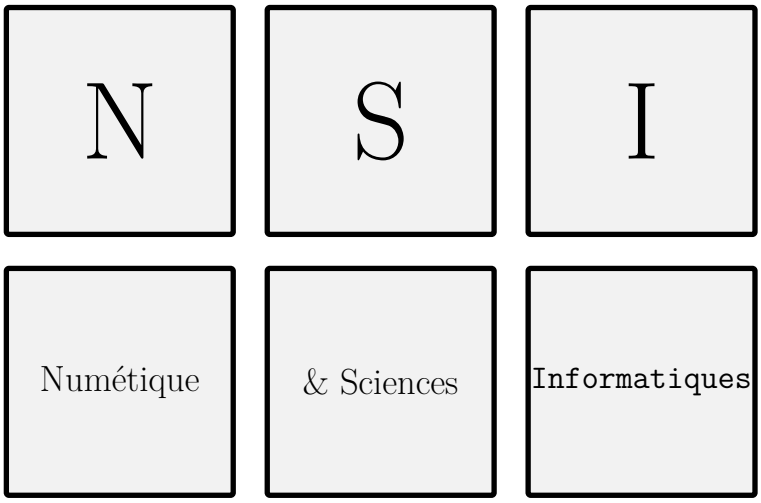
Synthèse de cours 2019-2020 de



Spécilialité de première générale

Avant-propos

Dans le programme officiel :

« Le programme est organisé autour de huit rubriques. Il ne constitue cependant pas un plan de cours. Il appartient aux professeurs de choisir leur progression, sans faire de chaque partie un tout insécable et indépendant des autres. Au contraire, les mêmes notions peuvent être développées et éclairées dans différentes rubriques, en mettant en lumière leurs interactions. »

L'objectif de ce document n'est pas de proposer une progression ni de faire de longues explications de cours, mais au contraire d'apporter une vue synthétique de chaque notion. Ainsi, et mise à part la dimension historique, trop transversale, cette présentation conserve l'organisation en parties du programme officiel. Il a fallu régulièrement faire des choix dans les développements, mais le principal était ici d'apporter le vocabulaire et les mots clés nécessaires à des recherches et des approfondissements personnels.

J'espère que ce document pourra être une ressource utile aux élèves qui y trouveront une sorte de lexique détaillé des connaissances à acquérir en première NSI.

Mickaël BARRAUD
mickael.barraud@ac-nantes.fr
Lycée Jean de Lattre De Tassigny
85000 La Roche-sur-Yon

Table des matières

Types et valeurs de base

Types construits

Traitement de données en tables

Interactions entre l'homme et la machine sur le Web

Architectures matérielles et systèmes d'exploitation

Langages et programmation

Algorithmique

Types et valeurs de base

- Programme officiel

Toute machine informatique manipule une représentation des données dont l'unité minimale est le bit 0/1, ce qui permet d'unifier logique et calcul. Les données de base sont représentées selon un codage dépendant de leur nature : entiers, flottants, caractères et chaînes de caractères. Le codage conditionne la taille des différentes valeurs en mémoire.

Contenus	Capacités attendues	Commentaires
Écriture d'un entier positif dans une base $b \geqslant 2$	Passer de la représentation d'une base dans une autre.	Les bases 2, 10 et 16 sont privilégiées.
Représentation binaire d'un entier relatif	Évaluer le nombre de bits nécessaires à l'écriture en base 2 d'un entier, de la somme ou du produit de deux nombres entiers. Utiliser le complément à 2.	Il s'agit de décrire les tailles courantes des entiers (8, 16, 32 ou 64 bits). Il est possible d'évoquer la représentation des entiers de taille arbitraire de Python.
Représentation approximative des nombres réels : notion de nombre flottant	Calculer sur quelques exemples la représentation de nombres réels : 0,1, 0,25 ou 1/3.	$0{,}2 + 0{,}1$ n'est pas égal à 0,3. Il faut éviter de tester l'égalité de deux flottants. Aucune connaissance précise de la norme IEEE-754 n'est exigible.
Valeurs booléennes : 0, 1. Opérateurs booléens : and, or, not. Expressions booléennes	Dresser la table d'une expression booléenne.	Le ou exclusif (xor) est évoqué. Quelques applications directes comme l'addition binaire sont présentées. L'attention des élèves est attirée sur le caractère séquentiel de certains opérateurs booléens.
Représentation d'un texte en machine. Exemples des encodages ASCII, ISO-8859-1, Unicode	Identifier l'intérêt des différents systèmes d'encodage. Convertir un fichier texte dans différents formats d'encodage.	Aucune connaissance précise des normes d'encodage n'est exigible.

Types et valeurs de base(I)

Représentation des nombres

1 - Codage des entiers naturels

La première règle pour savoir compter est d'être capable de donner l'entier suivant n'importe quel autre. En partant de 0 cela donne :

$b=2$	10	16
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

$b=2$	10	16
1 0000	16	10
1 0001	17	11
1 0010	18	12
1 0011	19	13
1 0100	20	14
1 0101	21	15
1 0110	22	16
1 0111	23	17
1 1000	24	18
1 1001	25	19
1 1010	26	1A
1 1011	27	1B
1 1100	28	1C
1 1101	29	1D
1 1110	30	1E
1 1111	31	1F

$b=2$	10	16
10 0000	32	20
10 0001	33	21
10 0010	34	22
10 0011	35	23
10 0100	36	24
...	...	...
1111 0110	246	F6
1111 0111	247	F7
1111 1000	248	F8
1111 1001	249	F9
1111 1010	250	FA
1111 1011	251	FB
1111 1100	252	FC
1111 1101	253	FD
1111 1110	254	FE
1111 1111	255	FF

« Il n'y a que 10 sortes de personnes : celles qui savent compter en binaire et les autres ! »

a) Lire un entier naturel écrit en base 2 ou 16

En base 10 : les chiffres ont chacun une signification selon leur position.

$$125 = \underset{1}{\text{centaines}}\ \underset{2}{\text{dizaines}}\ \underset{5}{\text{unités}} = \underset{1}{10^2}\ \underset{2}{10^1}\ \underset{5}{10^0} = 1\times10^2+2\times10^1+5\times10^0 = 125$$

En base 2 : $\texttt{11001}_2 = \underset{1}{2^4}\ \underset{1}{2^3}\ \underset{0}{2^2}\ \underset{0}{2^1}\ \underset{1}{2^0} = 1\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0 = 25$

De tête, on calcule d'abord les puissances : $\texttt{11001}_2 = \underset{1}{16}\ \underset{1}{8}\ \underset{0}{4}\ \underset{0}{2}\ \underset{1}{1} = 16+8+1 = 25$

À la machine, on évite les puissances : $\texttt{11001}_2 = (((1\times2+1)\times2+0)\times2+0)\times2+1 = 25$

En base 16 : $\texttt{C9}_{16} = \underset{12}{16^1}\ \underset{9}{16^0} = 12\times16+9 = 81$

Les trois écritures sont utilisables avec Python : `0b11001` vaut 25 et `0xC9` vaut 81.

b) Écrire un entier naturel en base 2 ou 16

De la base 10 vers la base 2

On part de l'astuce de factorisation vue plus haut :

$25 = 12\times2+1$ donc $25 = 12\times2+1 = \texttt{?...?1}_2$

$12 = 6\times2+0$ donc $25 = (6\times2+0)\times2+1 = \texttt{?...?01}_2$

$6 = 3\times2+0$ donc $25 = ((3\times2+0)\times2+0)\times2+1 = \texttt{?...?001}_2$

$3 = 1\times2+1$ donc $25 = (((1\times2+1)\times2+0)\times2+0)\times2+1 = \texttt{?...?1001}_2$

$1 = 0\times2+1$ donc $25 = (((1\times2+1)\times2+0)\times2+0)\times2+1 = \texttt{11001}_2$

On vient de procéder par divisions euclidiennes par 2 successives, jusqu'à trouver un quotient nul. L'écriture binaire est la succession des restes trouvés (à l'envers).

Avec Python : `bin(25)` donne bien `'0b11001'`.

De la base 10 vers la base 16

On peut procéder de même par division euclidienne par 16.

$2019 = 126 \times 16 + 3$ donc $2019 = 126 \times 16 + 3 = \texttt{?...?3}_{16}$

$126 = 7 \times 16 + 14$ donc $2019 = (7 \times 16 + 14) \times 16 + 3 = \texttt{?...?E3}_{16}$

$7 = 0 \times 16 + 7$ donc $2019 = (7 \times 16 + 14) \times 16 + 3 = \texttt{7E3}_{16}$

Avec Python : `hex(2019)` donne bien `'0x7e3'`.

De la base 2 vers la base 16

On fait des paquets de 4 !

$\texttt{1011 0110}_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$\texttt{1011 0110}_2 = (1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \times 2^4 + (0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0)$

$\boxed{\texttt{1011 0110}_2 = \texttt{1011}_2 \times 16 + \texttt{0110}_2}$

$\texttt{1011 0110}_2 = \texttt{B}_{16} \times 16 + \texttt{6}_{16}$

$\texttt{1011 0110}_2 = \texttt{B6}_{16}$

Avec Python : `hex(0b10110110)` donne bien `'0xb6'`.

2 - Codage des entiers relatifs

a) Recours au bit de signe

Pour représenter les entiers relatifs en binaire, la première démarche est de réserver un bit pour coder son signe.

- ⋆ (0)101 1101 est un entier positif (premier bit à gauche égal à 0)
- ⋆ (1)101 1101 est un entier négatif (premier bit à gauche égal à 1)

Le bit de signe étant à gauche il convient de normaliser la taille du mot binaire représentant ces entiers. On utilise généralement 8, 16, 32 ou 64 bits (1, 2, 4 ou 8 octets).

Pour simplifier les explications dans ce cours, on les représentera sur 4 bits seulement, mais on ne perd pas en généralité.

On peut alors se servir du reste des bits pour coder l'entier naturel correspondant.

Si l'entier est positif on retrouve le codage d'un entier naturel :

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111

mais si on utilise le même raisonnement pour coder les nombres négatifs, on se heurte à différents problèmes :

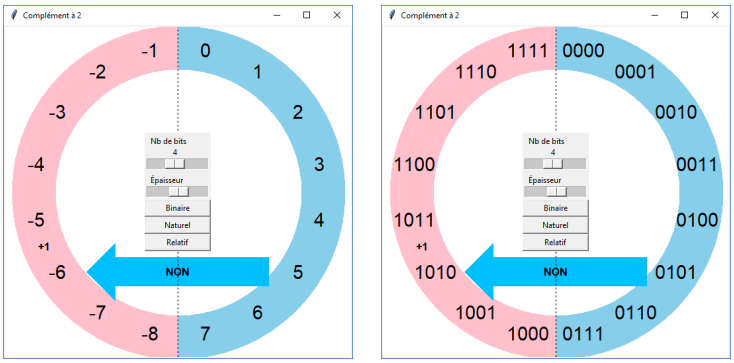
−0 ?	−1 ?	−2 ?	−3 ?	−4 ?	−5 ?	−6 ?	−7 ?
1000	1001	1010	1011	1100	1101	1110	1111

- ⋆ 1000 voudrait dire −0 donc on aurait deux représentations différentes de 0.
- ⋆ Ajouter 1 au nombre binaire ferait diminuer l'entier négatif de 1 !

On a donc opté pour le codage suivant :

−8	−7	−6	−5	−4	−3	−2	−1
1000	1001	1010	1011	1100	1101	1110	1111

On peut en donner une représentation circulaire :

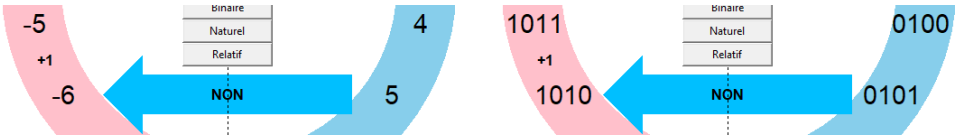


Les nombres binaires et les nombres relatifs sont écrits dans l'ordre en tournant dans le sens des aiguilles d'une montre, tout en conservant la signification du bit de signe.

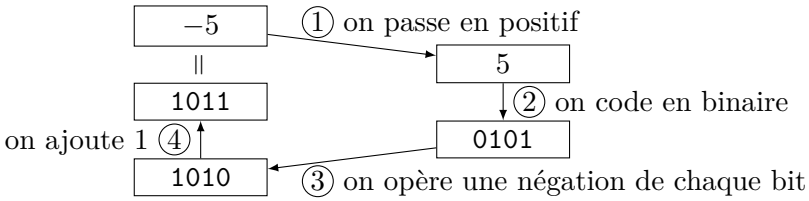
Comme on n'a pas de « -0 » il y a un décalage de 1 dans la symétrie entre entiers positifs et négatifs alors qu'elle est parfaite pour les binaires.

b) Complément à 2

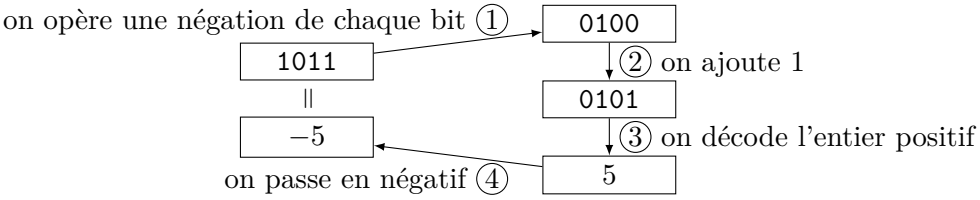
Les résultats précédents permettent de dégager une méthode de codage :



Pour coder un entier négatif (-5) par complément à 2 sur $n(=4)$ bits,



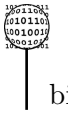
Pour décoder un nombre binaire (`1011`), on peut suivre un cycle similaire :



En fait, pour coder l'entier -5, on cherche le nombre n qui vérifie $n + 5 = 0$

$$\begin{array}{cccccl} 1 & 1 & 1 & 1 & & \leftarrow \text{retenues} \\ & 0 & 1 & 0 & 1 & \leftarrow 5 \\ + & 1 & 0 & 1 & 1 & \leftarrow -5 \\ \hline (1) & 0 & 0 & 0 & 0 & \leftarrow 16 = 2^4 \end{array}$$

D'un point de vue des écritures binaires, cela revient à provoquer une retenue qui dépasse le nombre n de bits de l'écriture pour n'obtenir (lui mis à part) que des bits à 0 : on cherche le nombre n qui vérifie $n + 5 = 2^4$ c'est à dire le complément de 5 à 2^4.



Complément à 2
L'écriture d'un nombre binaire négatif m en complément à 2 sur n bits est l'écriture binaire du nombre positif p tel que $p + (-m) = 2^n$, c'est à dire, le complément à 2^n de $-m$.

c) Nombre de bits nécessaires aux opérations

Un nombre codé sur n bits peut s'écrire de 2^n façons. n bits permettent donc de coder tous les **entiers naturels** de $\texttt{00...0}_2 = 0$ à $\texttt{11...1}_2 = 2^n - 1$.

codage en	plus grand entier	exemple
8 bits = 1 octet	255	primaire RVB
16 bits = 2 octets	65 535	
24 bits = 3 octets	16 777 215	couleur RVB
32 bits = 4 octets	4 294 967 295	IPv4
64 bits = 8 octets	18 446 744 073 709 551 615	SE actuels

Comme $1_2 = 2^0$, $10_2 = 2^1$, $100_2 = 2^2$, $1000_2 = 2^3$, $10000_2 = 2^4$, ...

Un nombre n a au plus k bits significatifs si et seulement si $n < 2^k$.

(**Pour aller plus loin :** Il existe une fonction mathématique définissant le réel y tel que $n = 2^y$. Il s'agit de $\log_2$, le logarithme à base 2 : $n = 2^{\log_2(n)}$.)

- ⋆ Additionner deux entiers naturels n et p avec au plus k bits significatifs en nécessite au plus $k + 1$ (en effet : $n < 2^k$ et $p < 2^k$ donnent $n + p < 2^k + 2^k = 2 \times 2^k = 2^{k+1}$)
  (Ce « +1 » correspondant à une éventuelle retenue.)
- ⋆ Multiplier deux entiers naturels n et p avec respectivement au plus k et l bits significatifs en nécessite au plus $k + l$ (en effet : $n < 2^k$ et $p < 2^l$ donnent $n \times p < 2^k \times 2^l = 2^{k+l}$)

Un nombre codé sur n bits peut s'écrire de 2^n façons. n bits permettent donc de coder tous les **entiers relatifs** de $\texttt{10...0}_2 = -2^{n-1}$ à $\texttt{01...1}_2 = -2^{n-1} - 1$.

codage en	plus petit entier	plus grand entier
8 bits = 1 octet	−128	127
16 bits = 2 octets	−32 768	32 767
32 bits = 4 octets	−2 147 483 648	2 147 483 647
64 bits = 8 octets	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

- ⋆ Additionner un négatif avec un positif ne pose jamais de problème.
- ⋆ Ajouter deux positifs trop grands risque de donner un résultat négatif.
  (la retenue se déportant sur le bit de signe.)
- ⋆ Ajouter deux négatifs trop grands risque de donner un résultat positif (même raison).

Il est difficile de mettre en évidence ces dépassements avec Python qui définit des entiers de taille arbitraire. On peut en avoir un aperçu avec la calculatrice de windows en mode programmeur, si on essaie de calculer 9 223 372 036 854 775 807 + 1 on trouve −9 223 372 036 854 775 808, les entiers relatifs étant codés sur 64 bits.

3 - Codage des « réels »

Les nombres irrationnels (comme $\sqrt{2}$ ou π) *ne* peuvent *pas* être représentés car ils contiennent une infinité de chiffres significatifs sans répétition pour les décrire. Et même si c'était possible, n'importe quel intervalle, aussi petit soit-il, contient une infinité de réels donc impossible à distinguer avec un nombre fini de bits. Tout ce que l'on peut faire, c'est coder quelques nombres rationnels choisis.

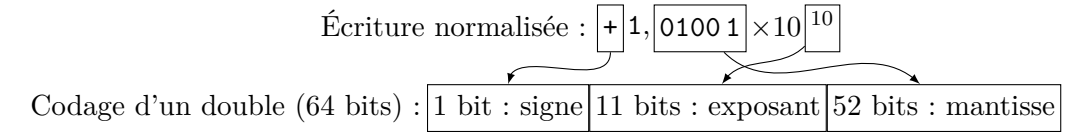
a) Codage en virgule flottante

L'écriture décimale d'un nombre se fait en deux parties finies séparées par une virgule. Par exemple : $324{,}125 = 3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$.

On peut utiliser la même notation en binaire :
$101{,}001_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 10^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} (= 5{,}125)$

L'écriture scientifique permet de standardiser l'écriture décimale : $324{,}125 = 3{,}241\,25 \times 10^2$

L'avantage pour un nombre binaire ($\neq 0$), est que le premier chiffre est toujours 1. Par exemple $101,001_2 = 1,01001_2 \times 10_2{}^{10_2}$. On parle d'écriture normalisée. Il suffit donc de se souvenir de la mantisse (chiffres après la virgule) et de l'exposant. Sans entrer plus dans les détails, voilà comment on procède :



b) Problème d'arrondis

On peut coder la partie décimale d'un nombre par soustractions successives des puissances de 2 d'exposant négatifs :

$0{,}625 = 0{,}125 + 1 \times 2^{-1}$ donc : $0{,}625 = \texttt{0,1?...}_2$
$0{,}125 = 0{,}125 + 0 \times 2^{-2}$ donc : $0{,}625 = \texttt{0,10?...}_2$
$0{,}125 = \quad 0 \quad + 1 \times 2^{-3}$ donc : $0{,}625 = \texttt{0,101}_2$

Si on essaie de déterminer l'écriture binaire de 0,1 :

$0{,}1 = 0{,}037\,5 + 1 \times 2^{-4}$ donc : $0{,}1 = \texttt{0,0001?...}_2$
$0{,}037\,5 = 0{,}006\,25 + 1 \times 2^{-5}$ donc : $0{,}1 = \texttt{0,00011?...}_2$
$0{,}006\,25 = 0{,}006\,25 + 0 \times 2^{-6}$ donc : $0{,}1 = \texttt{0,000110?...}_2$
$0{,}006\,25 = 0{,}006\,25 + 0 \times 2^{-7}$ donc : $0{,}1 = \texttt{0,0001100?...}_2$
$0{,}006\,25 = 0{,}002\,343\,75 + 1 \times 2^{-8}$ donc : $0{,}1 = \texttt{0,00011001?...}_2$
$0{,}002\,343\,75 = 0{,}000\,390\,625 + 1 \times 2^{-9}$ donc : $0{,}1 = \texttt{0,000110011?...}_2$
$0{,}000\,390\,625 = 0{,}000\,390\,625 + 0 \times 2^{-10}$ donc : $0{,}1 = \texttt{0,0001100110?...}_2$
$0{,}000\,390\,625 = 0{,}000\,390\,625 + 0 \times 2^{-11}$ donc : $0{,}1 = \texttt{0,00011001100?...}_2$
$0{,}000\,390\,625 = 0{,}000\,146\,484\,375 + 1 \times 2^{-12}$ donc : $0{,}1 = \texttt{0,000110011001?...}_2$
... $0{,}1 = \texttt{0,0001100110011001100110011001100110011001100110011001100110011...}_2$

Donc 0,1 *n'*est *pas* représentable par le codage binaire en virgule flottante !

