerveau** ?

...avec des 0 et des 1

Le plus simple : le **système binaire** avec deux symboles, 0 et 1.
Dans un ordinateur courant/absence de courant (processeur),
condensateur chargé ou non (RAM) ou spin positif/spin négatif
(disque dur) ...

Et dans un **cerveau** ?

Apprenons à compter (retour en maternelle) : 0, 1, 10, 11, 100,
101 ...

...avec des 0 et des 1

Le plus simple : le **système binaire** avec deux symboles, 0 et 1.
Dans un ordinateur courant/absence de courant (processeur),
condensateur chargé ou non (RAM) ou spin positif/spin négatif
(disque dur) ...

Et dans un **cerveau** ?

Apprenons à compter (retour en maternelle) : 0, 1, 10, 11, 100,
101 ...

Donner les puissances de 2 jusqu'à 2^{10} .

...avec des 0 et des 1

Le plus simple : le **système binaire** avec deux symboles, 0 et 1.
Dans un ordinateur courant/absence de courant (processeur),
condensateur chargé ou non (RAM) ou spin positif/spin négatif
(disque dur) ...

Et dans un **cerveau** ?

Apprenons à compter (retour en maternelle) : 0, 1, 10, 11, 100,
101 ...

Donner les puissances de 2 jusqu'à 2^{10} .

Calculer combien vaut le nombre $(1001101)_2$.

...avec des 0 et des 1

Le plus simple : le **système binaire** avec deux symboles, 0 et 1.
Dans un ordinateur courant/absence de courant (processeur),
condensateur chargé ou non (RAM) ou spin positif/spin négatif
(disque dur) ...

Et dans un **cerveau** ?

Apprenons à compter (retour en maternelle) : 0, 1, 10, 11, 100,
101 ...

Donner les puissances de 2 jusqu'à 2^{10} .

Calculer combien vaut le nombre $(1001101)_2$.

Comment fait-on pour transformer un nombre en base 10 en un
nombre en base 2 ?

Changement de base

Observation :

- ▶ La division entière d'un nombre par 2 donne un nombre qui a un symbole de moins en base 2.
- ▶ Ce symbole est donné par le reste par la division entière.

Changement de base

Observation :

- ▶ La division entière d'un nombre par 2 donne un nombre qui a un symbole de moins en base 2.
- ▶ Ce symbole est donné par le reste par la division entière.

Justification : $(a_k a_{k-1} \dots a_1 a_0)_2$ vaut $a_k * 2^k + \dots + a_1 * 2 + a_0$.

$$\text{Or } a_k * 2^k + \dots + a_1 * 2 + a_0 = 2 \underbrace{(a_k * 2^{k-1} + \dots + a_1)}_{\text{quotient}} + \underbrace{a_0}_{\text{reste}}.$$

Changement de base

Observation :

- ▶ La division entière d'un nombre par 2 donne un nombre qui a un symbole de moins en base 2.
- ▶ Ce symbole est donné par le reste par la division entière.

Justification : $(a_k a_{k-1} \dots a_1 a_0)_2$ vaut $a_k * 2^k + \dots + a_1 * 2 + a_0$.

$$\text{Or } a_k * 2^k + \dots + a_1 * 2 + a_0 = 2 \underbrace{(a_k * 2^{k-1} + \dots + a_1)}_{\text{quotient}} + \underbrace{a_0}_{\text{reste}}.$$

Algorithme : Tant que le nombre est différent de 0 le diviser par 2 (division entière).

La suite des restes écrits de droite à gauche donne l'écriture en base 2.

Changement de base

Observation :

- ▶ La division entière d'un nombre par 2 donne un nombre qui a un symbole de moins en base 2.
- ▶ Ce symbole est donné par le reste par la division entière.

Justification : $(a_k a_{k-1} \dots a_1 a_0)_2$ vaut $a_k * 2^k + \dots + a_1 * 2 + a_0$.

$$\text{Or } a_k * 2^k + \dots + a_1 * 2 + a_0 = 2 \underbrace{(a_k * 2^{k-1} + \dots + a_1)}_{\text{quotient}} + \underbrace{a_0}_{\text{reste}}.$$

Algorithme : Tant que le nombre est différent de 0 le diviser par 2 (division entière).

La suite des restes écrits de droite à gauche donne l'écriture en base 2.