L'écriture en Python : `x = 0.1` fait donc un arrondi (56 bits après la virgule).

Il n'est donc pas étonnant d'obtenir des résultats approchés par ce codage :

`print(0.1 + 0.2)` affiche `0.30000000000000004`

`print(0.1 + 0.2 == 0.3)` affiche `False`

Ce n'est donc *jamais* une bonne idée de faire des tests d'égalité avec des nombres à virgules flottantes. On préfèrera toujours, quand c'est possible, utiliser des entiers.

Types et valeurs de base(II)

Expressions booléennes

1 - Valeurs booléennes

L'ensemble des booléens comporte exactement deux valeurs : « vrai » ou « faux ».
Le type `bool` de Python les note `True` et `False`.
Les variables booléennes servent aux instructions conditionnelles.

```
if test :
    print('Le test est vrai')
else :
    print('Le test est faux')
```

```
while test :
    print('Le test est vrai')
    fait_un_truc()
```

Les autres types peuvent aussi être évalués comme un booléen :

```
for test in [True, False, 0, 1, 127, '0', 0.0, 12.34, [], [0], print] :
    if test:
        print(type(test),test,"= vrai.")
    else :
        print(type(test),test,"= faux.")
```

⋆ **FAUX** : `False`, 0 ou vide.

⋆ **VRAI** : Tout le reste (ce qui n'est pas faux est vrai).

Pour simplifier nous confondrons en particulier `False` avec 0 et `True` avec 1.
Dans un souci de lisibilité, on évitera d'écrire des conditionnelles d'un autre type.

2 - Opérateurs booléens

Comme il n'y a que deux valeurs booléennes, il suffit, pour décrire n'importe quelle fonction ou opérateur booléen, de dresser son tableau de valeurs, appelé : **table de vérité**.

a) Une variable

Il n'existe que $2^2 = 4$ tables de vérité portant sur une seule variable :

a	0	1
$f_0(a)$	0	0

a	0	1
$f_1(a)$	0	1

a	0	1
$f_2(a)$	1	0

a	0	1
$f_3(a)$	1	1

$f_0 = 0$ et $f_3 = 1$ et $f_1(a) = a$ n'ont pas besoin d'être nommées pour être utilisées.
f_2 par contre permet d'échanger `True` et `False` : c'est une négation.
Python la note `not` et elle s'utilise sans parenthèses. L'équivalent binaire est `~`.

a	False	True
not a	True	False

a	0b0	0b1
~a	0b1	0b0

b) Deux variables

Il existe $2^4 = 16$ fonctions à 2 variables booléennes. Les trois plus utilisées sont :

a	0	0	1	1
b	0	1	0	1
et	0	0	0	1

a	0	0	1	1
b	0	1	0	1
ou	0	1	1	1

a	0	0	1	1
b	0	1	0	1
ou exclusif	0	1	1	0

Python note « et » : `and` et « ou » : `or`, mais n'a pas de notation dédiée au « ou exclusif ». Les équivalents binaires sont respectivement `&`, `|` et `^`.

a	False	False	True	True
a	False	True	False	True
a and b	False	False	False	True

a	0b0	0b0	0b1	0b1
b	0b0	0b1	0b0	0b1
a & b	0b0	0b0	0b0	0b1

a	False	False	True	True
a	False	True	False	True
a or b	False	True	True	True

a	0b0	0b0	0b1	0b1
b	0b0	0b1	0b0	0b1
a | b	0b0	0b1	0b1	0b1

a	0b0	0b0	0b1	0b1
b	0b0	0b1	0b0	0b1
a ^ b	0b0	0b1	0b1	0b0

`not`, `and` et `or` sont appelés **opérateurs logiques**.
`~`, `&`, `|`, `^` sont des **opérateurs binaires**.
Comme les booléens sont identifiés aux binaires 0 et 1, on peut utiliser `^`.
`True ^ True` vaut bien `False`, `True ^ False` vaut bien `True`, etc...
Mais il faut faire attention aux priorités de calcul. Le calcul binaire est prioritaire par rapport au calcul booléen. Dans le doute : surcharger de parenthèses ; ne pas mélanger les types ; ou passer par une fonction (dont l'évaluation est encore plus prioritaire mais qui oblige les parenthèses).

On pourra par exemple définir une fonction `xor()` pour les booléens (ci-contre).

```
1 def xor(a, b):
2     return a ^ b
```

3 - Caractère séquentiel des opérateurs logiques

Pour évaluer `a and b`, Python évalue d'abord `a`.

⋆ S'il est `False`, quelque soit `b`, la réponse sera `False`, donc `b` n'est pas évalué.
C'est très pratique, par exemple quand on veut évaluer une valeur d'un tableau :
`if i < len(T) and T[i] > 0 :` si l'indice sort du tableau, `T[i]` ne sera pas évalué.

⋆ S'il est `True`, `b` détient la réponse : c'est sa valeur qui est renvoyée (sans être testée).
Cela permet un autre raccourci d'écriture :
`b = 0 <= i < len(T) and T[i]` vaut `False` si `i` sort du tableau et `T[i]` sinon.

Grossièrement `a and b` revient à :

```
1 def et(a,b):
2     if a :
3         resultat = b
4     else :
5         resultat = False
6     return resultat
```

De même, `a or b` revient à :

```
1 def ou(a,b):
2     if a :
3         resultat = True
4     else :
5         resultat = b
6     return resultat
```

On voit en particulier que `a and b` et `b and a` n'ont pas vraiment la même signification (alors que `a & b` et `b & a` si, en binaire)
En utilisant ces deux opérateurs, on arrive parfois à des instructions conditionnelles très courtes (qui ne sont spécialement à rechercher mais à savoir lire) comme :

```
parite = x % 2 == 0 and "pair" or "impair"
```

4 - Expressions booléennes

Une expression booléenne est une expression dont les variables sont booléennes, les opérateurs « non », « ou » et « et » avec éventuellement des parenthèses pour les priorités.

Exemple : `a and not b or not a and b`

Comme toute variable ne peut prendre que 2 valeurs, on peut toujours dresser un tableau de valeurs complet d'une expression booléenne, appelée table de vérité.

Priorité : `not` puis `and` puis `or`.

`a`	0	0	1	1
`b`	0	1	0	1
`not a`	1	1	0	0
`not b`	1	0	1	0
`a and not b`	0	0	1	0
`not a and b`	0	1	0	0
`a and not b or not a and b`	0	1	1	0

Donc `a and not b or not a and b` est une expression booléenne du « ou exclusif ».

Les **règles de calculs** (pour `a` et `b` booléens) :

- ⋆ Associativité :
  `(a or b) or c` = `a or (b or c)`
  `(a and b) and c` = `a and (b and c)`
- ⋆ Commutativité :
  `a or b` = `b or a`
  `a and b` = `b and a`
- ⋆ Distributivité :
  `(a or b) and c` = `(a and c) or (b and c)`
  `(a and b) or c` = `(a or c) and (b or c)`
- ⋆ Élément neutre :
  `a or False` = `a`
  `a and True` = `a`
- ⋆ Élément absorbant :
  `a or True` = `True`
  `a and False` = `False`
- ⋆ Négation :
  `a or not a` = `True`
  `a and not a` = `False`
- ⋆ Loi de De Morgan :
  `not (a or b)` = `not a and not b`
  `not (a and b)` = `not a or not b`

5 - Application au calcul binaire

On écrit ici 1 pour `True` et 0 pour `False` et on va réécrire les opérations binaires avec des formules logiques. Posons les additions à un bit :

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,1 \\ \hline 10 \end{array} \qquad \begin{array}{r} a \\ +\,b \\ \hline r\ s \end{array}$$

Il n'y a une retenue que si les deux bits sont à 1 : `r = a and b`

Sans la retenue, le résultat est 1 seulement si un seul bit est 1 : `s = xor(a,b)`

Pour additionner deux entiers écrits en binaire, il est nécessaire de tenir compte des retenues à chaque addition de deux bits :

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

		r2	r1	r0	0
		a3	a2	a1	a0
+		b3	b2	b1	b0
	r3	s3	s2	s1	s0

On veut ajouter `r0` à la somme de `a1` et `b1`.
Cela revient à l'ajouter à `xor(a1,b1)` avec une retenue `a1 and b1`.
On trouve donc `xor(r0,xor(a1,b1))` et une retenue si `a1 and b1` ou `r0 and xor(a1,b1))`.

`s1 = xor(r0,xor(a1,b1))` et `r1 = a1 and b1 or r0 and xor(a1,b1)`

```
def somme1(r,a,b) :
    return a and b or r and xor(a,b), xor(r,xor(a,b))
```

La somme de deux binaires serait alors :

```
def somme(a,b) :
    """ Calcule la somme des binaires a et b

    a et b sont sous la forme de tableaux de booléens de même taille,
    le bit de poids le plus faible en premier"""
    s = []
    r = False
    for k in range(len(a)):
        r, somme = somme1(r,a[k],b[k])
        s.append(somme)
    return r, s
```

Types et valeurs de base(III)

Représentation d'un texte

1940 À partir du moment où il est clairement apparu que les ordinateurs n'allaient plus servir à traiter que des informations numériques brutes, le problème du codage des caractères s'est posé.

1 - ASCII

ASCII : American Standart Code for Information Interchange :
C'est la première norme, adoptée après de nombreuses révisions, par l'« American National Standards Institute » (ANSI) dans les années 1960.

À l'époque les ordinateurs fonctionnaient en 8 bits, et, 1 bit de parité étant conservé pour la détection des erreurs de transmission, $2^7 = 128$ caractères étaient suffisants pour coder les lettres majuscules, minuscules, chiffres et ponctuations.

Le document ci-dessous date de 1972 :

USASCII code chart

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Source : https ://en.wikipedia.org/wiki/ASCII#/media/File :US-ASCII_code_chart.png

Par exemple, le caractère `'A'`, est le caractère numéro `100 0001`.
C'est-à-dire : $1\times2^6+0\times2^5+\ldots+0\times2^1+1\times2^0 = 65$ en décimal.
Le tableau permet d'en lire l'écriture hexadécimale : `41`.
On peut vérifier par le calcul : $4\times16^1+1\times16^0 = 65$.

Comme l'ASCII est la base de notre codage actuel, on obtient donc « A » indifféremment par `'A'` ou `chr(65)` ou `chr(0x41)` ou encore `chr(0b1000001)`, et son code par `ord('A')`.

2 - Formats ISO

L'amélioration de la fiabilité des transmissions et les besoins des européens, par exemple d'accentuer les lettres, ont abouti à l'utilisation du 8ème bit et donc à doubler le nombre de caractères codés.

latin-1

C'est la norme ISO 8859-1 pour l'encodage des caractères. Elle contient les caractères accentués utilisés dans les langues latines.

On peut citer également :

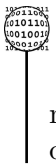
- ⋆ l'ISO 8859-7 qui contient les caractères grecs.
- ⋆ l'ISO 8859-15 ou latin 9, presque identique au latin 1 à 8 caractères près (le latin 9 contient par exemple les caractères œ, Œ et € qui ne sont pas dans la table latin 1).

L'encodage utilisé ne fait pas partie du contenu des fichier textes. Si on se trompe sur l'encodage utilisé, certains caractères seront donc mal interprétés. On voit encore apparaître des caractères sur des pages ou courriels dont l'encodage est mal spécifié. La phrase « C'est très raté ! » par exemple devient « C'est trÃ¨s ratÃ© ! ».

3 - Format Unicode

1990 Apparaît la norme Unicode, codée sur 2 octets (16 bits), pour réunir tous les caractères dans une seule table. On peut l'utiliser pour notre `'A'` : `'\u0041'`.

Cette norme est encore en construction aujourd'hui, et contient actuellement plus de 135 000 symboles codé sur 21 bits. Le code unicode n'est pas utilisé directement comme encodage de fichiers (sous peine de voir la taille des fichiers texte multipliée par 3). Ce sont des encodages comme UTF-8 qui permettent d'utiliser Unicode.



UTF-8

C'est un encodage des caractères sur un seul octet, qui utilise des séquences d'échappement pour accéder à d'autres parties de la table. En UTF-8, tous les caractères n'occupent donc pas la même place. Certains sont codés sur 1 octet, et d'autres sur 2 à 4 octets.

Pour expliquer le problème « C'est trÃ¨s ratÃ© ! », regardons comment est encodé « è » :
`'è'.encode('utf8')` donne `b'\xc3\xa8'`.
On voit que le caractère d'échappement a le code hexadécimal : `C3`.
S'il n'était pas considéré comme échappement il serait `chr(0xc3)` qui donne `Ã`

Et pour le « é » ?
`'é'.encode('utf8')` donne `b'\xc3\xa9'`. C'est le même échappement.

Pour le « € » il faut aller plus loin, l'échappement est différent :
`'€'.encode('utf8')` donne `b'\xe2\x82\xac'`.

Actuellement, la bonne pratique est d'utiliser UTF-8 dès que c'est possible.

```
# Supposons que fichier.txt soit encodé en latin1 !
with open('fichier.txt', encoding="latin1") as fichier :
    text = fichier.read()
with open('fichier.txt', mode="w", encoding="utf8") as fichier :
    fichier.write(text)
# Le problème est réglé.
```

Types construits

- Programme officiel

À partir des types de base se constituent des types construits, qui sont introduits au fur et à mesure qu'ils sont nécessaires.

Il s'agit de présenter tour à tour les p-uplets (tuples), les enregistrements qui collectent des valeurs de types différents dans des champs nommés et les tableaux qui permettent un accès calculé direct aux éléments. En pratique, on utilise les appellations de Python, qui peuvent être différentes de celles d'autres langages de programmation.

Contenus	Capacités attendues	Commentaires
p-uplets. p-uplets nommés	Écrire une fonction renvoyant un p-uplet de valeurs.	
Tableau indexé, tableau donné en compréhension	Lire et modifier les éléments d'un tableau grâce à leurs index. Construire un tableau par compréhension. Utiliser des tableaux de tableaux pour représenter des matrices : notation a [i] [j]. Itérer sur les éléments d'un tableau.	Seuls les tableaux dont les éléments sont du même type sont présentés. Aucune connaissance des tranches (slices) n'est exigible. L'aspect dynamique des tableaux de Python n'est pas évoqué. Python identifie listes et tableaux. Il n'est pas fait référence aux tableaux de la bibliothèque NumPy.
Dictionnaires par clés et valeurs	Construire une entrée de dictionnaire. Itérer sur les éléments d'un dictionnaire.	Il est possible de présenter les données EXIF d'une image sous la forme d'un enregistrement. En Python, les p-uplets nommés sont implémentés par des dictionnaires. Utiliser les méthodes keys(), values() et items().

Types construits(I)

Tableaux indexés

1 - Définition et lecture

Tableau

D'un point de vue algorithmique, un *tableau* est une structure de données définie par une séquence finie d'éléments de même type, auxquels on accède par leur position (entière positive) dans la séquence, appelé indice.

Le langage Python n'implémente pas directement ce type. On utilisera le type `list`, plus permissif (par exemple qui autorise des valeurs de types différents).

- ⋆ On peut définir un tableau par la séquence de ses valeurs : `Tableau = [15,18,20]`.
- ⋆ Sa taille est donnée par la fonction `len()` : `len(Tableau)` vaut 3.
- ⋆ On accède à chaque valeur par son indice (entre 0 et 2) : `Tableau[1]` vaut 18.
- ⋆ Ses valeurs sont modifiables (le type tableau est mutable) : `Tableau[1]=12`.
- ⋆ C'est comme si on avait trois variables : `Tableau[0]`, `Tableau[1]` et `Tableau[2]`.

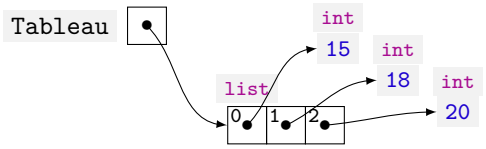
Représentation humaine :

Tableau	15	18	20

Implémentation (liste Python) :

`Tableau = [15,18,20]`.

Représentation en mémoire :



On peut obtenir des représentations sur : http://pythontutor.com/live.html#mode=edit.

2 - Itération

Pour parcourir les valeurs d'un tableau, on peut utiliser leur indice.

Par exemple le code ci-contre affiche les valeurs du tableau en passant à la ligne après chaque bloc de quatre valeurs.

```
1 valeur = [0,1,1,0,1,0,1,0,1,0,1,1]
2 for i in range(len(valeur)):
3     print(valeur[i], end = '')
4     if i % 4 == 3 : print()
```

Mais les indices peuvent ne pas avoir d'importance et seulement surcharger l'écriture du code. Dans ce cas, on peut itérer directement sur les éléments du tableau :

Avec les indices :

```
1 valeur = [10,13,15,11,12]
2 somme = 0
3 for i in range(len(valeur)):
4     somme = somme + valeur[i]
```

Sans les indices :

```
1 valeur = [10,13,15,11,12]
2 somme = 0
3 for val in valeur:
4     somme = somme + val
```

Pour aller plus loin : Il peut se produire des cas où l'on aimerait avoir les deux avantages. (Notamment si l'itération ne porte pas sur un tableau mais sur un autre itérable dont on ne sait pas déterminer facilement la taille.) On peut les obtenir avec la syntaxe suivante :

```
1 valeur = [10,13,15,11,12]
2 for (i,val) in enumerate(valeur):
3     print('valeur[', i, '] = ',val, sep='')
```

3 - Tableaux en compréhension

On peut construire une liste Python à l'aide de la méthode `append()`.

Par exemple pour dresser le tableau des 10 premiers carrés d'entiers naturels, on peut écrire le code Python ci-contre.

```
carre = []
for x in range(10):
    carre.append(x ** 2)
```

Il existe une manière condensée de faire la même chose : `carre = [x**2 for x in range(10)]`

Dans l'exemple ci-contre, on utilise cette façon de faire pour appliquer un traitement à tous les éléments d'un autre tableau.

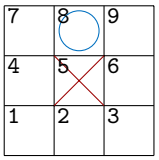
```
lettres = ['a', 'e', 'i', 'o', 'u']
lettres = [ord(l) for l in lettres]
```

On peut également imposer une condition à l'inclusion de la valeur :
`chiffresPairs = [n for n in range(10) if n%2 == 0]`

Tableau en compréhension
On définit un tableau en compréhension en utilisant une description de ses termes.
Avec Python, elle prend la forme suivante :
`[fonction(item) for item in iterable if condition(item)]`

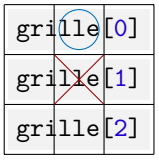
4 - Tableau de tableaux

L'idée des tableaux de tableaux est de pouvoir représenter des tableaux à 2 dimensions. Par exemple, la grille d'un jeu de morpion est une grille 3×3. Voici deux codages possibles :



- ⋆ On fait correspondre un numéro de case à un indice :
  `[-1,0,0,0,0,1,0,0,2,0]` (1 : joueur 1 et 2 : joueur 2)
- ⋆ On dresse la liste des cases jouées, la parité des indices donne le joueur :
  `[5,8,0,0,0,0,0,0,0]`

Voici maintenant une autre possibilité, utilisant un tableau de tableaux :



- ⋆ La grille est un tableau de lignes :
  `grille[0] = [0,2,0]`, `grille[1] = [0,1,0]`, `grille[2] = [0,0,0]`
  C'est à dire `grille = [ [0,2,0], [0,1,0], [0,0,0] ]`
  Et on accède par exemple à la valeur 2 par : `grille[0][1]`.

Ce ne sont évidemment pas les seules options (chaque élément du tableau pourrait être une colonne, ou avec des indices dans l'autre sens, etc.)

On peut écrire des tableaux de tableaux en compréhension :

```
grille = [[3 * ligne + colonne for colonne in range(3)] for ligne in range(3)]
```

Et on peut itérer sur ses indices ou ses éléments par une double boucle :

```
for i in len(grille) :
    for j in len(grille[i]) :
        grille[i][j] = 0
```

```
for ligne in grille :
    for valeur in ligne :
        valeur = 0
```

Types construits(II)

p-uplets (tuples) et Dictionnaires

Le chapitre précédent présentait les tableaux : une structure de données de taille fixe, dont les éléments sont de même type, ordonnés et accessibles par leur indice. Comme de plus ils peuvent être modifiés, on dit que c'est un type *mutable*. On utilise les listes pour les représenter car elles ont des propriétés similaires avec certaines restrictions en moins. Mais d'autres types s'adaptent à d'autres situations :

type construit	tableau	`list`	`tuple`	`dict`
valeurs de même type	✓	X	X	X
ordonné et indicé	✓	✓	✓	X
champs nommés	X	X	X	✓
mutable	✓	✓	X	✓
taille fixe	✓	X	✓	X

Il existe aussi des p-uplets nommés, qui ont les mêmes caractéristiques que les p-uplets mais dont on peut nommer des champs. Dans la pratique au lycée, nous utiliserons préférentiellement les dictionnaires dans ce cas.

1 - p-uplets

a) Définition

p-uplets

Un *p-uplet* est une structure de données définie par une séquence finie d'éléments non modifiables, auxquels on accède par leur position (entière positive) dans la séquence, appelé indice.

Un 2-uplet est un couple, par exemple (latitude, longitude). Un 3-uplet est un triplet, par exemple les coordonnées d'un point dans l'espace. Un 4-uplet est un quadruplet, etc.

b) Fonctionnalité

Le langage Python implémente ce type sous la dénomination `tuple`.

- ⋆ On peut définir un p-uplet par la séquence de ses valeurs : `point = ('A',3)`.
- ⋆ Les parenthèses ne sont pas obligatoires, on peut aussi écrire : `point = 'A',3`.
- ⋆ Pour un 1-uplet, l'absence de virgule ne permettrait pas de savoir si les parenthèses sont de simples parenthèses de calcul, ou la définition d'une structure. Python permet de terminer la séquence de valeurs par une virgule pour lever toute ambiguïté : `point = 'A',3,`, et pour un 1-uplet : `lettre = 'A',`.
- ⋆ Pour un 0-uplet, cette notation ne suffit plus. On utilise alors : `vide = tuple()`.
- ⋆ C'est, pour les mêmes raisons, la notation à utiliser pour une définition en compréhension : `chiffresPairs = tuple(n for n in range(10) if n%2 == 0)`
- ⋆ Sa taille est donnée par la fonction `len()` : `len(point)` vaut 2.
- ⋆ On accède à chaque valeur par son indice : `point[1]` vaut 3.
- ⋆ Ses valeurs ne sont pas modifiables : `point[1]=12` provoquera une erreur.
- ⋆ C'est comme si on avait deux constantes : `point[0]` et `point[1]`.
- ⋆ On peut *itérer* un p-uplet.

Avec les indices :

```
valeur = (10,13,15,11,12)
somme = 0
for i in range(len(valeur)):
    somme = somme + valeur[i]
```

Sans les indices :

```
valeur = (10,13,15,11,12)
somme = 0
for val in valeur:
    somme = somme + val
```

c) Utilisation

Le fait que ce type ne soit pas mutable peut être la propriété recherchée.
Cela fait des données des constantes.

Les éléments d'un p-uplet peuvent être des noms de variables (ou de fonctions).
C'est ce qui permet les affections multiples donc les écritures comme :

```
x, y = 1, 2 # Le tuple n'est pas mutable, la variable x est toujours la variable x.
x, y = y, x # C'est la valeur que pointe (représente) x qui change !
```

On utilise cette structure pour permettre à une fonction de renvoyer plusieurs valeurs.

```
def divisionEuclidienne(x,y):
    return x//y, x%y

quotient, reste = divisionEuclidienne(7,2)
```

Pour aller plus loin : Unpacking

On peut aussi vouloir se servir de cette structure pour passer un nombre indéterminé de valeurs à une fonction :

```
def produit(valeurs):
    produit = 1
    for val in valeurs:
        produit = produit * val
    return produit

p = produit((1,2,3))
```

Mais cela surcharge l'écriture de l'appel de la fonction.
On utilise alors l'opérateur *splat* `*` pour récupérer les arguments en surnombre dans un seul un p-uplet :

```
def produit(*valeurs):
    produit = 1
    for val in valeurs:
        produit = produit * val
    return produit

p = produit(1,2,3)
```

Dans l'appel de chacune des fonctions précédentes, la variable locale `valeurs` vaut la même chose : le triplet `(1,2,3)`.

Cet opérateur permet de « dépacker » un tuple. Si par exemple on dispose d'un n-uplet et qu'on souhaite l'utiliser pour les dernières valeurs d'un (n+1)-uplet :

```
coord = (3, -5)
point = ('A',*coord)  # revient à point = ('A', 3, -5)
```

2 - Dictionnaires

a) Définition

Dictionnaire
Un dictionnaire est une structure de données définie par un ensemble d'éléments, auxquels on accède par mot clé.

Exemple : `{'nom' : 'A', 'x' : 3, 'y' :-5}` ou `{'nom' : 'A', 'coord' : (3, -5)}` ou encore `{(3, -5) : 'A'}`...

b) Fonctionnalité

Le langage Python implémente ce type sous la dénomination `dict`.

- ⋆ il est défini par l'ensemble de ses éléments (clé : valeurs) : `point = {'x' : 3, 'y' :-5}`. Une clé doit être unique (une clé comme un indice de liste ne référence qu'une valeur).
- ⋆ Le dictionnaire vide s'écrit : `vide = dict()`.
- ⋆ Un dictionnaire en compréhension s'écrit : `{n : n ** 3 for n in range(10) if n > 0}`
- ⋆ Sa taille est donnée par la fonction `len()` : `len(point)` vaut 2.
- ⋆ La liste des clés est `list(point.keys())`.
- ⋆ La liste des valeurs est `list(point.values())`.
- ⋆ La liste des couples (clé, valeur) est `list(point.items())`.
- ⋆ On accède à chaque valeur par sa clé : `point['x']` vaut 3.

  Si une clé peut ne pas être présente, on utilise la méthode `get()` :

  `point.get('z',0)]` renvoie `point['z']` si `'z'` est bien une clé ; `0` sinon (par défaut).

  (On peut tester la présence d'une clé avec la méthode `has_key()`.)
- ⋆ C'est comme si on avait plusieurs variables : `point['x']`, `point['y']`.
- ⋆ On peut *itérer* un dictionnaire par ses clés :

  De manière implicite :
  ```
  for clef in point :
      print(clef, ':', point[clef])
  ```
  De manière explicite :
  ```
  for clef in point.key() :
      print(clef, ':', point[clef])
  ```
- ⋆ On peut itérer sur les valeurs :
  ```
  for valeur in point.values() :
      print(valeur)
  ```
- ⋆ On peut itérer sur les éléments :
  ```
  for clef, valeur in point.items() :
      print(clef, ':', valeur)
  ```

c) Exemple d'utilisation

Cet exemple suppose que la bibliothèque `pillow` soit installée, et la présence d'une image dans le dossier de travail.

Les données EXIF d'une image sont de la forme `clé-numérique : valeur`.

Pillow contient un dictionnaire `TAGS` d'éléments `clé-numérique : signification`.

```
from PIL import Image
from PIL.ExifTags import TAGS

with Image.open('image.jpg') as img:
    exif = img._getexif()
    for tag in exif :
        print(TAGS.get(tag, tag),':',exif[tag])
```

Traitement de données en tables

- Programme officiel

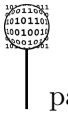
Les données organisées en table correspondent à une liste de p-uplets nommés qui partagent les mêmes descripteurs. La mobilisation de ce type de structure de données permet de préparer les élèves à aborder la notion de base de données qui ne sera présentée qu'en classe terminale. Il s'agit d'utiliser un tableau doublement indexé ou un tableau de p-uplets, dans un langage de programmation ordinaire et non dans un système de gestion de bases de données.

Contenus	Capacités attendues	Commentaires
Indexation de tables	Importer une table depuis un fichier texte tabulé ou un fichier CSV.	Est utilisé un tableau doublement indexé ou un tableau de p-uplets qui partagent les mêmes descripteurs.
Recherche dans une table	Rechercher les lignes d'une table vérifiant des critères exprimés en logique propositionnelle.	La recherche de doublons, les tests de cohérence d'une table sont présentés.
Tri d'une table	Trier une table suivant une colonne.	Une fonction de tri intégrée au système ou à une bibliothèque peut être utilisée.
Fusion de tables	Construire une nouvelle table en combinant les données de deux tables.	La notion de domaine de valeurs est mise en évidence.

Traitement de données en tables(I)

Indexation de tables

1 - Open data

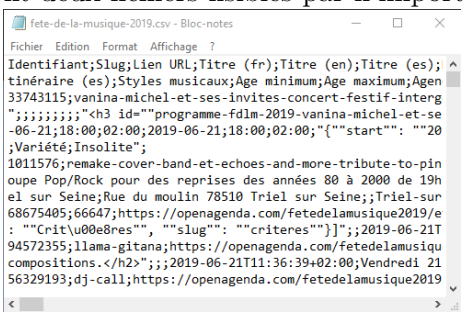


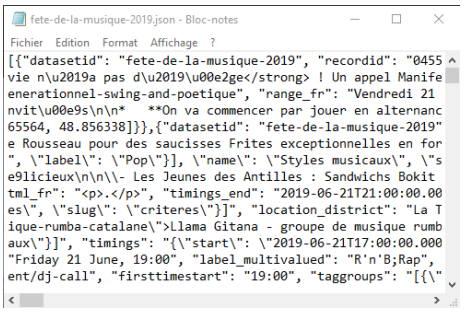
Open data

Un contenu est dit en *open data* s'il est librement utilisable, modifiable et partageable par n'importe qui et pour n'importe quelle raison.

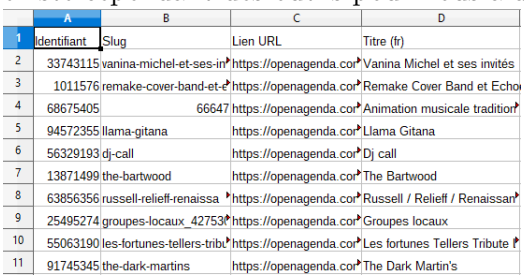
- ⋆ En France, Etalab s'occupe de data.gouv.fr, site de partage des données publiques.
- ⋆ Des données sont également disponibles localement, par exemple : data.larochesuryon.fr.
- ⋆ Certaines données sont collaboratives, par exemple : fr.openfoodfacts.org autour des produits alimentaires ou openstreetmap.fr sur la cartographie.

Les formats de données les plus courants sont CSV et JSON.
Prenons l'exemple de https://www.data.gouv.fr/fr/datasets/fete-de-la-musique-2019/
Ce sont deux fichiers lisibles par n'importe quel éditeur de texte...

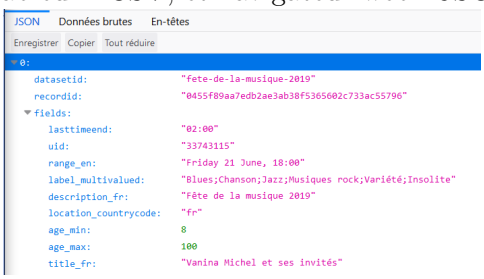




mais difficilement pour un humain, il y a un peu de travail !
Il existe cependant des outils pour nous aider (tableur : CSV, et navigateur web : JSON) :



	A	B	C	D
1	Identifiant	Slug	Lien URL	Titre (fr)
2	33743115	vanina-michel-et-ses-in	https://openagenda.cor	Vanina Michel et ses invités
3	1011576	remake-cover-band-et-e	https://openagenda.cor	Remake Cover Band et Echo
4	68675405	66647	https://openagenda.cor	Animation musicale tradition
5	94572355	llama-gitana	https://openagenda.cor	Llama Gitana
6	56329193	dj-call	https://openagenda.cor	Dj call
7	13871499	the-bartwood	https://openagenda.cor	The Bartwood
8	63856356	russell-relieff-renaissa	https://openagenda.cor	Russell / Relieff / Renaissan
9	25495274	groupes-locaux_42753	https://openagenda.cor	Groupes locaux
10	55063190	les-fortunes-tellers-tribu	https://openagenda.cor	Les fortunes Tellers Tribute
11	91745345	the-dark-martins	https://openagenda.cor	The Dark Martin's



Si on regarde la première ligne du JSON :
`[{"datasetid": "fete-de-la-musique-2019", "recordid": "0455f89aa7edb2ae3···`
on peut reconnaitre les formatage de listes et de dictionnaires Python. C'est ça qui permet de comprendre ce format. Si on regarde les deux premières lignes du CSV :
`Identifiant;Slug;Lien URL;Titre (fr);Titre (en);Titre (es);···`
`33743115;vanina-michel-et-ses-invi···ique;https://open···`
La structure est beaucoup plus pauvre :

- ⋆ Il n'y a qu'un séparateur (ici le « ; »).
- ⋆ La première ligne est réservée aux *descripteurs*.
- ⋆ Chacune des lignes suivantes correspond à une donnée définie par les valeurs correspondantes à ces descripteurs.

Le but de cette partie est de traiter les données en tables donc au format CSV. Pour le format JSON, comme dit plus haut, on utilise un dictionnaire.

2 - Lecture/écriture d'un fichier texte

Une table CSV est avant tout un fichier texte avec un certain format.
Le langage Python dispose d'une fonction `open()` pour ouvrir un fichier en mode texte.
Il dispose également d'une méthode `close()` pour le fermer :

```
fichier = open('fichier.txt', 'w')
for i in range(1,4):
    fichier.write('Ligne '+srt(i)+'\n')
fichier.close()
```

L'argument `'w'` pour « write » ouvre (quitte à le créer) le fichier en écriture. Cet argument est `'r'` pour « read » (ouverture en lecture) par défaut, et peut aussi être `'a'` pour « append » (ouverture en ajout).

Pour éviter les problèmes de non-fermeture de fichier ouvert (si par exemple l'exécution du programme s'interrompt avant) on utilise le mot clé `with` qui gère le problème :

```
with open('fichier.txt') as fichier :
    print(fichier.read())
```

- ⋆ `fichier.read()` revoit une chaine de caractère contenant le fichier entier.
- ⋆ `fichier.readlines()` aurait séparé chaque ligne dans une liste.
- ⋆ On peut aussi itérer sur le fichier pour ne charger qu'une ligne en mémoire à la fois :

```
with open('grosfichier.txt') as fichier :
    for ligne in fichier :
        print(ligne)
```

Supposons disposer du fichier `notes.txt` contenant les trois lignes ci-dessous.

```
Nom;DS1;DS2;DS3;DS4;DS5
Bob;10;12;5;14;13
Rick;12;10;ABS;12;11
```

Si on perçoit les données comme un tableau à deux dimensions, on peut les extraire comme présenté ci-contre.

`ligne.rstrip('\r\n')` permet de nettoyer les fins de lignes et `ligne.split(';')` permet de créer une liste en séparant la chaîne au niveau des `';'`.

```
# Extraction
notes = []
with open('notes.txt') as fnotes :
    for ligne in fnotes :
        ligne = ligne.rstrip('\r\n')
        notes.append(ligne.split(';'))

# Génération de l'index
index = {}
for (i, idx) in enumerate(notes[0]) :
    index[idx] = i
del(notes[0])
```

La première ligne contenait les index. On répertorie leurs indices dans le dictionnaire `index` (`= {'Nom' : 0, 'DS1' : 1,` ...) et on la supprime du tableau de données `del(notes[0])`.

A présent la note de Bob au DS3 est `notes[0][index['DS3']]`.

3 - Importer une table CSV

Les séparateurs `';'` et `'\r\n'` ne sont pas les seuls possibles. De plus, une des données à séparer pourrait être un texte comportant ces symboles... Une « bonne » écriture d'extraction

d'un fichier CSV est donc bien plus délicate à produire. Heureusement elle existe déjà dans la bibliothèque du même nom.

Avec notre fichier `notes.txt`, cela donne :

```
# Extraction
import csv

notes = []
with open('notes.csv') as fnotes:
    csvnotes = csv.reader(fnotes, delimiter=';')
    for ligne in csvnotes:
        notes.append(ligne)
```

Les traitements comme `ligne.replace('\n','')` ou `ligne.split(';')` sont tous contenues dans `csv.reader()` et le rendu correspond aux listes attendues.

On peut avoir besoin de préciser l'encodage du texte en argument de la fonction `open()`. Pour le fichier `fete-de-la-musique-2019.csv` donné en exemple au début on pourra écrire : `open('fete-de-la-musique-2019.csv', encoding='utf8')`

Pour aller plus loin : Avec un fichier JSON, on peut utiliser la bibliothèque du même nom, et on obtient une liste de dictionnaires avec :

```
import json

with open('fete-de-la-musique-2019.json', encoding='utf8') as ffete:
    fete = json.loads(ffete.read())
```

Traitement de données en tables(II)

Manipulation de tables

Il existe des modules pour le traitement de données issues de tables, comme `pandas`. Le parti pris pour la suite est de ne pas les utiliser. On considèrera donc les « tables » de données comme étant des listes de listes ou de p-uplets Python (on peut inclure les dictionnaires, si l'ordre n'a pas d'importance).

L'exemple utilisé vient du fichier `fete-de-la-musique-2019.csv`, téléchargeable sur :
https://www.data.gouv.fr/fr/datasets/fete-de-la-musique-2019/

Et on suppose avoir fait l'indexation suivante :

```
import csv

table = []
with open('fete-de-la-musique-2019.csv', encoding='utf8') as ffete:
    csvfete = csv.reader(ffete, delimiter=';')
    for ligne in csvfete:
        table.append(ligne)

index = {}
for (i, idx) in enumerate(table[0]) :
    index[idx] = i
del(table[0])
```

1 - Recherche dans une table

En open data, on a souvent de gros fichiers, apportant beaucoup plus d'informations que ce que l'on cherche réellement. Il est donc important de pouvoir les sélectionner. Sur l'exemple de la fête de la musique, on peut vouloir ne s'intéresser qu'à une ville. On parcourt toutes les lignes de données et on récupère celles qui satisfont à ce critère :

```
laRocheSurYon = []
for ligne in table :
    if ligne[index['Ville']] == 'La Roche-sur-Yon' :
        laRocheSurYon.append(ligne)
```

Il suffit de pouvoir exprimer ce critère avec des expression booléennes (en logique propositionnelle) : on emploie à loisir les `and`, `or`, `not`...

On peut également vouloir filtrer certains index et donc supprimer certaines colonnes. Pour ce faire, on peut par exemple reconstruire une liste en sélectionnant ceux que l'on veut garder :

```
simple = []
for ligne in laRocheSurYon :
    simple.append([ ligne[index['Titre (fr)']],
                    ligne[index['Nom du lieu']],
                    ligne[index['Première ouverture']] ])
```

Une fois ces sélections effectuées (ou d'autres opérations), il peut arriver que certaines lignes soient identiques (par exemple, s'il ne reste plus que les valeurs de `Première ouverture` dans chaque ligne). On peut alors avoir besoin d'identifier/supprimer ces doublons.

On peut pour cela parcourir les données en recopiant celles qui ne l'ont pas déjà été :

```
unique = []
for ligne in simple:
    if ligne not in unique:
        unique.append(ligne)
```

On peut améliorer ce code si la table était triée (les valeurs identiques seraient alors côte à côte.)

Tests de cohérence ?

2 - Tri d'une table

Des algorithmes de tri sont expliqués dans la partie algorithmique et peuvent bien sûr être appliqués. Pour ne pas surcharger, on propose ici d'utiliser la méthode `sort()` qui s'applique à une liste pour la trier selon une clé. Cette clé est une fonction qui prend en paramètre l'élément à trier et renvoie la valeur de cet élément dans ce tri :

```
# On va trier selon la colonne d'indice 2
def valeurPourTrier(ligne):
    return ligne[2]

simple.sort(key = valeurPourTrier)
```

On peut trier selon plusieurs critères, en faisant renvoyer un p-uplet comme clé.

On obtient un ordre total selon toutes les colonnes dans l'ordre si on ne précise pas de clé. (L'amélioration de la suppression de doublon est alors possible...)

3 - Fusion de tables

On suppose disposer de deux tables comportant un champ en commun. Leur fusion consiste à les réunir en une seule table regroupant toutes les informations.

Prenons un exemple simple de deux tables déjà chargées et filtrées :

```
index1 = ['Prénom', 'age']
table1 = [['Pierre', 12],
          ['Paulette', 14],
          ['Jack', 13]]
```

```
index2 = ['nom', 'Prénom']
table2 = [['Dupond','Pierre'],
          ['Pagnol','Pauline'],
          ['Potter','Jack']]
```

Le champ commun est `'Prénom'` .