Calculer la représentation en base 2 de 79.

Les unités en informatique

Un symbole binaire stocké en mémoire est appelé **un bit**.

Les bits sont regroupés en :

- ▶ quatre bits qui forment un symbole hexadécimal, utile pour l'interprétation humaine
- ▶ huit bits qui forment un **octet** ou byte en anglais (plus petite unité de mémoire manipulable)
- ▶ un **mot machine** de largeur 32-bits ou 64-bits typiquement (unité manipulé naturellement par le processeur)

Représentation informatique

Il existe deux autres bases classiques en informatique :

- ▶ l'**octal**, c'est à dire la base 8 (plus très utilisée),
- ▶ l'**hexadécimal** c'est à dire la base 16 qui utilise les symboles de 0 à 9 puis $A(10)$, $B(11)$, $C(12)$, $D(13)$, $E(14)$, $F(15)$.

Représentation informatique

Il existe deux autres bases classiques en informatique :

- ▶ l'**octal**, c'est à dire la base 8 (plus très utilisée),
- ▶ l'**hexadécimal** c'est à dire la base 16 qui utilise les symboles de 0 à 9 puis $A(10)$, $B(11)$, $C(12)$, $D(13)$, $E(14)$, $F(15)$.

Un chiffre en octal peut se voir comme un nombre de trois chiffres en binaire.

Par exemple $(5)_8 = (101)_2$.

De même $(B)_{16} = (1011)_2$.

Représentation informatique

Il existe deux autres bases classiques en informatique :

- ▶ l'**octal**, c'est à dire la base 8 (plus très utilisée),
- ▶ l'**hexadécimal** c'est à dire la base 16 qui utilise les symboles de 0 à 9 puis $A(10), B(11), C(12), D(13), E(14), F(15)$.

Un chiffre en octal peut se voir comme un nombre de trois chiffres en binaire.

Par exemple $(5)_8 = (101)_2$.

De même $(B)_{16} = (1011)_2$.

On a un entier sur 16 bits en mémoire : $\underbrace{1001}_9 \underbrace{1101}_D \underbrace{0100}_4 \underbrace{0101}_5$.

Attention, l'ordre de lecture a une importance (endiannes) !

Représenter des entiers positifs

Un entier sur votre ordinateur peut être codé sur 32 bits : il a une valeur entre 0 et 4294967295.

En C, cela correspond au type **unsigned int**.

Représenter des entiers positifs

Un entier sur votre ordinateur peut être codé sur 32 bits : il a une valeur entre 0 et 4294967295.

En C, cela correspond au type **unsigned int**.

Que se passe-t-il si vous faites $4294967295 + 1$ en C ?

Représenter des entiers positifs

Un entier sur votre ordinateur peut être codé sur 32 bits : il a une valeur entre 0 et 4294967295.

En C, cela correspond au type **unsigned int**.

Que se passe-t-il si vous faites $4294967295 + 1$ en C ?

0

On utilise une **représentation modulaire**.

Représenter des entiers positifs

Un entier sur votre ordinateur peut être codé sur 32 bits : il a une valeur entre 0 et 4294967295.

En C, cela correspond au type **unsigned int**.

Que se passe-t-il si vous faites $4294967295 + 1$ en C ?

0

On utilise une [représentation modulaire](#).

Les entiers sur 32 bits ne sont pas forcément suffisant, que faire ?

Représenter des entiers positifs

Un entier sur votre ordinateur peut être codé sur 32 bits : il a une valeur entre 0 et 4294967295.

En C, cela correspond au type **unsigned int**.

Que se passe-t-il si vous faites $4294967295 + 1$ en C ?

0

On utilise une [représentation modulaire](#).

Les entiers sur 32 bits ne sont pas forcément suffisant, que faire ?

Les architectures modernes supportent des entiers sur 64 bits qui vont de 0 à 18446744073709551615 (et même des entiers sur 128 bits, 256 bits et 512 bits).

Cela correspond aux types **unsigned long int** et **unsigned long long int** en C.

Taille arbitraire

En Python, les entiers ont une taille arbitraire : ils peuvent être aussi grand qu'on veut. Dans d'autres langages, on a aussi parfois besoin de manipuler des nombres plus grand que 2^{64} .