Un premier algorithme pour répondre à cette demande :

On prépare la fusion en créant les champs nécessaires à partir d'une de table,

```
index = [*index2, index1[1]]
fusion = []
for i in range(len(table2)):
    fusion.append([*table2[i],-1])
```

puis on parcourt l'autre table en cherchant pour chaque enregistrement s'il y en a un qui correspond pour les fusionner.

```
14 for i in range(len(table1)):
15     fusionOK = False
16     for j in range(len(fusion)):
17         if fusion[j][1] == table1[i][0]:
18             fusion[j][2] = table1[i][1]
19             fusionOK = True
20     if not fusionOK :
21         fusion.append(['',*table1[i]])
```

⋆ S'il y avait eu deux prénoms identiques, la construction serait faussée.
Il est important que le champ commun identifie clairement chaque enregistrement.
Il est sinon possible de prendre plusieurs champs commun pour assurer cette unicité.

⋆ On voit que la première partie complète le champ vide avec `-1` alors que dans la seconde, il est complété avec `''`. Il est important pour des traitements ultérieurs de conserver le type de donnée associé à un champ. En proposant `-1` pour un âge qui devrait être positif et `''` pour un Nom qui devrait être au moins un caractère, on se place juste en dehors du domaine de valeurs du champ, pour spécifier qu'il n'est pas connu.

Si les deux tables étaient triées selon la valeur du champ commun, un algorithme de fusion beaucoup plus efficace existe : il parcourt les deux tables en même temps et ajoute à la fusion, à chaque étape, l'enregistrement le plus petit s'ils diffèrent entre les deux tables, ou leur fusion s'ils sont identiques.

Interactions entre l'homme et la machine sur le Web

- Programme officiel

Lors de la navigation sur le Web, les internautes interagissent avec leur machine par le biais des pages Web.

L'Interface Homme-Machine (IHM) repose sur la gestion d'événements associés à des éléments graphiques munis de méthodes algorithmiques.

La compréhension du dialogue client-serveur déjà abordé en classe de seconde est consolidée, sur des exemples simples, en identifiant les requêtes du client, les calculs puis les réponses du serveur traitées par le client.

Il ne s'agit pas de décrire exhaustivement les différents éléments disponibles, ni de développer une expertise dans les langages qui permettent de mettre en œuvre le dialogue tels que PHP ou JavaScript.

Contenus	Capacités attendues	Commentaires
Modalités de l'interaction entre l'homme et la machine Événements	Identifier les différents composants graphiques permettant d'interagir avec une application Web. Identifier les événements que les fonctions associées aux différents composants graphiques sont capables de traiter.	Il s'agit d'examiner le code HTML d'une page comprenant des composants graphiques et de distinguer ce qui relève de la description des composants graphiques en HTML de leur comportement (réaction aux événements) programmé par exemple en JavaScript.
Interaction avec l'utilisateur dans une page Web	Analyser et modifier les méthodes exécutées lors d'un clic sur un bouton d'une page Web.	
Interaction client-serveur. Requêtes HTTP, réponses du serveur	Distinguer ce qui est exécuté sur le client ou sur le serveur et dans quel ordre. Distinguer ce qui est mémorisé dans le client et retransmis au serveur. Reconnaître quand et pourquoi la transmission est chiffrée.	Il s'agit de faire le lien avec ce qui a été vu en classe de seconde et d'expliquer comment on peut passer des paramètres à un site grâce au protocole HTTP.
Formulaire d'une page Web	Analyser le fonctionnement d'un formulaire simple. Distinguer les transmissions de paramètres par les requêtes POST ou GET.	Discuter les deux types de requêtes selon le type des valeurs à transmettre et/ou leur confidentialité.

IHM sur le Web(I)

Les langages : HTML, CSS et JavaScript

1 - Balises HTML

HTML : HyperText Markup Language
C'est un langage de description de page web.
Un fichier HTML est un fichier texte, qui utilise des *balises* pour définir la structure et le contenu d'une page Web (indépendamment de la façon de l'afficher).

Tout fichier HTML en français, devrait contenir au moins les dix lignes proposées ci-dessous.

Les éléments commençant et terminant par les chevrons `<` et `>` sont des *balises*.

- ⋆ `<balise>` est une balise ouvrante,
- ⋆ `</balise>` une balise fermante et
- ⋆ `<balise />` une balise auto-fermante.

Toute balise ouverte doit être fermée.
Pour afficher les caractères < et > on utilise les entités : `<` et `>`.

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <title><!-- Un titre --></title>
  </head>
  <body>
    <!-- Du contenu -->
  </body>
</html>
```

Quelques balises « indispensables » :

Structurer le contenu	
`<h1>`	Titre principal
`<h2>`	Titre secondaire
⋮	⋮
`<h6>`	Plus petit sous-titre
`<p>`	Paragraphe
` `	Retour à la ligne

```
<h1>Langage HTML</h1>

<h2>Les balises</h2>

<p>Ce sont les éléments commençant par
  &lt; et terminant par &gt;.<br />
  Exemple : &lt;br /&gt;</p>
```

Les balises `<p>` et `<h1>`, ... sont de type *block* : leur contenu prend toute la largeur disponible sur la page. Elles provoquent donc automatiquement un passage à la ligne. La balise générique de ce type est : `<div>`

Faire des liens	
`<a>`	Lien hypertexte
`<img />`	Inclure une image

```
<p>Cliquez l'image ci-dessous pour
  aller sur mon site.<br />
  <a href="www.monsite.fr">
    <img src="../images/logo.png" />
  </a></p>
```

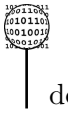
Les balises `<a>` et `<img />`, ... sont de type *inline* : leur contenu prend seulement la place dont il a besoin (pas de passage à la ligne). La balise générique de ce type est : `<span>`

Faire une liste	
`<ul>`	Liste non numérotée
`<ol>`	Liste numérotée
`<li>`	Élément d'une liste

```
<ul>
  <li> les balises <em>block</em></li>
  <li> les balises <em>inline</em></li>
</ul>
```

La balise `<em>` permet de mettre du texte en valeur (emphase). Il existe beaucoup d'autres balises dont on trouve facilement des listes et descriptions par exemple sur www.w3schools.com/.

2 - Éléments de CSS



CSS : Cascading Style Sheets
Un fichier CSS, appelé souvent « feuille de style » est un fichier texte permettant de décrire la manière d'afficher ce qui est défini dans une page HTML.

Pour lier la feuille de style `style.css`, on inclut un lien dans l'entête de la page :

```
<head>
  <!-- entête -->
  <link rel="stylesheet" type="text/css" href="chemin/style.css"/>
</head>
```

La feuille de style permet alors de décrire l'affichage de chaque balise.

Par exemple on peut préciser la couleur d'un élément par un nom (dans une liste prédéfinie), par ses composantes `rgb` ou par son code hexadécimal.

```
/* ---- généralités --- */
body {
  background-color : AliceBlue;
  color : rgb(128,128,255);
  border-color : #A1BBFF;
  border-style : solid;
}
```

Pour associer un style particulier à une seule balise du fichier HTML, on associe à cette balise un identifiant unique par le sélecteur `id` :

```
<p id="particulier">Ce paragraphe
  est traité de manière très
  particulière.</p>

<p>Celui-ci ne change pas </p>
```

```
/* ---- cas particulier --- */
#particulier {
  text-align : center;
  font-size : 1.5vw; /* ou px ou em*/
}
```

Pour associer un style particulier à plusieurs balises (éventuellement différentes) du fichier HTML, on associe à ces balises une classe commune par le sélecteur `class` :

```
<h2 class="important"> Ce titre est
important.</h2>
<p> Voila pourquoi </p>
<p class="important">Ce paragraphe
  est important.</p>
```

```
/* ---- classe --- */
.important {
  text-decoration : underline;
  font-weight : bold;
}
```

On peut préciser le style d'une balise selon son état. En particulier pour les liens :

```
Pour la définition, voir le
<a href="mapage.html#chapitre1">
chapitre 1</a>.
```

```
/* ---- états --- */
a:link { color: red; }
a:visited { color: green; }
a:hover { color: hotpink; }
a:active { color: blue; }
```

L'état `:hover`, en particulier, est utilisable pour les autres balises.

3 - Notions de JavaScript

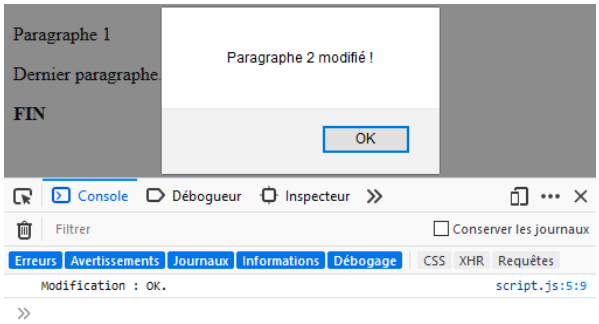
JavaScript
C'est un langage de programmation, interprété par les navigateurs web.

`<script>/* instructions */</script>` permet d'inclure des instructions JavaScript dans un document html. Mais on privilégie l'inclusion de fichiers : `<script src="script.js"></script>` pour garder la lisibilité de structure HTML. On peut, en sortie, écrire dans

- ⋆ la console du navigateur avec la fonction `console.log()` ;
- ⋆ le document en train de s'afficher avec la fonction `document.write()` ;
- ⋆ une fenêtre d'alerte avec la fonction `window.alert()` ;
- ⋆ un élément de la page déjà affiché par son attribut `innerHTML`.

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <title>Tests de sorties</title>
  </head>
  <body>
    <p>Paragraphe 1</p>
    <p id="fin">Paragraphe 2</p>
    <script src="script.js"></script>
  </body>
</html>
```

```javascript
/* Fichier script.js*/
document.write("<strong>FIN</strong>");
var e = document.getElementById("FIN")
e.innerHTML = "Dernier paragraphe.";
console.log("Modification : OK.")
window.alert("Paragraphe 2 modifié !");
```



Les instructions JavaScript, comme le contenu HTML, est interprété dans l'ordre des lignes.

Une fois un élément de la page HTML portant un identifiant récupéré par la fonction `document.getElementById()`, on peut :

- ⋆ modifier ses attributs : `e.setAttribute("attribut", "valeur");`,
- ⋆ supprimer ses attributs : `e.removeAttribute("attribut");`,
- ⋆ agir sur sa classe (donc son style) : `e.className = 'classe';`.

Et comme c'est un langage de programmation, on a les structures habituelles :

Variables

```javascript
var x, y;
x = 7;
y = 7 ** 2;
```

Boucles bornées

```javascript
for (var i=0; i<9; i++) {
  console.log(i);
}
```

Conditionnelles

```javascript
if (x < y) {
  console.log("ok");
} else {
  console.log("!!");
}
```

Appels de fonction

```javascript
function double(x) {
  return 2 * x;
}
console.log(double(7));
```

Boucles non bornées

```javascript
var n = 0;
while (n < 3) {
  n++; console.log(n);
}
```

(Chaque instruction étant terminée par un point-virgule.)

4 - Réaction aux événements

Événement

Un événement est un changement d'état de l'environnement. Il peut être créé par l'utilisateur (avec le clavier, la souris...), par le document lui-même (fin de chargement d'une image, ou de la page...) ou encore par l'exécution d'un code JavaScript.

Gestionnaire d'événements

Un événement une fois créé va se propager aux différents niveaux de définition de la page (document, html, body, jusqu'à l'élément cible et retour). Le rôle d'un gestionnaire d'événement est de l'intercepter et de réagir (souvent par un appel à une fonction JavaScript).

Un gestionnaire d'événement peut être directement placé dans une balise HTML :

```
<!-- Dans l'en-tête -->
<script src="script.js"></script>
<link rel="stylesheet" type="text/css" href="style.css"/>
<!-- Dans le corps -->
<spam onmouseover="rouge(event)" onmouseout="noir(event)">Survolez-moi</spam>
```

avec dans `style.css` :

```
.noir {
  color : black;
}
.rouge {
  color : red;
}
```

et dans `script.js` :

```
function rouge(e) {
  e.target.className = 'rouge'
}
function noir(e) {
  e.target.className = 'noir'
}
```

Dans cet exemple, passer sur les mots `Survolez-moi` ou les quitter avec la souris génère un événement. Il est capté par `onmouseover` ou `onmouseout` qui le transmettent aux fonctions JavaScript qui agissent sur le style graphique de ces mots. On peut citer de plus :

- ⋆ `onchange` : un élément a changé.
- ⋆ `onclick` : l'utilisateur a cliqué sur un élément.
- ⋆ `onkeydown` : l'utilisateur a appuyé sur une touche.
- ⋆ `onload` : la page a fini d'être chargée.

L'élément cliquable par excellence est le bouton :

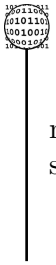
```
<button type="button" onclick="alert(event.target.innerHTML)">GO !</button>
```

IHM sur le Web(II)

Client, serveur et protocole HTTP

1 - Modèle client/serveur

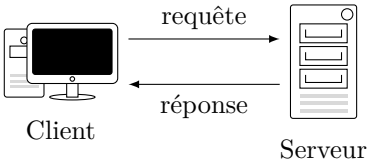
Pour présenter ce modèle, on se place dans un réseau et sous le protocole de communication TCP/IP (voir partie Architectures matérielles). On ne considère que deux ordinateurs en train de communiquer.



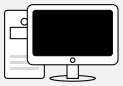
Modèle client/serveur

C'est un modèle de communication dissymétrique entre deux ordinateurs sur un réseau.

- ⋆ Le client envoie une requête au serveur.
- ⋆ Le serveur répond au client.



Voici une proposition minimaliste d'un client et d'un serveur en Python.


Client

```
1  import socket
2
3  hote = "localhost"
4  port = 12800
5
6  connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  connexion.connect((hote, port))
8  print("Connexion établie avec le serveur sur le port",port)
9
10 msg_a_envoyer = input("> ").encode()
11 connexion.send(msg_a_envoyer)
12 msg_recu = connexion.recv(1024)
13 print(msg_recu.decode())
14
15 connexion.close()
```

On utilise un socket pour spécifier, selon le protocole TCP/IP, l'adresse du serveur et son port de communication. Ici, le serveur test est sur le même ordinateur que le client (`"localhost"`) et utilise le port 12800.

On peut observer les trois actions principales d'un client.

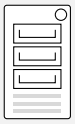
1. La connexion avec le serveur (lignes 7)
2. L'échange de messages (lignes 11 et 12)
3. La déconnexion du serveur (ligne 15)

Les messages échangés sont du domaine des applications (plus haute couche du modèle). Cela concerne par exemple les pages html (protocoles http ou https), l'échange de fichiers (protocole ftp) ou de mails (protocole smtp)...

La requête (message envoyé au serveur) peut contenir des informations que le serveur traitera pour envoyer une réponse adaptée et personnelle à l'utilisateur (on parle de site dynamique, à opposer aux sites statiques qui ont un contenu indépendant de l'utilisateur). Ce traitement se fait à l'aide de programmation (avec différents langages : php ou python ou ...).

La réponse peut également contenir des éléments dynamiques pour l'utilisateur (en JavaScript par exemple).

```python
import socket

hote = ''
port = 12800

serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind((hote, port))
serveur.listen(5)
print("Le serveur écoute sur le port",port)

client, infos = serveur.accept()

msg_recu = client.recv(1024)
print(msg_recu.decode())
client.send(b"5 / 5")

client.close()
serveur.close()
```



Serveur

On peut observer les quatre actions principales d'un serveur.

1. L'attente d'une connexion avec le client (lignes 8)
2. La connexion avec un client (ligne 11)
3. L'échange de messages (lignes 13 et 15)
4. La déconnexion du client (ligne 17)

Les réponses ne sont en général pas générées directement par le serveur. Il peut lui-même faire des requêtes à une base de données, récupérer des modèles et/ou demander à un autre programme de générer la vue à transmettre au client (le Modèle-Vue-Contrôleur : MVC).

2 - Protocole HTTP

HTTP (HyperText Transfert Protocol)
C'est un protocole de la couche application (la plus haute sur modèle TCP/IP) permettant la demande et l'envoi de ressources. Il a été conçu pour la communication entre un navigateur web (client) et un serveur web.

Pour les premiers tests, on utilisera le serveur HTTP à l'adresse http://httpbin.org/ qui renvoie en écho les données utilisées dans la requête.

a) Requête HTTP

Une requête HTTP est de la forme :

```
VERBE URI HTTP/X.X
En-têtes
Corps
```

⋆ Le `VERBE` est le plus souvent `GET` pour *obtenir* une ressource, ou `POST` pour *envoyer*, par exemple, le contenu d'un formulaire.

- ⋆ L'`URI` (Uniform Ressource Identifier) est l'URL (Uniform Ressource Locator) complété pour préciser la ressource visée à cette adresse.
- ⋆ `HTTP/X.X` indique la version du protocole : `HTTP/1.1`
- ⋆ `En-tête` (optionnel) permet de préciser la ressource et le comportement du client/serveur. Chaque ligne est composée d'un nom, de deux points `:` et de valeurs.
  Par exemple : `Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3`
- ⋆ `Corps` contient des données précisant la demande de ressource. Cette partie est vide pour la méthode GET (qui peut tout de même préciser sa demande via l'URI).

En Python, on utilisera la bibliothèque `requests` (à installer).
Code minimal pour une requête GET, avec précisions au bout de l'URL :

```
import requests

ressource = requests.get('http://httpbin.org/get?test=True&valeur=2')
print(ressource.text)
```

Affiche :

```
{
  "args": {
    "test": "True",
    "valeur": "2"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.22.0"
  },
  "origin": "xxx.xxx.xxx.xxx, xxx.xxx.xxx.xxx",
  "url": "https://httpbin.org/get?test=True&valeur=2"
}
```

On peut observer quelques éléments d'en-tête et que les précisions en bout d'URL ont bien été captées.

Code minimal pour une requête POST, avec envoi de données encodées comme si elles venaient d'un formulaire HTML :

```
import requests

donnees = {'clef1': 'valeur1', 'clef2': 'valeur2'}
ressource = requests.post('http://httpbin.org/post', data = donnees)
print(ressource.text)
```

Les données ont bien été transmises comme venant d'un formulaire :

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "test": "True",
    "valeur": "2"
```

```
  },
  ...
```

Si ces données ne sont pas trop importantes, elles peuvent être envoyées par la méthode GET mais seront visibles dans l'URI. Celles transmises par la méthode POST ne le sont pas, mais restent transmises en clair, dans le corps de la requête. Si on souhaite une transmission sécurisée, il faudra en plus utiliser un système de chiffrement.

Pour aller plus loin : D'autres façons de transmettre les données sont possibles.
Par exemple en JSON :

```
import requests
import json

donnees = {'test': True, 'valeur': 2}
donneesJSON = json.dumps(donnees)
ressource = requests.post('http://httpbin.org/post', data = donneesJSON)
```

Ou sinon directement en chaîne de caractères :

```
import requests

donnees = "Ma structure"
ressource = requests.post('http://httpbin.org/post', data = donnees)
```

b) Réponse HTTP

Une réponse HTTP a la même forme qu'une requête sauf pour la première ligne :

```
HTTP/X.X CODE MESSAGE
En-têtes
Corps
```

- ⋆ `CODE` et `MESSAGE` indiquent le statut de la communication.
  Exemples : `200 OK` ou le fameux `404 Not Found`. Voir .https ://httpstatuses.com/
  On peut obtenir le CODE d'une réponse par : `ressource.status_code`.
- ⋆ On peut obtenir l'`en-tête` par : `ressource.headers` (donne un dictionnaire).
- ⋆ On peut obtenir le `corps` au format texte par : `ressource.text`.
  En précisant *avant* l'encodage si besoin : `ressource.encoding = 'utf-8'`.
- ⋆ On peut obtenir le corps au format JSON par : `ressource.json`.

IHM sur le Web(III)

Formulaires

Les formulaires permettent facilement de faire des requêtes directement depuis une page HTML. Pour y répondre, un autre langage (Python, PHP, JavaScript, ...) sera nécessaire.

1 - Construction et envoi d'une requête

Un formulaire HTML est délimité par la balise `<form>` en précisant deux attributs :

- `method` égal à `"get"` ou `"post"` selon le type de requête.
- `action` égal à l'URL de la page ou du programme qui devra traiter cette requête.

Le bouton d'envoi peut s'écrire : `<input type="submit" value="Envoyer" />`
Un formulaire minimal provoquant une requête GET pourrait être :

```
<!doctype html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <title>Le formulaire GET</title>
    </head>
    <body>
        <form action="http://httpbin.org/get" method="get">
            <input type="submit" value="GET !" />
        </form>
    </body>
</html>
```

`<a href="http://httpbin.org/get">GET !</a>` aurait tout aussi bien fait l'affaire !
Un formulaire n'a effectivement de sens que si l'utilisateur peut sélectionner ses entrées.
C'est encore la balise `<input />` qui joue ce rôle.

`<input type="text" name="nom" />` par exemple permet à l'utilisateur d'entrer une chaine de caractères, qui sera envoyé sous le nom `"nom"`.

Pour une interface graphique claire, on peut lier un tel champ d'entrée (s'il est identifié par un `id`) avec une description en utilisant la balise `<label>` (en donnant à son attribut `for` l'identifiant en question) :

```
        <form action="http://httpbin.org/get" method="get">
            <label for="nom">Nom</label> :
            <input type="text" name="nom" id="nom" />
            <input type="submit" value="GET !" />
        </form>
```

L'équivalent avec la méthode POST :

```
        <form action="http://httpbin.org/post" method="post">
            <label for="nom">Nom</label> :
            <input type="text" name="nom" id="nom" />
            <input type="submit" value="POST !" />
        </form>
```

On peut préciser la taille du champ avec l'attribut `size`, le nombre maximum de caractères à saisir avec `maxlength`, une valeur prédéfinie par `value` ou une simple indication de contenu par

`placeholder`. On peut rendre un champ obligatoire avec l'attribut `required`, non modifiable avec `readonly`...

Valeurs possibles de l'attribut `type` :

- ⋆ `<input type="password" name="psw" />` (La valeur n'apparait pas.)
- ⋆ `<input type="reset" />` (Le formulaire est réinitialisé.)
- ⋆ `<input type="radio" name="genre" value="homme" checked /> Homme<br />`.
  `<input type="radio" name="genre" value="femme" /> Femme<br />`.
  (Un seul peut être sélectionné à la fois.)
- ⋆ `<input type="checkbox" name="finrepas" value="fromage" /> Fromage<br />`.
  `<input type="checkbox" name="finrepas" value="dessert" checked /> Dessert<br />`.
  (Sélection libre.)
- ⋆ `<input type="button" onclick="alert('Bonjour !')" value="Bonjour !">`
- ⋆ et des types « text enrichi » (ramenés à text simple si non pris en charge par le navigateur) : `"color"`, `"date"`, `"datetime-local"`, `"email"`, `"month"`, `"number"`, `"range"`, `"search"`, `"tel"`, `"time"`, `"url"` ou `"week"`

2 - Réponse ?

Pour répondre, on sous-entend être du côté serveur. Déjà, il faut arriver à capter la demande. Le langage dédié sur le web est le PHP :

```php
<?php
$nom = htmlspecialchars($_GET["nom"]);
?>
```

```php
<?php
$nom = htmlspecialchars($_POST["nom"]);
?>
```

Mais ce langage n'est pas un objectif de première.
On peut proposer un serveur minimal en Python (navigateur en `Localhost:8765/valeur`) :

```python
from http.server import BaseHTTPRequestHandler, HTTPServer

class Gestionnaire(BaseHTTPRequestHandler):

    def do_GET(self):
        html = """<!doctype html><html>
<head><meta charset="utf-8"><title>Serveur minimal GET</title></head>
<body><p>""" + str(self.path) + "</p></body></html>"
        self.send_response(200)
        self.send_header("Content-Type", "text/html; charset=utf-8")
        self.end_headers()
        self.wfile.write(html.encode("utf-8"))

if __name__ == "__main__":
    print("Lancement du serveur minimal")
    httpd = HTTPServer(('', 8765), Gestionnaire)
    httpd.serve_forever()
    httpd.server_close()
```

mais là encore, on dépasse les attendus de première !

On ne s'étendra donc pas sur le sujet.

Architectures matérielles et systèmes d'exploitation

- Programme officiel

Exprimer un algorithme dans un langage de programmation a pour but de le rendre exécutable par une machine dans un contexte donné. La découverte de l'architecture des machines et de leur système d'exploitation constitue une étape importante.

Les circuits électroniques sont au cœur de toutes les machines informatiques. Les réseaux permettent de transmettre l'information entre machines. Les systèmes d'exploitation gèrent et optimisent l'ensemble des fonctions de la machine, de l'exécution des programmes aux entrées-sorties et à la gestion d'énergie.

On étudie aussi le rôle des capteurs et actionneurs dans les entrées-sorties clavier, interfaces graphiques et tactiles, dispositifs de mesure physique, commandes de machines, etc.

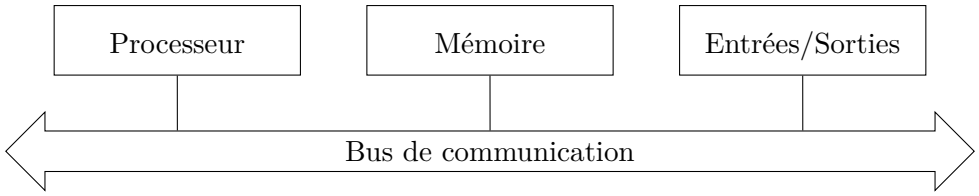
Contenus	Capacités attendues	Commentaires
Modèle d'architecture séquentielle (von Neumann)	Distinguer les rôles et les caractéristiques des différents constituants d'une machine. Dérouler l'exécution d'une séquence d'instructions simples du type langage machine.	La présentation se limite aux concepts généraux. On distingue les architectures monoprocesseur et les architectures multiprocesseur. Des activités débranchées sont proposées. Les circuits combinatoires réalisent des fonctions booléennes.
Transmission de données dans un réseau Protocoles de communication Architecture d'un réseau	Mettre en évidence l'intérêt du découpage des données en paquets et de leur encapsulation. Dérouler le fonctionnement d'un protocole simple de récupération de perte de paquets (bit alterné). Simuler ou mettre en œuvre un réseau.	Le protocole peut être expliqué et simulé en mode débranché. Le lien est fait avec ce qui a été vu en classe de seconde sur le protocole TCP/IP. Le rôle des différents constituants du réseau local de l'établissement est présenté.
Systèmes d'exploitation	Identifier les fonctions d'un système d'exploitation. Utiliser les commandes de base en ligne de commande. Gérer les droits et permissions d'accès aux fichiers.	Les différences entre systèmes d'exploitation libres et propriétaires sont évoquées. Les élèves utilisent un système d'exploitation libre. Il ne s'agit pas d'une étude théorique des systèmes d'exploitation.
Périphériques d'entrée et de sortie Interface Homme-Machine (IHM)	Identifier le rôle des capteurs et actionneurs. Réaliser par programmation une IHM répondant à un cahier des charges donné.	Les activités peuvent être développées sur des objets connectés, des systèmes embarqués ou robots.

Architectures matérielles et SE(I)

Modèle d'architecture

1 - Qu'est-ce qu'un ordinateur ?

1945 Le mathématicien John von Neumann élabore la première description d'un ordinateur. On peut la simplifier ainsi :



Ordinateur

Un ordinateur est une machine *électronique* capable d'exécuter des *instructions* effectuant des *opérations* sur des *nombres*.

a) Entrées/sorties

Entrées/sorties

- ⋆ Un périphérique d'entrée permet de fournir des informations au système.
  Exemples : Clavier, souris, scanner, micro, etc.
- ⋆ Un périphérique de sortie permet de faire sortir des informations du système.
  Exemples : Écran, imprimante, haut-parleurs, etc.
- ⋆ Un périphérique d'entrée/sortie permet l'échange d'informations dans les deux sens.
  Exemples : Clé USB, Carte son, carte réseau, etc.

b) Mémoires

Mémoire vie / Mémoire morte

- ⋆ La mémoire vive ou temporaire est celle des barrettes de RAM (Random Access Memory), qui ne persiste pas hors tension.
- ⋆ La mémoire morte ou persistante est celle du disque dur (Hard Drive), qui permet une mémorisation permanente.

La mémoire vive est indispensable pour sa vitesse d'accès ($\approx$ 10ns contre $\approx$ 5ms pour la mémoire morte) et la mémoire morte est indispensable pour sa persistance.

De nos jours, les processeurs eux-mêmes sont dotés de leur propre petite mémoire vive, dite mémoire cache, pour augmenter les vitesses d'accès ($\approx$ 5ns). Ils fonctionnent par ailleurs en manipulant de touts petits espaces mémoires appelés registres, encore plus rapides ($\approx$ 1ns).

c) Processeur

Processeur
Le processeur est le composant électronique qui permet d'effectuer les calculs.

L'élément constitutif de base est le *transistor*, sorte d'interrupteur, contrôlé, qui permet, ou non, de laisser passer le courant, donc aboutissant à deux états possibles : 0 ou 1. Le processeur ne travaille fondamentalement qu'avec *deux chiffres*, donc avec des *nombres binaires*.


tube


transistor


circuit itégré

1943 Le premier ordinateur, le Colossus était conçu à partir de tubes électroniques.

De nos jours, les transistors des processeurs ne font que quelques nanomètres et se comptent en milliards. Combiner plusieurs transistors permet la création de *circuits logiques*. Il en existe de deux types :

⋆ les circuits combinatoires, si les sorties ne sont fonction que des entrées.

⋆ les circuits séquentiels, si les sorties dépendent également du temps et des états antérieurs.

2 - Circuits combinatoires

a) Porte non

Le circuit combinatoire le plus simple celui qui permet de changer d'état :

⋆ Si l'entrée est $E = 1$ alors la sortie est $S = 0$

⋆ Si l'entrée est $E = 0$ alors la sortie est $S = 1$

On identifie 0 à « faux » et 1 à « vrai » et ce circuit correspond alors à une négation : ce qui était vrai devient faux et ce qui était faux devient vrai. On appelle ce circuit une « porte non » :

Porte non

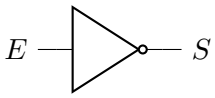
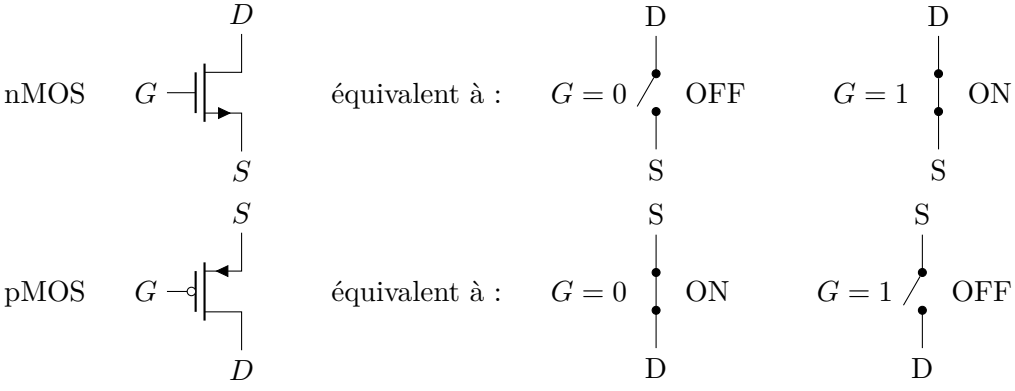
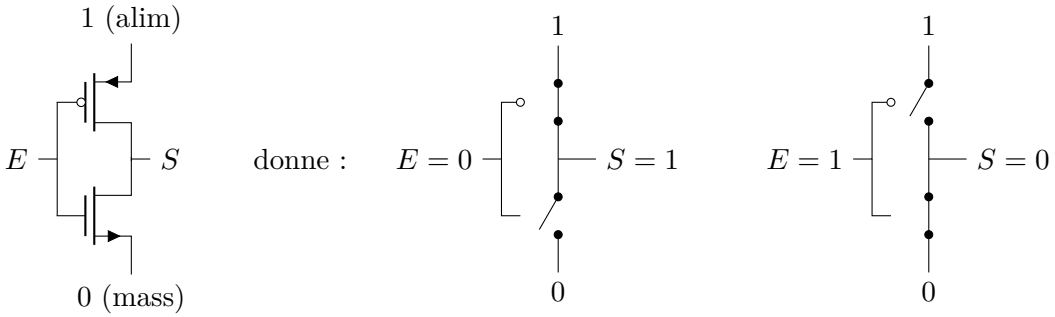


Table de vérité :

E	S
0	1
1	0

Pour aller plus loin : ce circuit n'est composé que de deux transistors :

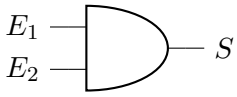




b) Portes et / ou / ou exclusif

Les portes logiques suivantes admettent deux entrées et une sortie.

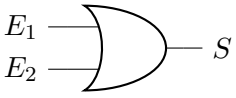
Porte et




```
S = E1 & E2
```

E_1	E_2	S
0	0	0
0	1	0
1	0	0
1	1	1

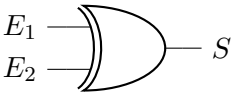
Porte ou




```
S = E1 | E2
```

E_1	E_2	S
0	0	0
0	1	1
1	0	1
1	1	1

Porte ou exclusif




```
S = E1 ^ E2
```

E_1	E_2	S
0	0	0
0	1	1
1	0	1
1	1	0

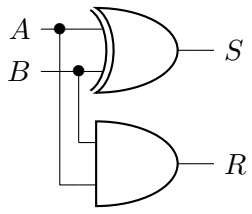
Sur ce modèle, on pourrait dresser $2^4 = 16$ tables de vérité différentes (même si certaines n'ont pas d'intérêt : S toujours à 0 ou toujours à 1...). Pour arriver à faire des additions (binaires), ces trois portes suffisent.

c) Demi additionneur (half adder)

Posons les additions à un bit :

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

Il n'y a une retenue que si les deux bit sont à 1 ($\Rightarrow$ porte et)
Mise à part la retenue, le résultat est 1 seulement si un seul bit est 1 ($\Rightarrow$ porte ou exclusif)



La somme :

```
S = A ^ B
```

La retenue :

```
R = A & B
```

A	B	R	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

d) Plein additionneur (full adder)

Pour additionner deux entiers écrits en binaire, il est nécessaire de tenir compte des retenues à chaque addition de deux bits :

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

Notons R_0 la retenue dont il faut tenir compte.

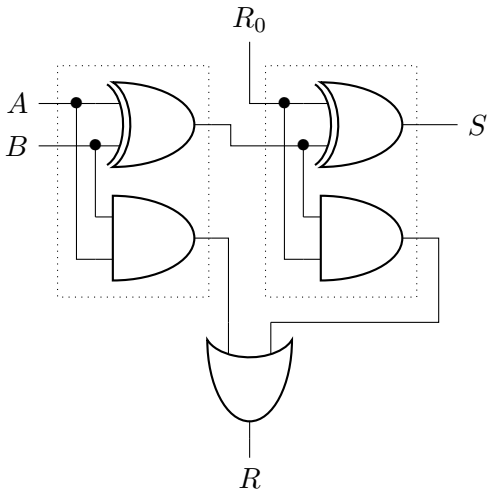
On peut compléter la table de vérité (ci-dessous).

Pour calculer S on utilise de nouveau un demi additionneur pour ajouter R_0 ; et la retenue R vaut 1 dès que l'un des deux demi additionneurs renvoie une retenue à 1.

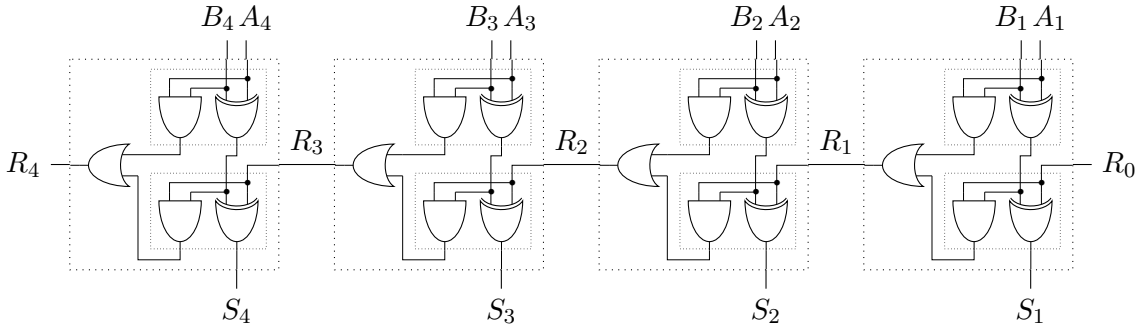
R_0	A	B	R	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

```
S = A ^ B ^ R0
```

```
R = (A & B) | (R0 & (A ^ B))
```



Enfin, pour additionner des nombres binaires, il suffit de mettre plusieurs plein additionneurs en cascade :



3 - Processeur et langage machine

On peut distinguer trois constituants pour un processeur.

⋆ L'*unité arithmétique et logique* (UAL ou ALU en anglais).
Elle est chargée des calculs.
On y retrouve par exemple des circuits du type de l'additionneur.

⋆ L'*unité de commande*.
Elle est chargée de lire et décoder les instructions en mémoire.

⋆ Les *registres*.
Ils sont chargés de mémoriser de l'information (instructions ou données).

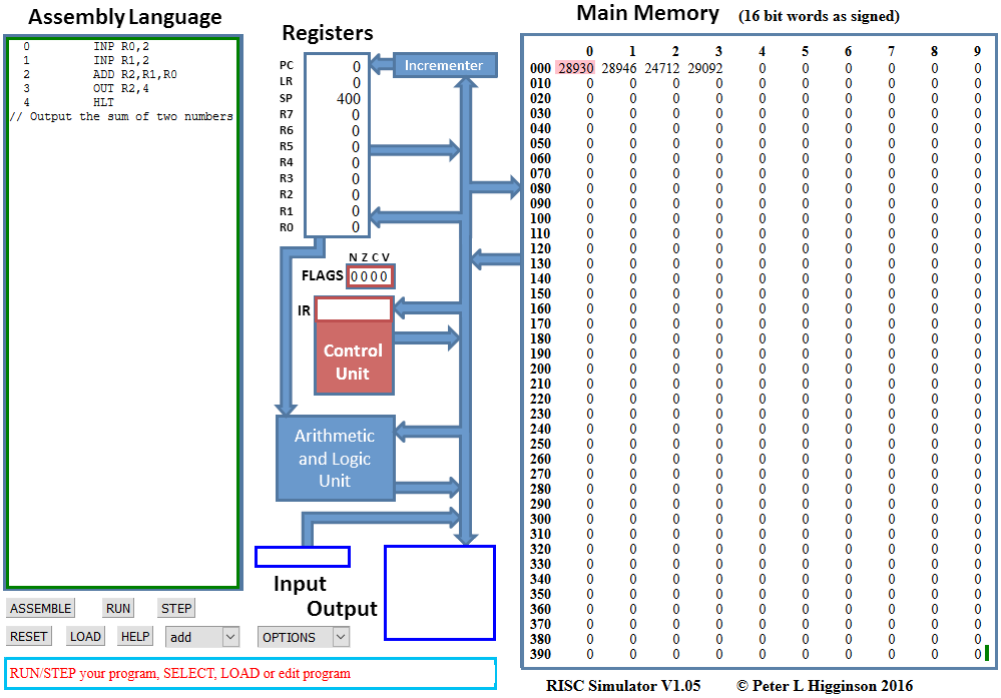
On peut visionner une simulation d'un processeur RISC (Reduced instruction set computer) sur peterhigginson.co.uk/RISC.

Langage machine et assembleur

Le *langage machine* est le langage compris et exécutable par le processeur. C'est une suite de mots binaires écrits en mémoire. Chaque type de processeur « parle » son propre langage machine.

Le *langage assembleur* est le langage de bas niveau permettant une correspondance (1 pour 1) entre une instruction compréhensible par un humain et la même en langage machine.

Sur le simulateur RISC proposé à l'étude, on retrouve les éléments énoncés plus haut.



Trois registres ont des fonctions précises.

⋆ Le registre PC : Compteur Programme (adresse de la prochaine instruction).

⋆ Le registre LR : Adresse de retour (Link Register).

⋆ Le registre SP : Pointeur de pile (Stack Pointer).

Quatre drapeaux sont représentés (N : Négatif, Z : zéro, C : retenue, V : Dépassement).

Algorithme

```
1 R1 ← 10
2 tant que R1 ≠ 0 faire
3 |   R1 ← R1 − 1
4 |   Affiche R1
```

Assembleur

```
1          MOV R1,#10
2 debut:   CMP R1,#0
3          BEQ fin
4          SUB R1,#1
5          OUT R1,4
6          BRA debut
7 fin:     HLT
```

Machine