Taille arbitraire

En Python, les entiers ont une taille arbitraire : ils peuvent être aussi grand qu'on veut. Dans d'autres langages, on a aussi parfois besoin de manipuler des nombres plus grand que 2^{64} .

On peut représenter un nombre plus petit que 2^{128} comme $n = n_1 + 2^{64}n_2$ avec n_1 et n_2 des entiers sur 64 bits. Autrement dit l'entier n s'écrit n_2n_1 et peut se représenter par les deux entiers n_1 et n_2 .

Taille arbitraire

En Python, les entiers ont une taille arbitraire : ils peuvent être aussi grand qu'on veut. Dans d'autres langages, on a aussi parfois besoin de manipuler des nombres plus grand que 2^{64} .

On peut représenter un nombre plus petit que 2^{128} comme $n = n_1 + 2^{64}n_2$ avec n_1 et n_2 des entiers sur 64 bits. Autrement dit l'entier n s'écrit n_2n_1 et peut se représenter par les deux entiers n_1 et n_2 .

En Python, un entier est représenté en interne par une liste d'entiers sur 64 bits.

Limites

Défauts dus à la représentation des entiers de Python.

- ▶ Les entiers nécessitent quatre fois plus de place pour être stockés.
- ▶ Il faut implémenter les opérations de base (plus lent que les opérations processeurs).

Étant donné $s = (s_1, s_2)$, $m = (m_1, m_2)$ et $n = (n_1, n_2)$, comment faire l'opération $s = n + m$?

Limites

Défauts dus à la représentation des entiers de Python.

- ▶ Les entiers nécessitent quatre fois plus de place pour être stockés.
- ▶ Il faut implémenter les opérations de base (plus lent que les opérations processeurs).

Étant donné $s = (s_1, s_2)$, $m = (m_1, m_2)$ et $n = (n_1, n_2)$, comment faire l'opération $s = n + m$?

```
s1 = n1 + m1
```

```
s2 = n2 + m2
```

```
Si s1 >= 2{64}
```

```
Alors
```

```
    s1 = s1 - 2{64} #automatique si s1 sur 64 bits
```

```
    s2 = s2 + 1
```

```
Si s2 >= 2{64}
```

```
Alors Integer Overflow
```

Représentation négative

Comment représenter un nombre négatif :

- ▶ **Bit de signe** : le nombre -19 s'écrit sur 8 bits 10010011
alors que 19 s'écrit 00010011.

Représentation négative

Comment représenter un nombre négatif :

- ▶ **Bit de signe** : le nombre -19 s'écrit sur 8 bits 10010011 alors que 19 s'écrit 00010011.

Problèmes :

- ▶ 00000000 et 10000000 représentent 0.
- ▶ L'algorithme d'addition dépend du signe.

Représentation négative

Comment représenter un nombre négatif :

- ▶ **Bit de signe** : le nombre -19 s'écrit sur 8 bits 10010011 alors que 19 s'écrit 00010011.

Problèmes :

- ▶ 00000000 et 10000000 représentent 0.
- ▶ L'algorithme d'addition dépend du signe.
- ▶ **Complément à deux** : le nombre négatif a sur b bits s'écrit $2^b - |a|$. Le nombre positif a s'écrit a .

Représentation négative

Comment représenter un nombre négatif :

- ▶ **Bit de signe** : le nombre -19 s'écrit sur 8 bits 10010011 alors que 19 s'écrit 00010011.

Problèmes :

- ▶ 00000000 et 10000000 représentent 0.
- ▶ L'algorithme d'addition dépend du signe.
- ▶ **Complément à deux** : le nombre négatif a sur b bits s'écrit $2^b - |a|$. Le nombre positif a s'écrit a .
 - ▶ Le nombre 19 s'écrit en binaire 00010011 sur 8 bits
 - ▶ Le nombre -19 s'écrit sur 8 bits comme $2^8 - 19 = 237$ i.e. 11101101

La deuxième solution est utilisée dans les processeurs modernes car elle est pratique pour les opérations arithmétiques.