```
1 0010100100001010
2 0010000100000000
3 1000001000000110
4 0001100100000001
5 0111000110010100
6 1000000000000001
7 0000000000000000
```

Certaines opérations fondamentales sont disponibles dans la plupart des jeux d'instructions.

⋆ Pour le traitement des données. (Calculs arithmétiques, logiques et comparaisons.)

⋆ Pour le transfert des données (des registres depuis ou vers la mémoire).

⋆ Pour réaliser des « branchements ». (Saut non incrémentale d'adresses dans le programme.)

Exemples disponibles avec le simulateur :
(liste complète peterhigginson.co.uk/RISC/instruction_set.pdf)

⋆ Traitement des données.
`ADD R1,#10` ajoute le nombre 10 au registre R1 ($R_1 \leftarrow R_1 + 10$),
`ADD R1,R2,R3` ajoute R3 à R2 et place la somme dans R1 ($R_1 \leftarrow R_1 + R_2$),
`MOV R1, R2` place la valeur de R2 dans R1 ($R_1 \leftarrow R_2$),
`CMP R1, #27` compare la valeur de R1 avec 27 (Résultats dans les drapeaux).

⋆ Transfert des données.
`STR R1,10` enregistre le contenu du registre R1 en mémoire à l'adresse 10.
`LDR R1,10` charge dans le registre R1 le contenu de la mémoire à l'adresse 10.

⋆ Branchements.
`BRA 7` saute à l'instruction en mémoire à l'adresse mémoire 7.
`BRA ici` saute à l'instruction en mémoire à l'adresse du label `ici:`.
À partir de la valeur des drapeaux, saut d'instruction si
`BEQ ici` il y a égalité ; `BNE ici` il n'y a pas égalité
`BGT ici` la valeur est plus grande ; `BLT ici` la valeur est plus plus petite.

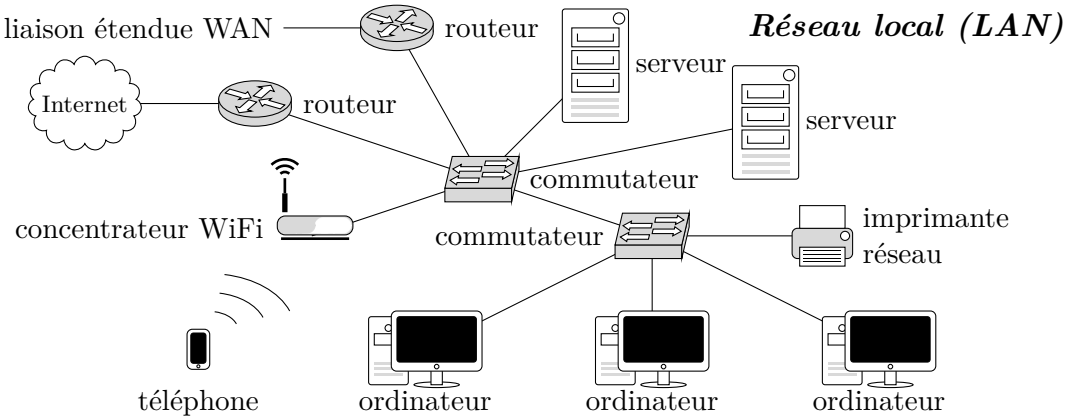
Architectures matérielles et SE(II)

Réseau

1 - Introduction

Réseaux

Un réseau est un ensemble de *matériels*, connectés de manière à pouvoir *échanger* des données numériques et *partager* des ressources.



On distingue différents types de réseaux selon leurs tailles (nombre de matériels connectés), leurs structures (façon de les connecter), leurs étendues géographiques (LAN < WAN)...

1969 Transmission du premier message « login » sur ARPANET.

(ARPANET - Advanced Research Projects Agency Network - est le premier réseau, développé aux États-Unis, entre des universités.)

- ⋆ en 1981 : 213 machines connectés sur Internet.
- ⋆ en 1989 : 100 000 machines connectés - 1er navigateur web graphique (par le CERN). (CERN : Conseil européen pour la recherche nucléaire.)
- ⋆ De nos jours, il y en a des dizaines de milliards.

2 - Réseaux locaux

a) Définition

Réseau local (RL) - Local Area Network (LAN)

C'est un réseau de taille réduite couvrant un domaine privé (entreprises, administrations, universités,...)

02/1980 Constitution du comité IEEE 802, pour standardiser les transmissions d'informations.

(IEEE : Institute of Electrical and Electronics Engineers)

En particulier, pour les réseaux locaux :

⋆ Ethernet - filaire (IEEE 802.3) ;
⋆ 1996 Fast Ethernet (IEEE 802.14) ;
⋆ WLAN - réseau sans fil (IEEE 802.11) ;
⋆ 2002 Bluetooth (802.15) ; ...

b) Matériel

Les réseaux Ethernet sont constitués d'ordinateurs et autres machines possédant une carte réseau, reliés, par des câbles Ethernet, à un ou plusieurs commutateurs (switchs), eux-mêmes reliés entre eux.

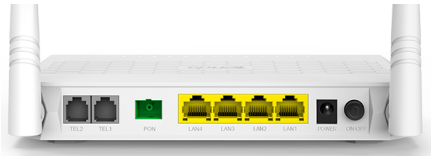

carte réseau


câble Ethernet


commutateur (switch)

Pour aller plus loin :
Pour relier de tels réseaux entre eux, on utilise des routeurs, possédant, eux, plusieurs cartes réseaux. D'autres supports existent alors, comme la fibre optique.


fibre optique


routeur


câble Ethernet

c) Adressage physique (couche Réseau)

Adresse physique (MAC) (MAC : Media Access Control)
C'est un identifiant, unique dans le monde, pour chaque carte réseau.

Elle se présente sous la forme de 6 octets (48 bits), en notation hexadécimale, séparés par un double point ou un tiret. Exemple : `88-AD-43-F7-26-F8`.

Les adresses physiques désignent, dans une *trame*, les protagonistes directs d'un transport de datagramme sur le réseau.

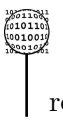
Trame : ⋯ @MAC-source ; @MAC-destination ⋯ datagramme ⋯

C'est chaque bit de cette trame qui est transmis physiquement sur le réseau.

Un *commutateur* (switch)  est capable de décoder cette trame.

⋆ Il met à jour sa liste, appelée *table SAT*, des machines qui lui sont connectées.
⋆ Il ne transmet la trame qu'au destinataire.

d) Adressage réseaux (couche Internet)



Adresse IP (IP : Internet Protocol)
C'est un identifiant (l'adresse), unique dans un réseau (le réseau public : internet, ou un réseau privé), pour chaque carte réseau.

Elle se présente (en version 4) sous la forme de 4 octets (32 bits), en notation décimale, séparés par un point. Exemple : `192.168.1.67`

On peut demander ses propres identifiants en ligne de commande :

ipconfig (Windows)

```
C:\>ipconfig /all

Carte Ethernet Ethernet:

   Adresse physique . . . . . . . . . . . . : 88-AD-43-F7-26-F8
   Adresse IPv4. . . . . . . . . . . . . . .: 192.168.1.67
```

ip (Unix)

```
moi@ici:~$ ip addr show

2: enp0s3:
    link/ether 88:AD:43:F7:26:F8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.67/24 brd 192.168.1.255 scope global dynamic enp0s3
```

Pour tester si un équipement est joignable sur un réseau IP, on peut utiliser :

ping (Windows / Unix)

```
C:\>ping 192.168.1.1

Envoi d'une requête 'Ping'  192.168.1.1 avec 32 octets de données :
Réponse de 192.168.1.1 : octets=32 temps<1ms TTL=64
...

C:\>ping 192.168.1.7

Envoi d'une requête 'Ping'  192.168.1.7 avec 32 octets de données :
Réponse de 192.168.1.67 : Impossible de joindre l'hôte de destination.
```

Elle désigne, dans un *datagramme*, les protagonistes initiateur et final et permettent l'acheminement (le routage) de paquets de données.

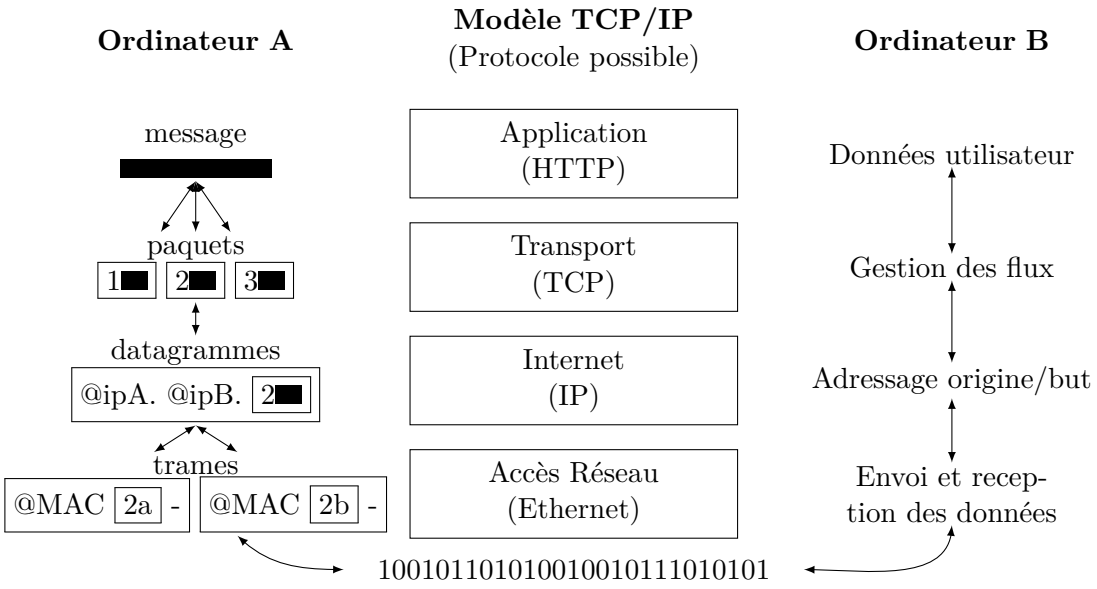
Datagramme : $\boxed{\cdots \text{ @IP-initiale ; @IP-finale } \cdots \boxed{\text{paquet}}}$

C'est l'assemblage de plusieurs paquets qui forme le message envoyé et reçu par les applications.

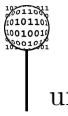
e) Communication entre applications

Le but de tout réseau est de pouvoir faire communiquer des logiciels sur des ordinateurs distants. Pour transporter ses données :

- ⋆ l'application fait appel à un protocole de transport : TCP (Transmission Control Protocol) qui va faire des paquets de données (numérotés) à envoyer.
- ⋆ Chaque paquet reçoit l'adresse IP source et l'adresse IP destination, pour former un datagramme, routable sur internet.
- ⋆ Chaque échange sur les réseaux se fait par trames de données, en identifiant les voisins, de proche en proche, par leur adresse MAC.



Pour aller plus loin :



Numéro de port
C'est l'identifiant d'une application. Associé à l'adresse IP de l'ordinateur, ils définissent un *socket* qui permet une connexion TCP (Transmission Control Protocol).

Exemples de ports pré-définis :

⋆ 25 : SMTP (Simple Mail Transfert Protocol)

⋆ 80 : HTTP (HyperText Transfer Protocol)

⋆ 110 : POP3 (Post Office Protocol, version 3)

f) Masque de réseau

Une adresse IP contient toujours au moins deux informations distinctes :

⋆ l'identifiant du réseau sur lequel est la machine,

⋆ et l'identifiant de la machine elle-même sur ce réseau.

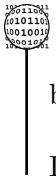
$$\overbrace{\texttt{192.168.1.}}^{\text{Réseau } \texttt{192.168.1.0}}\overbrace{\texttt{67}}^{\text{machine } \texttt{67}}$$

Plus précisément :

$$\overbrace{\texttt{1100 0000 . 1010 1000 . 0000 0001}}^{\text{Réseau } \texttt{192.168.1.0}} . \overbrace{\texttt{0100 0011}}^{\text{machine } \texttt{67}}$$

Mais le nombre de bits attribués au réseau peut être différent, par exemple :

$$\overbrace{\texttt{1100 0000 . 1010 1000 . 0000 0001 . 0100}}^{\text{Réseau } \texttt{192.168.1.64}}\overbrace{\texttt{0011}}^{\text{machine } \texttt{3}}$$



Masque de réseau

Le masque d'un réseau se présente sous la forme d'une adresse IP pour laquelle tous les bits composant le réseau sont égaux à 1 et tous les autres égaux à 0.

Elle permet, avec l'opération binaire `&` de déterminer le réseau correspondant à l'adresse IP d'une machine.

```
   1100 0000 . 1010 1000 . 0000 0001 . 0100 0011
&  1111 1111 . 1111 1111 . 1111 1111 . 1111 0000
-------------------------------------------------
   1100 0000 . 1010 1000 . 0000 0001 . 0100 0000
```

Sur l'exemple ci-dessus, le masque est `255.255.255.240`.

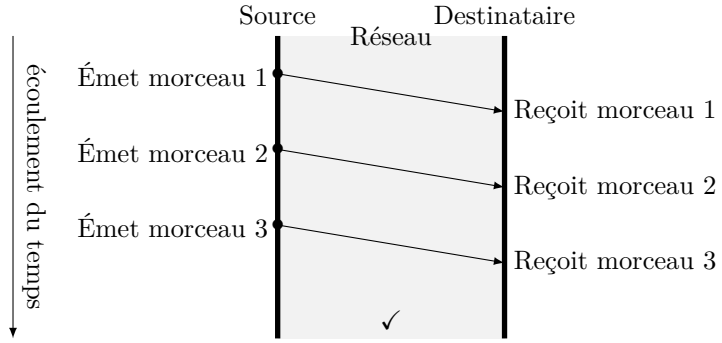
Pour simplifier, on peut juste donner le nombre de bits à 1 pour le masque. Par rapport à l'exemple précédent, on peut distinguer `192.168.1.67/24` et `192.168.1.67/28` qui ne sont pas sur le même réseau.

Cette distinction est importante car seules les machines partageant le même réseau peuvent communiquer directement ou à travers un commutateur.

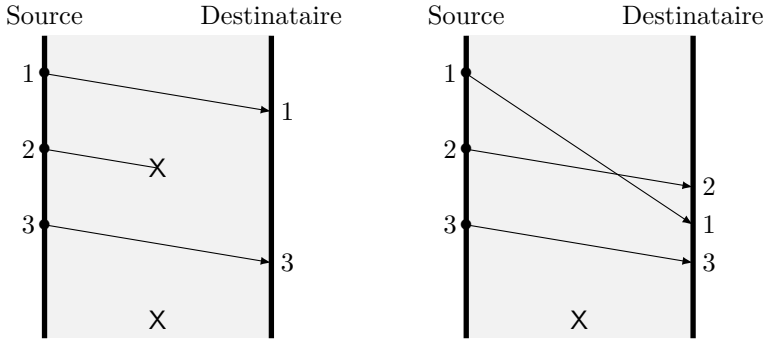
3 - Contrôle de flux

Les messages qui transitent sur les réseaux sont découpés en morceaux (dès la prise en charge par le protocole TCP, qui en fait des paquets). Il peut donc s'avérer important que la source s'assure de la bonne réception de chaque morceau dans le bon ordre pour valider l'envoi (et sinon renvoyer l'information).

On représente la situation par deux axes verticaux représentant l'écoulement du temps au niveau de la source et du destinataire des messages, séparé par un espace représentant le réseau de communication. On symbolise le routage des données par des flèches allant d'une verticale à l'autre. Idéalement :



On souhaite éviter la perte d'un morceau ou l'arrivée des morceaux dans le désordre :



a) Envoyer et attendre (Send and Wait)

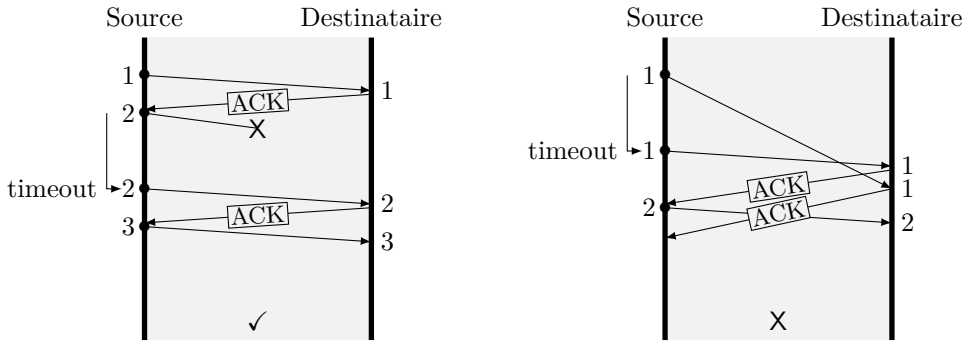
Le principe est le suivant.

⋆ La source n'envoie qu'un morceau à la fois et attend un *aquittement* (ACK) du destinataire.

⋆ Le destinataire n'envoie un ACK que s'il reçoit un morceau en bon état.

⋆ Après un délai pré-établi (timeout) sans ACK, la source renvoie le même morceau.

⋆ Après réception d'un ACK, la source envoie le morceau suivant.



Ça ne solutionne qu'une partie du problème.

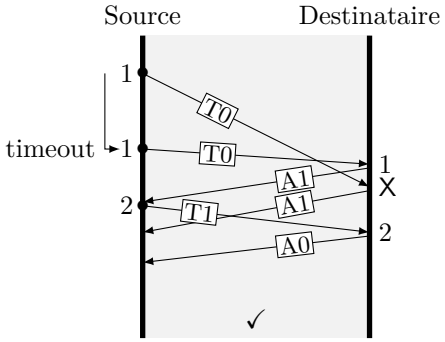
b) Le bit alterné

Pour mettre de l'ordre dans les morceaux reçus, on ne peut pas vraiment se passer de les numéroter. L'idée du bit alterné, est de ne les numéroter que sur un bit (par économie) que l'on fait alterner entre 0 et 1, en envoi et dans les acquittements.

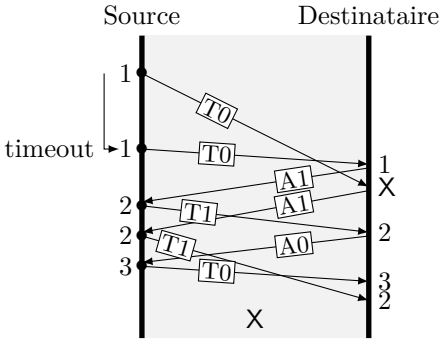
Notons `T0` (ou `T1`) une trame envoyée avec le bit à 0 (ou 1).

L'acquittement demandera la trame suivante en renvoyant le bit opposé : `A1` (ou `A0`) l'acquittement correspondant :

On résout le problème précédent :



Mais pas tous les problèmes :



Pour aller plus loin : Au niveau du protocole TCP, chaque segment porte un numéro de séquence et un numéro d'acquittement, chacun sur 32 bit pour assurer la bonne réception de l'ensemble dans le bon ordre.

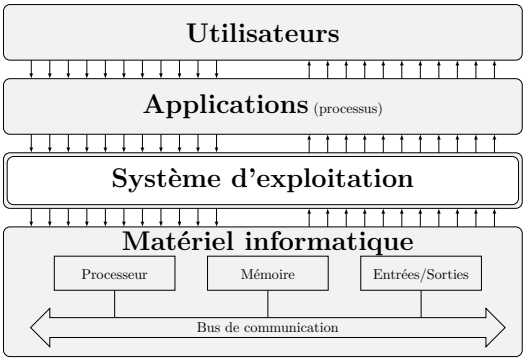
Par comparaison, le protocole UDP (sans connexion) n'en utilise aucun, pour gagner en rapidité. (Dans un flux vidéo par exemple perdre une partie d'une image est moins grave que de faire une pause pour l'attendre.)

Architectures matérielles et SE(III)

Système d'exploitation

1 - Fonctions d'un système d'exploitation

Système d'exploitation
Un système d'exploitation est l'interface logicielle entre les applications et le matériel informatique.



(Les utilisateurs utilisent les applications, qui demandent des ressources au système d'exploitation, qui pilote le matériel informatique, le partageant entre différentes applications et différents utilisateurs.)

Il existe différentes familles de systèmes d'exploitations.

1970 Naissance d'UNIX, père des systèmes d'exploitation modernes (macOS, iOS , Android, Linux sont de la famille UNIX). UNIX étant un système propriétaire (les sources appartiennent à une entreprise et ne sont pas publiques), c'est en clonant ce système (sans utiliser ses sources) que Linux a été écrit (1991).

1985 Naissance de Windows (sur MS-DOS). Mais il en existe bien d'autres...

On peut distinguer quatre grands types de fonctions (trois directement liées au matériel).

- ⋆ La gestion des processus.
- ⋆ La gestion de la mémoire.
- ⋆ La gestion des périphériques d'entrée/sortie.
- ⋆ La gestion du système de fichiers.

2 - IHM et Ligne de commande

Interface Homme-Machine
Une Interface Homme-Machine - IHM est un ensemble de mécanismes, à la fois matériels et logiciels, mis à la disposition des utilisateurs pour leur permettre d'interagir avec un système interactif.

Cette interface est mise en œuvre sur un *terminal*, ensemble de périphériques d'entrée / sortie, à l'extrémité d'un ordinateur ou d'un réseau. Souvent intégré (clavier, écran tactile, ...) mais pas toujours (Terminal bancaire), il peut aussi être virtuel (console d'un système d'exploitation).




Sous Ubuntu, c'est l'application `Terminal` ; sous Windows, `cmd`.

```
moi@mongroupe:~$
```

L'invite de commande Ubuntu se compose du nom d'utilisateur, de `@` et de son groupe, suivi de `:` , du répertoire courant (`~` pour le répertoire utilisateur) et termine par `$` .

```
C:\Users\moi>
```

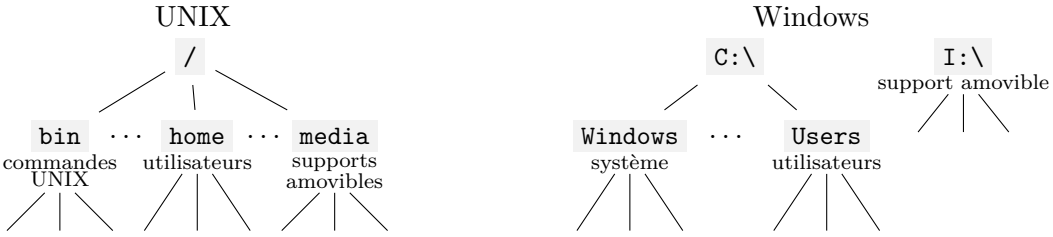
L'invite de commande « MS-DOS » de Windows se compose du nom du lecteur, suivi de `:`, du répertoire courant et termine par `>`.

Une commande est alors composée d'un nom suivi ou non d'options. Les commandes spécifiques au système (que ce soit Ubuntu ou Windows) sont listées par la commande `help`

Pour effacer la console par exemple : `clear` (UNIX) ou `cls` (Windows).

a) Navigation dans l'arborescence des fichiers

Les données en mémoire sont répertoriées par des fichiers, caractérisés par leur nom, taille, droit d'accès, propriétaire, date de création, de modification, ... Ces fichiers sont eux-mêmes répertoriés dans une structure arborescente dont chaque nœud est un dossier. Dans un système UNIX, tout part d'une même racine (root) : `/` . Dans un système Windows, il y a une arborescence par disque logique, identifiée par une lettre.



UNIX	Windows	Action
`pwd`	`cd`	Affiche le répertoire courant
`cd`	`cd %HOMEPATH%`	Positionne au répertoire utilisateur
`cd ..`	`cd ..`	Positionne au répertoire parent
`cd /`	`cd \`	Positionne à la racine

De manière générale : `cd` suivi d'un chemin positionne au bout du chemin.
Le chemin peut être *relatif* (au répertoire courant) ou *absolu* (commençant par `/` ou `C:\`).
`..` est le répertoire précédent, `.` le répertoire courant.
La touche de tabulation permet une complétion automatique des noms de fichiers.

UNIX	Windows	Action
`ls`	`dir`	Liste les fichiers du répertoire courant

En faisant suivre le nom d'une commande par `-h` ou `--help` (UNIX) ou `/?` (Windows), on obtient des détails sur leurs utilisations et options. Par exemple, pour obtenir une liste récursive des fichiers du répertoire courant et de ses sous-répertoires : `ls -R` ou `dir /S`.

Autres actions possibles sur les dossiers :

UNIX	Windows	Action
`mkdir monRep`	`md monRep`	Crée le répertoire `monRep`
`mv monRep rep`	`ren monRep rep`	Renomme le répertoire `monRep` en `rep`
`rmdir monRep`	`rd monRep`	Supprime le répertoire vide `monRep`

b) Visualisation et manipulation des fichiers

Quelques actions possibles sur les fichiers :

UNIX	Windows	Action
`touch fichier.txt`	Pas d'équivalent⋆	Crée
`cat fichier.txt`	`type fichier.txt`	Affiche le contenu
`mv fichier.txt test.txt`	`ren fichier.txt test.txt`	Renomme
`mv test.txt monRep`	`move test.txt monRep`	Déplace
`cp test.txt monRep`	`copy test.txt monRep`	Copie
`rm test.txt`	`del test.txt`	Supprime

(⋆ Dans le cas où le fichier n'existe pas déjà on peut écrire `type Nul > fichier.txt`)

On peut lancer des applications depuis la ligne de commande.
Exemples : `gedit fichier.txt` (UNIX) ou `notepad fichier.txt` (Window).

3 - Droits et permissions d'accès aux fichiers

Sous Windows, il n'existe pas vraiment d'équivalent aux lignes de commandes proposées ci-dessous. On ne considère donc ici que les fichiers d'un système de type UNIX. On les distingue par :

⋆ 4 types de fichier
- `-` Ordinaire
- `d` Répertoire (directory)
- `l` Lien symbolique (link)
- `c` ou `b` Spécial

⋆ 3 catégories d'utilisateurs
- `u` Propriétaire (user)
- `g` Groupe (group)
- `o` Autres (others)

⋆ 3 types de droit
- `r` Lecture (read)
- `w` Écriture (write)
- `x` Exécution (execute)

On peut visualiser ces informations par un listing détaillé : `ls -l`.

Les trois types de droit pouvant ou non être accordés indépendamment, ils génèrent $2^3 = 8$ possibilités, que l'on numérote comme s'ils écrivaient un nombre binaire :

`---`	`--x`	`-w-`	`-wx`	`r--`	`r-x`	`rw-`	`rwx`
0	1	2	3	4	5	6	7

En juxtaposant les droits des 3 catégories d'utilisateurs, on obtient alors un nombre écrit en octal.

Exemple : Un répertoire `d`, pour lequel le propriétaire a tous les droits `rwx`, que le groupe n'a juste pas le droit de modifier `r-x` et qui ne laisse le droit qu'en lecture aux autres `r--` est de type `drwxr-xr--`. Ces droits se lisent en octal : 754.

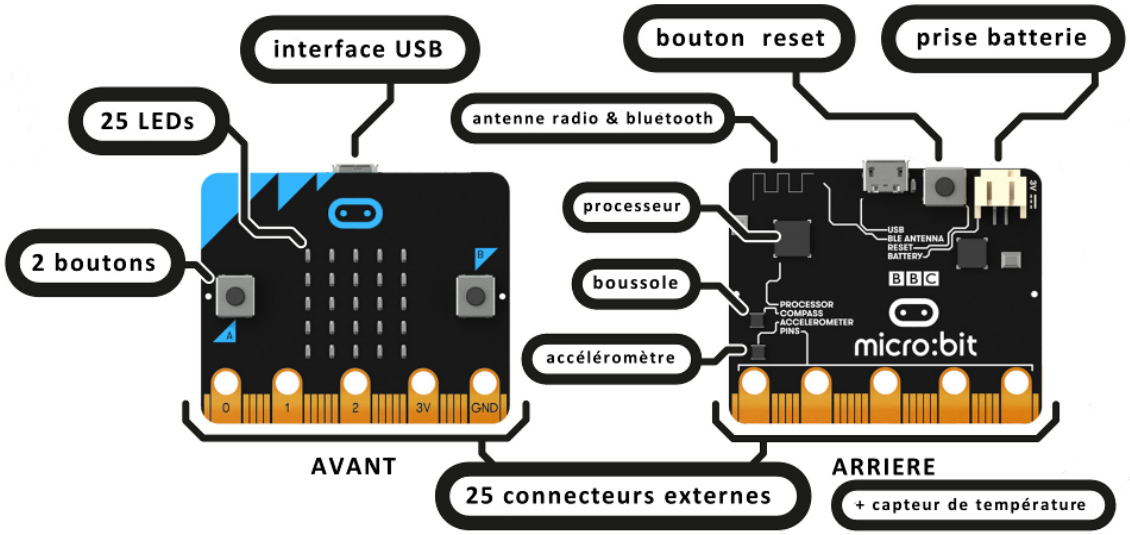
On peut changer les droits en donnant sa valeur en octal :
`chmod 777 fichier.txt` donne tous les droits à tout le monde sur le fichier.

On peut modifier les droits relativement à certains utilisateurs :
`chmod o - w fichier.txt` enlève le droit d'écriture aux autres utilisateurs.
(o peut être g ou u, - peut être + et w peut être r ou x)

Architectures matérielles et SE(IV)

Exemple concret d'un nano-ordinateur

On prend ici l'exemple du nano-ordinateur micro:bit.



On utilise l'IDE Mu qui permet facilement de flasher la carte, et la bibliothèque dédiée :

```
1 from microbit import *
```

On pourra trouver une introduction et des précisions sur le langage MicroPython utilisé : https://microbit-micropython.readthedocs.io/fr/latest/tutorials/introduction.html

1 - Capteurs et actionneurs

Capteur

Un capteur est un périphérique d'entrée. Il permet l'acquisition d'une grandeur physique (température, luminosité, présence, distance, ...), qu'il transforme en un signal électrique (logique, analogique ou numérique) qui pourra être traité par le reste du système.

Par exemple, avec la micro:bit :

⋆ Deux boutons `button_a` et `button_b` de méthodes :

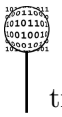
`is_pressed()`	renvoie `True` ou `False` selon si le bouton A est enfoncé,
`was_pressed()`	renvoie `True` ou `False` selon si le bouton A a été appuyé et relâché,
`get_presses()`	renvoie et remet à zéro le nombre de fois où A a été appuyé.

⋆ Un accéléromètre `accelerometer` de méthodes :

`get_x()`	renvoie un entier relatif indiquant la direction selon l'axe (Ox),
`get_y()`	renvoie un entier relatif indiquant la direction selon l'axe (Oy),
`get_z()`	renvoie un entier relatif indiquant la direction selon l'axe (Oz),
`get_values()`	renvoie un triplet d'entiers relatifs indiquant la direction (x,y,z),
`current_gesture()`	renvoie le nom du geste en cours.

MicroPython comprend les gestes : `"up"`, `"down"`, `"left"`, `"right"`, `"face up"`, `"face down"`, `"freefall"`, `"3g"`, `"6g"`, `"8g"`, `"shake"`.

- ⋆ Une boussole `compass` de méthodes : `calibrate()` (indispensable en début d'utilisation), `is_calibrated()` et `clear_calibration()` .
  `get_x()` , `get_y()` , `get_z()` comme pour l'accéléromètre.
  `heading()` qui renvoie un angle entier positif en degrés (0 pour le nord à 360).
  `get_field_strength()` qui renvoie un entier représentant l'intensité magnétique.
- ⋆ Un capteur de température, donnée par `temperature()` .



Actionneur
Un actionneur est un périphérique de sortie. Il permet la conversion de l'énergie (électrique) en une action physique (mécanique, lumineuse, . . .).

La micro:bit seule ne possède qu'un actionneur : une grille de LED 5×5 : `display` .
Ses méthodes principales :

- ⋆ `get_pixel(x, y)` renvoie un entier entre 0 et 9 d'intensité lumineuse,
- ⋆ `set_pixel(x, y, valeur)` règle l'intensité lumineuse par une valeur entre 0 et 9,
- ⋆ `clear()` règle toutes les intensités lumineuses à 0,
- ⋆ `show(value, delay=400, wait=True, loop=False, clear=False)` :
  `value` peut être un nombre, une chaine de caractères, une image ou une liste d'images.
  Une image pouvant être l'une des images prédéfinies, `Image.HEART` , `Image.HEART_SMALL` , `Image.HAPPY` , `Image.SMILE` , `Image.SAD` . . . ou créée en indiquant l'intensité lumineuse de chaque LED (de gauche à droite et de haut en bas) :
  `image = Image("90009:09090:00900:09090:90009")` .
  `delay` est le temps en microsecondes entre deux images d'une liste.
  `wait` bloque l'exécution si `True` , affiche en tache de fond si `False` .
  `loop` détermine s'il y a répétition ou non.
  `clear` détermine si l'affichage est effacé après l'exécution.
- ⋆ `scroll(value, delay=150, wait=True, loop=False, monospace=False)` .

2 - Réalisation d'une Interface Homme Machine

Si l'on veut faire se déplacer un point lumineux selon l'inclinaison de la carte et terminer en fixant sa position, on a besoin de :

- ⋆ initialiser la position. Par exemple au centre : `x = 3` et `y = 3` .
- ⋆ utiliser une boucle qui ne s'arrête que si l'utilisateur le demande par une action,
  Par exemple : `while not button_a.is_pressed() :`
- ⋆ lire les données du bon capteur.
  Ici : `accelerometer.get_x()` et `accelerometer.get_y()` .
- ⋆ utiliser ces données pour mettre à jour les variables du programme.
- ⋆ agir sur le bon actionneur en fonction de cette mise à jour.
  Ici : `display.set_pixel(int(x),int(y),9)`

Ce qui donne finalement :

```
from microbit import *

x = 3
y = 3
while not button_a.is_pressed() :
    x = x + accelerometer.get_x() / 1000
    y = y + accelerometer.get_y() / 1000
    if(x>4): x = 4
    if(x<0): x = 0
    if(y>4): y = 4
    if(y<0): y = 0
    sleep(100)
    display.clear()
    display.set_pixel(int(x),int(y),9)
```

Langages et programmation

- Programme officiel

Les langages de programmation Turing-complets sont caractérisés par un corpus de « constructions élémentaires ». Sans introduire cette terminologie, il s'agit de montrer qu'il existe de nombreux langages de programmation, différents par leur style (impératif, fonctionnel, objet, logique, événementiel, etc.), ainsi que des langages formalisés de description ou de requêtes qui ne sont pas des langages de programmation.

L'importance de la spécification, de la documentation et des tests est à présenter, ainsi que l'intérêt de la modularisation qui permet la réutilisation de programmes et la mise à disposition de bibliothèques. Pour les programmes simples écrits par les élèves, on peut se contenter d'une spécification rapide mais précise.

Contenus	Capacités attendues	Commentaires
Constructions élémentaires	Mettre en évidence un corpus de constructions élémentaires.	Séquences, affectation, conditionnelles, boucles bornées, boucles non bornées, appels de fonction.
Diversité et unité des langages de programmation	Repérer, dans un nouveau langage de programmation, les traits communs et les traits particuliers à ce langage.	Les manières dont un même programme simple s'écrit dans différents langages sont comparées.
Spécification	Prototyper une fonction. Décrire les préconditions sur les arguments. Décrire des postconditions sur les résultats.	Des assertions peuvent être utilisées pour garantir des préconditions ou des postconditions.
Mise au point de programmes	Utiliser des jeux de tests.	L'importance de la qualité et du nombre des tests est mise en évidence. Le succès d'un jeu de tests ne garantit pas la correction d'un programme.
Utilisation de bibliothèques	Utiliser la documentation d'une bibliothèque.	Aucune connaissance exhaustive d'une bibliothèque particulière n'est exigible.

Langages et programmation(I)

Constructions élémentaires

Cette partie est volontairement non détaillée pour tenir sur une page de référence. Ces notions ont déjà été rencontrées en particulier en mathématiques avant l'entrée en Première.

Valeurs

- ⋆ `1` est de type `int`
- ⋆ `1.0` est de type `float`
- ⋆ `True` est de type `bool`
- ⋆ `'1'` est de type `str`
- ⋆ `[1, 2]` est de type `list`
- ⋆ `(1, 2)` est de type `tuple`
- ⋆ `{'un' : 1, 'deux' : 2}` de type `dict`

Variables

L'affectation $x \leftarrow 1$: `x = 1`.

Conditionnelle

1 $x \leftarrow 96$
2 **si** x *est pair* **alors** $x \leftarrow \frac{x}{2}$

```
x = 96
if x % 2 == 2 :
    x = x // 2
```

1 $x \leftarrow 96$
2 **si** x *est pair* **alors** $x \leftarrow \frac{x}{2}$
3 **sinon** $x \leftarrow x * 3 + 1$

```
x = 96
if x % 2 == 2 :
    x = x // 2
else :
    x = x * 3 + 1
```

1 $x \leftarrow$ entier aléatoire entre -5 et 5
2 **si** $x > 0$ **alors** $x \leftarrow x + 1$
3 **sinon si** $x < 0$ **alors** $x \leftarrow x - 1$
4 **sinon** $x \leftarrow 1$ ou -1 au hasard

```
from random import randint

x = randint(-5,5)
if x > 0 :
    x = x + 1
elif x < 0 :
    x = x - 1
else :
    x = 1 - 2 * randint(0,1)
```

Boucle non bornée

1 $x \leftarrow 96$
2 **tant que** x *est pair* **faire**
3 $\quad x \leftarrow \frac{x}{2}$

```
x = 96
while x % 2 == 0 :
    x = x // 2
```

Boucle bornée

1 **pour** k **allant de** 0 **à** 9 **faire**
2 $\quad$ affiche(k)

```
for k in range (10) :
    print(k)
```

- ⋆ `range(n)` de 0 à $n-1$
- ⋆ `range(d,n)` de d à $n-1$
- ⋆ `range(d,n,p)` de d à $n-1$ de p en p

Avec d, n et p entiers.

Peut aussi être utilisé directement sur les types `str`, `list`, `tuple` et `dict`.

Fonction

```
def f(x) :
    x = x * 2
    return x

x = 3
y = f(x)
print('f(',x,') = ',y, sep = '')
```

La variable `x` de la fonction est dite locale. Elle copie la référence obtenue par le paramètre. Changer sa valeur ne change donc pas celle de la variable `x` externe à la fonction.

Pour un type construit (un tableau), cette référence donne accès aux éléments qui, si modifiés, le sont alors de manière globale.

Langages et programmation(II)

Différents langages

En introduction, voici quelques façons de produire dans différents langages, l'affichage des mots « Hello, world ! », à commencer par Python :

```python
print('Hello, world!')
```

Langage Bash :

```bash
#!/bin/bash
echo 'Hello, world!'
```

Langage C :

```c
#include <stdio.h>

int main()
{
    printf("hello, world");
    return 0;
}
```

Langage HTML5 :

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Hello, world!</title>
    </head>
    <body>
        <p>Hello, world!</p>
    </body>
</html>
```

Langage Java :

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Langage JavaScript :

```javascript
console.log("Hello, world!");
```

Langage LaTeX(utilisé pour ce document) :

```latex
\documentclass{minimal}
\begin{document}
Hello, world!
\end{document}
```

Langage PHP :

```php
<?php
 echo "Hello, world!";
?>
```

À première vue, ils semblent très différents, mais le plus important est qu'ils peuvent tous faire une même chose : produire un affichage de texte !

HTML et LATEX sont deux langages à balises de définition de documents. On peut identifier `<html>...</html>` et `\begin{document}...\end{document}`. L'un destine son résultat à l'écran et l'autre à l'impression, mais ils sont très comparables.

Python, C, Java, ... sont des langages de programmation impératifs. On y retrouve

- des séquences d'instructions, séparées par un simple passage à la ligne pour Python, par un point-virgule `;` dans beaucoup d'autres langages ;
- des affectations, `x = 3` en Python, C, Java, JavaScript... ; `$x = 3` en PHP ;
  (`x := 3` , `x ← 3` ou encore `3 → x` ... dans d'autres langages.)
- des instructions conditionnelles :

  en Python :

  ```python
  if (x == 3) :
      print("Ok")
  ```

  en JavaScript :

  ```javascript
  if (x == 3) {
      console.log("Ok");
  }
  ```

  Dans beaucoup de langages, les blocs d'instructions sont délimités par des accolades `{ ... }`, quand, en Python, ils commencent après `:` et sont identifiés par une indentation.

⋆ des boucles :

boucle `pour` en Python :

```
chiffres = ""
for i in range(10) :
    chiffres = chiffres + i
```

en JavaScript :

```
var chiffres = "";
for (var i = 0; i < 10; i++) {
  chiffres = chiffres + i;
}
```

En JavaScript la boucle `pour` est définie par une initialisation, une condition à vérifier pour continuer et une instruction à exécuter en fin de boucle. (Une autre syntaxe, proche de celle de Python existe en JavaScript). Certains langages sont plus proches du `pour` algorithmique.

La boucle `while` est semblable pour les langages proposés ici. On remplace `if` par `while` pour répéter le bloc conditionnel.

⋆ des branchements.

Il s'agit de possibilités de sauter d'une partie du programme à une autre. En particulier l'appel à une fonction.

Fonction en Python :

```
def f(x) :
    return x * x
```

En JavaScript :

```
function f(x) {
  return x * x;
}
```

En C :

```
int f(int x)
{
    return x * x;
}
```

Et l'appel (commun à ces langages) `f(7)` renvoie 49.

Il existe d'autres types de programmation qui pourront être discutés en Terminale.

Langages et programmation(III)

Spécification et tests

1 - Prototype d'une fonction

Prototype

Le prototype d'une fonction est sa *signature*. Il définit :

- ⋆ le nom de la fonction,
- ⋆ le nombre et le type de ses paramètres,
- ⋆ le type de la valeur qu'elle renvoie.

Avec les langages C ou Java par exemple, on écrit le prototype des fonctions. Cela permet par exemple d'écrire la partie principale en début de code contrairement au langage Python. La commande `def` définit la fonction en place ; elle n'a aucune existence pour le code la précédant.