Une méthode pour calculer le complément à deux

Pour trouver le codage de $-a$ avec a positif :

- ▶ On considère la représentation binaire de a , par exemple $12 = (00001100)_2$
- ▶ On inverse les bits de a , $(11110011)_2$
- ▶ On ajoute 1, $(11110100)_2$ ce qui vaut $2^8 - 12 = 244$

Une méthode pour calculer le complément à deux

Pour trouver le codage de $-a$ avec a positif :

- ▶ On considère la représentation binaire de a , par exemple $12 = (00001100)_2$
- ▶ On inverse les bits de a , $(11110011)_2$
- ▶ On ajoute 1, $(11110100)_2$ ce qui vaut $2^8 - 12 = 244$

Ajouter 1 correspond à trouver le 0 le plus à droite et inverser tous les bits à partir de ce 0.

Représentation flottante

On veut représenter des nombres à virgule comme 2,125. Il est donné par 3 champs (ici **float** sur 32 bits) :

- ▶ Un bit de signe b .
- ▶ Un exposant e sur 8 bits.
- ▶ Une mantisse m sur 23 bits.

Le nombre codé est $-1^b \times (1, m)_2 \times 2^{e-127}$.

Représentation flottante

On veut représenter des nombres à virgule comme 2,125. Il est donné par 3 champs (ici **float** sur 32 bits) :

- ▶ Un bit de signe b .
- ▶ Un exposant e sur 8 bits.
- ▶ Une mantisse m sur 23 bits.

Le nombre codé est $-1^b \times (1, m)_2 \times 2^{e-127}$.

- ▶ $2,125 = (10,001)_2$
- ▶ $(10,001)_2 = (1,0001)_2 \times 2^1$
- ▶ $b = 0, e = 128 = (10000000)_2, m = 0001$

Difficulté de manipulation des types réels

Si on veut plus de précision en flottant on peut utiliser des **double** sur 64 bits.

Difficulté de manipulation des types réels

Si on veut plus de précision en flottant on peut utiliser des **double** sur 64 bits.

Problème de précision : Si on met $0.3 - 0.2 == 0.1$ dans un programme C ou Python on obtient **false** !

Pourquoi ?

Difficulté de manipulation des types réels

Si on veut plus de précision en flottant on peut utiliser des **double** sur 64 bits.

Problème de précision : Si on met $0.3 - 0.2 == 0.1$ dans un programme C ou Python on obtient **false** !

Pourquoi ? Parce que les représentations machines de 0.3, 0.2 et 0.1 sont des approximations de la valeur exacte.

Difficulté de manipulation des types réels

Si on veut plus de précision en flottant on peut utiliser des **double** sur 64 bits.

Problème de précision : Si on met $0.3 - 0.2 == 0.1$ dans un programme C ou Python on obtient **false** !

Pourquoi ? Parce que les représentations machines de 0.3, 0.2 et 0.1 sont des approximations de la valeur exacte.

Valeurs spéciales :

- ▶ $+\infty$ et $-\infty$ avec mantisse et exposant à 0.
- ▶ Le symbole **NaN** (Not a Number) pour des mantisses non nulles et un exposant 00000000 ou 11111111. Résultat d'une division par zéro ou tout autre opération illégale.

Les tableaux de nombres

Tableau := suite de taille fixée d'éléments du même type. Un tableau est stocké par la suite des représentations de ses éléments. Sa taille peut être stockée avant les valeurs.

On stocke le tableau d'entiers sur 8 bits $[2, 0, 5, 1]$ par

000000100.00000010.00000000.00000101.00000001

Les tableaux de nombres

Tableau := suite de taille fixée d'éléments du même type. Un tableau est stocké par la suite des représentations de ses éléments. Sa taille peut être stockée avant les valeurs.

On stocke le tableau d'entiers sur 8 bits $[2, 0, 5, 1]$ par

000000100.00000010.00000000.00000101.00000001

Un tableau à deux dimensions (matrice) est stocké ligne par ligne avec les informations de dimension au début. On représente

$$\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$$

par

00000010.00000010.00000001.00000010.00000100.00000101

Résumé

En informatique un des problèmes fondamentaux est la représentation des idées et du réel par des 0 et des 1.

C'est ce qu'on appelle l'**encodage** des données. Cet encodage est une convention qui doit être partagée par tous les utilisateurs de la donnée (par exemple little-endian vs big-endian). On peut ensuite passer d'une représentation à une autre facilement.

Les encodages de nombres sont plus ou moins efficaces :

- ▶ pour la taille de la représentation (e.g. unaire vs binaire)
- ▶ pour la facilité à faire des opérations dessus