Fonction en Python :

(Pas de prototype)

```
1 def f(x) :  # Définition en place
2     return x * x
```

En C :

```
1 int f(int x)  // Prototype
2 {
3     return x * x;
4 }
```

Un des avantages d'une fonction sans prototype est qu'elle peut éventuellement être utilisée avec des valeurs de types différents : `f(3.5)` sera évalué correctement en Python, mais sera refusé par le compilateur C qui attend un argument entier.

Un des inconvénients est qu'on perd en lisibilité sur l'intention de la fonction. C'est pourquoi il est important de commenter son code, en particulier chaque fonction pour préciser sa spécification.

2 - Spécification

Spécification

La spécification d'un algorithme (et donc d'une fonction) définit :

- ⋆ les entrées et sorties
- ⋆ le rôle
- ⋆ les préconditions
- ⋆ les postconditions

- ⋆ Les entrées et sorties sont ce qui fait le prototype d'une fonction.
- ⋆ Le rôle est une description du problème que résout l'algorithme (la fonction).
- ⋆ Les préconditions sont des précisions sur la nature exacte des entrées.
- ⋆ Les postconditions sont des précisions sur la nature exacte des sorties.

Les préconisations sont les hypothèses (au sens mathématique) sur les entrées, permettant de démontrer que l'algorithme (ou la fonction) joue bien son rôle, c'est à dire produit des sorties vérifiant les postconditions.

Avec le langage Python on écrit la spécification avec le code dans des *docstring*. C'est un texte entouré de guillemets triples qui se place juste en début de code :

```
1 def f(x):
2     """Renvoie le carré du nombre donné en argument."""
3     return x * x
```

Pour un algorithme simple comme celui-ci, une seule ligne précisant son rôle peut suffire. Mais il n'est pas rare que le *docstring* soit plus long que le code lui-même :

```
def centSur(x):
    """Calcule le quotient entier de 100 par x.

    Cette fonction opère la division entière de 100 par x et en
    renvoie le quotient.

    :param x: Un entier non nul.
    :type x: int
    :return: Le quotient de la division entière de 100 par x
    :rtype: int
    """
    return 100 // x
```

Cette exemple utilise la syntaxe RST mais ce n'est pas une obligation.
On obtient alors l'affichage de cette documentation avec la commande `help(centSur)`.

3 - Tests et assertions

Un *docstring* permet de documenter en écrivant la spécification des fonctions. Il permet aussi la mise au point des programmes à l'aide de jeux de tests. En effet, juste après la spécification, et avant même de réfléchir à l'algorithme, on peut mettre en place un jeu de tests qu'il devrait passer pour avoir une chance de résoudre le problème posé.

```
def centSur(x):
    """Calcule le quotient entier de 100 par x.

    >>> centSur(3)
    33
    """
    return 100 // x
```

La bibliothèque `doctest` permet effectivement de lancer tous les tests par la commande `doctest.testmod()`. Pour voir tous les tests effectués, on peut compléter cette commande en `doctest.testmod(verbose = True)`.

La réussite à ces tests n'est pas une preuve de correction, mais permet de guider la programmation et l'utilisateur.

En phase de conception ou plutôt de débogage, on peut vouloir s'assurer que les préconditions sont effectivement remplies. Dans ce cas, on utilise des assertions : on exprime chacune des préconditions après le mot clef `assert`.

```
def centSur(x):
    """Calcule le quotient entier de 100 par x."""
    assert isinstance(x, int)
    assert x != 0
    return 100 // x
```

Avec ce code, par exemple, `centSur(5.0)` provoquera une erreur, alors que sans la première assertion, il aurait donné une réponse.

Langages et programmation(IV)

Utilisation de bibliothèques

Bibliothèque

Une bibliothèque, on dit souvent un *module* en Python, est un fichier contenant un ensemble de fonctions, de classes (types personnalisés), et de constantes, que l'on peut importer dans son propre programme pour les utiliser.

1 - Utilisation de modules existants

Il existe de très nombreux modules en Python (c'est un des avantages de ce langage). Certains sont dits *standards* et fournis avec l'installation du langage. D'autres doivent être installés manuellement.

Parmi les standards, notons :

- ⋆ `random` dont la fonction `randint()` permet d'obtenir un entier aléatoire dans un intervalle,
- ⋆ `math` qui contient par exemple `cos()` et `sin()` ,
- ⋆ `doctest` pour implémenter des tests automatiques de fonctions,
- ⋆ `csv` pour le traitement de tables CSV,
- ⋆ `json` pour le traitement de données JSON,
- ⋆ `socket` pour les connexions TCP/IP,
- ⋆ `os` et `sys` pour les fonctions système.

Parmi les autres, on peut citer :

- ⋆ `PIL` pour le traitement d'images (à charger sous le nom `pillow`),
- ⋆ `requests` pour les requêtes HTTP GET et POST,
- ⋆ `http.server` pour un serveur HTTP (réponse aux GET et POST).

Pour utiliser une bibliothèque déjà installée, on peut employer une des méthodes suivantes :

```
1 import random
2
3 x = random.randint(1,9)
```

```
1 from random import randint
2
3 x = randint(1,9)
```

```
1 import random as hasard
2
3 x = hasard.randint(1,9)
```

```
1 from random import randint as hasard
2
3 x = hasard(1,9)
```

Pour une bibliothèque non standard, il faut passer en ligne de commande pour la charger et l'installer avec le module `pip` . Par exemple pour `PIL` :

```
PS C:\WINDOWS\system32> py -m pip install Pillow
Collecting Pillow
  Downloading https://files.pythonhosted.org/packages/ec/ca/7af5b6628ecf770645f8cc3c9da3c2bb5c5ffc7384a9ff0666fdb818b4d5
/Pillow-5.4.1-cp36-cp36m-win_amd64.whl (1.9MB)
    100% |████████████████████████████████| 1.9MB 436kB/s
Installing collected packages: Pillow
Successfully installed Pillow-5.4.1
```

2 - Conception de modules

N'importe quel fichier Python est un module dont le nom est celui du fichier. Il est donc très simple d'en concevoir un :

Fichier `Bonjour.py` :

```
""" Ceci est le module Bonjour """

def hello():
    """ Affiche : Bonjour ! """
    print('Bonjour !')

hello()
```

Script dans le même répertoire :

```
import Bonjour

Bonjour.hello()
```

Ce second script s'exécute bien...
Mais dit bonjour deux fois !

Au moment de l'importation, le script contenu dans le module est entièrement exécuté. Comme il contient un appel à la fonction `hello()`, on obtient un premier affichage, puis un second dû à l'appel `Bonjour.hello()` du second script.

On pourrait enlever le `hello()` du module, mais dans la pratique, on souhaite souvent garder à un module la possibilité de s'exécuter en tant que script principal. Il suffit pour cela d'ajouter une conditionnelle :

```
""" Ceci est le module Bonjour """

def hello():
    """ Affiche bonjour """
    print('Bonjour !')

if __name__ == "__main__":
    hello()
```

Par exemple, en phase de développement, on pourra utiliser :

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Si on prend l'exemple proposé pour l'explication sur les assertions, on obtient :

```
""" Module de test sur les assertions """

def centSur(x):
    """Calcule le quotient entier de 100 par x.

    Cette fonction opère la division entière de 100 par x et en
    renvoie le quotient.

    :param x: Un entier non nul
    :type x: int
    :return: Le quotient de la division entière de 100 par x
    :rtype: int

    >>> centSur(3)
    33

    >>> centSur(3.0)
    Traceback (most recent call last):
        ...
    AssertionError: x doit être entier

    >>> centSur(0)
    Traceback (most recent call last):
        ...
    AssertionError: x ne doit pas être nul
    """
```

```
27      assert isinstance(x, int), 'x doit être entier'
28      assert x != 0, 'x ne doit pas être nul'
29      return 100 // x
30
31  if __name__ == '__main__':
32      import doctest
33      doctest.testmod(verbose=True)
```

Algorithmique

- Programme officiel

Le concept de méthode algorithmique est introduit ; de nouveaux exemples seront vus en terminale. Quelques algorithmes classiques sont étudiés. L'étude de leurs coûts respectifs prend tout son sens dans le cas de données nombreuses, qui peuvent être préférentiellement des données ouvertes.

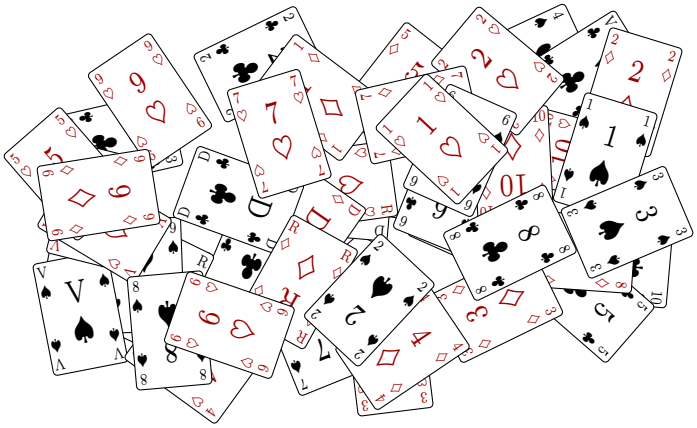
Il est nécessaire de montrer l'intérêt de prouver la correction d'un algorithme pour lequel on dispose d'une spécification précise, notamment en mobilisant la notion d'invariant sur des exemples simples. La nécessité de prouver la terminaison d'un programme est mise en évidence dès qu'on utilise une boucle non bornée (ou, en terminale, des fonctions récursives) grâce à la mobilisation de la notion de variant sur des exemples simples.

Contenus	Capacités attendues	Commentaires
Parcours séquentiel d'un tableau	Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque. Écrire un algorithme de recherche d'un extremum, de calcul d'une moyenne.	On montre que le coût est linéaire.
Tris par insertion, par sélection	Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.	La terminaison de ces algorithmes est à justifier. On montre que leur coût est quadratique dans le pire cas.
Algorithme des k plus proches voisins	Écrire un algorithme qui prédit la classe d'un élément en fonction de la classe majoritaire de ses k plus proches voisins.	Il s'agit d'un exemple d'algorithme d'apprentissage.
Recherche dichotomique dans un tableau trié	Montrer la terminaison de la recherche dichotomique à l'aide d'un variant de boucle.	Des assertions peuvent être utilisées. La preuve de la correction peut être présentée par le professeur.
Algorithmes gloutons	Résoudre un problème grâce à un algorithme glouton.	Exemples : problèmes du sac à dos ou du rendu de monnaie. Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale.

Algorithmique(I)

Recherches dans un tableau

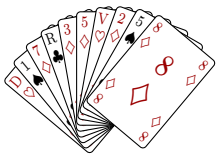
Comment trouver une information parmi beaucoup d'autres ?



Une solution élémentaire est de faire un tas et de passer en revue toutes les cartes une à une.

1 - Parcours séquentiel d'un tableau

Pour simplifier l'exemple, supposons que l'on ait tiré une main et notons les valeurs des cartes dans un tableau :



DC	1P	7c	RT	3c	5c	VC	2c	5P	8c

```
cartes = ['DC', '1P', '7c', 'RT', '3c', '5c', 'VC', '2c', '5P', '8c']
```

a) Spécification possible

- ⋆ **Entrée :** Tableau T de longueur n, et une valeur v.
- ⋆ **Sortie :** Un indice i de v dans T ou -1 si v n'est pas dans T.
- ⋆ **Rôle :** Recherche d'une occurrence de v dans T.
- ⋆ **Précondition :** v est comparable avec les éléments de T.
- ⋆ **Postcondition :** ($i = -1$ et v n'est pas dans T) ou ($0 \leqslant i < n$ et $T[i]$ est v).

b) Algorithme

```
1 i ← −1
2 pour carte allant de 0 à n − 1 faire
3 |   si T[carte] == v alors
4 |   |   i ← carte
```

c) Correction et complexité

⋆ **Correction partielle :** On peut prendre comme invariant, pour la boucle `pour`, la post-condition limitée au sous-tableau déjà parcouru.

Invariant : ($i = -1$ et si $0 \leqslant j \leqslant carte$ alors $\mathrm{T}[j] \neq v$) ou ($0 \leqslant i \leqslant carte$ et $\mathrm{T}[i] = v$).

Avant la première itération $carte$ n'est pas définie et on a bien $i = -1$.

Après la première itération $carte = 0$; si la condition $\mathrm{T}[carte] == v$ a été vérifiée, alors on a $0 = i = carte$ donc $0 \leqslant i \leqslant carte$ et $\mathrm{T}[i] = v$; sinon, on a toujours $i = -1$ et pour $0 \leqslant j \leqslant carte$ c'est à dire $j = 0$, on a aussi $\mathrm{T}[j] \neq v$. L'invariant est donc vérifié.

S'il est vérifié avant une itération, alors on peut distinguer 4 cas :

— ($i = -1$ et si $0 \leqslant j \leqslant carte$ alors $\mathrm{T}[j] \neq v$) et la conditionnelle se réalise ;

— ($i = -1$ et si $0 \leqslant j \leqslant carte$ alors $\mathrm{T}[j] \neq v$) et la conditionnelle ne se réalise pas ;

— ($0 \leqslant i \leqslant carte$ et $\mathrm{T}[i] = v$) et la conditionnelle se réalise ;

— ($0 \leqslant i \leqslant carte$ et $\mathrm{T}[i] = v$) et la conditionnelle ne se réalise pas.

Dans chaque cas, on peut vérifier comme précédemment que l'invariant est bien conservé.

Enfin, l'expression de l'invariant après la dernière itération donne la postcondition.

⋆ **Terminaison :** Une boucle `pour` termine.

⋆ **Complexité :** La taille des entrées est la longueur n du tableau.

Le pire cas vient d'une réalisation systématique de la condition $\mathrm{T}[carte] = v$, c'est à dire un tableau dont toutes les valeurs sont égales à v.

Dans ce cas, on peut compter 3 opérations par itération (mise à jour de $carte$, comparaison de la conditionnelle, et affectation de i). La boucle faisant n itérations, on obtient en tout $3n + 1$ opérations.

$3n + 1 = O(n)$. La complexité est linéaire.

d) Applications

Appliquons l'algorithme précédent à un tableau de nombres :

```
tableau = [1, 5, 7, 3, 2, 9, 4, 6]
```

Recherche d'un maximum

La valeur recherchée n'est pas connue, mais vérifie un critère.

Ajoutons la **précondition** suivante pour initialiser : T n'est pas vide.

On peut alors adapter l'algorithme précédent :

```
1 max ← T[0]
2 pour i allant de 0 à n − 1 faire
3 |   si T[i] > max alors
4 |   |   max ← T[i]
```

Une preuve similaire s'applique et la complexité est toujours linéaire.

Calcul d'une moyenne

Toutes les valeurs sont utilisées : l'algorithme est simplifié (plus de conditionnelle).

```
1 somme ← 0
2 pour i allant de 0 à n − 1 faire
3 |  somme ← somme + T[i]
4 moyenne ← somme/n
```

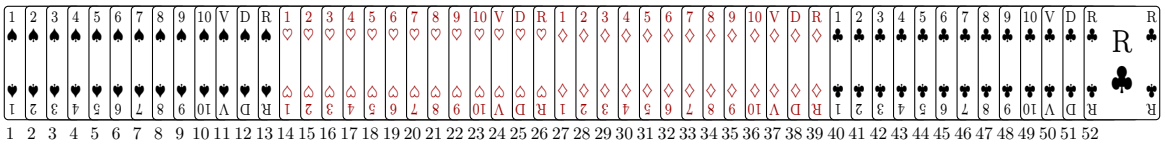
Un invariant possible pour la correction partielle : *somme* est la somme des valeurs de T d'indices inférieurs à i.

La complexité est toujours linéaire.

2 - Recherche dichotomique

On rajoute ici une précondition importante : Le tableau est *trié*.

On pourrait définir plusieurs critères (couleur / valeur / points dans un jeu ...). Dans ce qui suit, on supposera établi un ordre total qui permet la numérotation :



On peut réécrire notre main :

25	1	33	52	29	31	24	28	5	34

```
cartes = [25,1,33,52,29,31,24,28,5,34]
```

Et la trier :

1	5	24	25	28	29	31	33	34	52

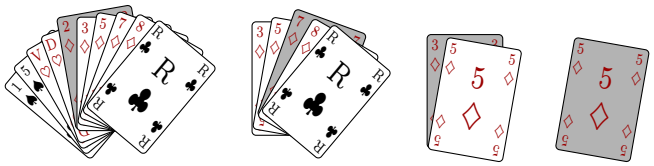
```
cartes = [1,5,24,25,28,29,31,33,34,52]
```

On ajoute à la **précondition :** T est non vide et trié.

a) Algorithme

Une recherche *dichotomique* consiste, par comparaison, à éliminer au moins la moitié des possibilités à chaque étape. L'idée est donc de faire ces comparaisons avec des médianes parmi des valeurs triées.

Sur l'exemple, cherchons le cinq de carreau (le 31). Pour la médiane du sous tableau entre les indices min et max, on choisit la valeur d'indice $(min + max)//2$:



min	1	2	3	med	5	6	7	8	max
1	5	24	25	28	29	31	33	34	52

$(0+9)//2 = 4$ et la médiane est $28 < 31$

min	6	med	8	max
29	31	33	34	52

$(5+9)//2 = 7$ et la médiane est $33 > 31$

min	max
29	31

$(5+6)//2 = 5$ et la médiane est $29 < 31$

$min = max$
31

```
1  min ← 0
2  med ← 0
3  max ← n − 1
4  tant que min < max faire
5  |   med ← (min + max)//2
6  |   si T[med] < v alors
7  |   |   min ← med + 1
8  |   sinon si T[med] > v alors
9  |   |   max ← med − 1
10 |   sinon
11 |   |   max ← med
12 |   |   min ← med
13 si T[min] = v alors
14 |   i ← min
15 sinon
16 |   i ← −1
```

b) Correction

⋆ **Correction partielle**

Montrons l'invariant : « $T[min] \leqslant v \leqslant T[max]$ ou v n'est pas dans le tableau ».

Il est vrai au début de la boucle puisque tout le tableau est entre min et max.

Supposons-le vrai au début d'une itération.

Comme on ne modifie jamais le tableau, si v n'est pas dans le tableau, il n'y sera jamais.

On va donc supposer que v est dans le tableau.

— Comme $min < max$, le calcul (ligne 6) de med donne : $min \leqslant med < max$.

— Cas du passage ligne 7 :

Comme le tableau est trié, avant l'exécution : $T[min] \leqslant T[med] < v \leqslant T[max]$

Si on avait $T[med+1] > v$ alors on aurait $T[med] < v < T[med+1]$ et v ne serait pas dans le tableau, donc $T[med+1] \leqslant v \leqslant T[max]$ et la ligne 7 conserve l'invariant.

— Cas du passage ligne 9 :

Comme le tableau est trié, avant l'exécution : $T[min] \leqslant v < T[med] \leqslant T[max]$

Si on avait $T[med-1] < v$ alors on aurait $T[med-1] < v < T[med]$ et v ne serait pas dans le tableau, donc $T[min] \leqslant v \leqslant T[med-1]$ et la ligne 9 conserve l'invariant.

— Cas du passage lignes 11 et 12 :

En excluant les deux inégalités précédentes, il ne reste que $T[med] = v$

Les lignes 11 et 12 conservent donc l'invariant.

— L'invariant est donc bien toujours vérifié.

Si on sort de la boucle : $min \geqslant max$ donc $T[min] \geqslant T[max]$.

L'invariant donne alors « $T[min] = v = T[max]$ ou v n'est pas dans le tableau »

Donc v est dans le tableau si et seulement si $T[min] = v$.

⋆ **Terminaison :**

L'expression $max - min$ est minorée par la condition de la boucle à 0.

Si on ne passe jamais par les lignes 11 et 12, $max - min$ est strictement décroissante.

Dans ce cas on a bien un variant qui permet d'assurer que l'algorithme termine.

Si on passe par les lignes 11 et 12 alors $max - min$ devient 0 et l'algorithme termine.

c) Complexité

La taille des entrées est la longueur n du tableau.

Comme on élimine à chaque itération au moins la moitié des valeurs du tableau, si on peut déterminer pour quel entier N on a $2^{N-1} < n \leqslant 2^N$, et si on note n_k la longueur du tableau restant après k itérations :

$n_0 = n \leqslant 2^N$ puis $n_1 \leqslant \frac{n}{2} \leqslant 2^{N-1}$ puis $n_2 \leqslant 2^{N-2}$ ... puis $n_N \leqslant 2^{N-N} = 1$.

Donc il faut au plus N itérations.

Il existe une fonction mathématique : $\log_2()$ qui donne le réel vérifiant $n = 2^{\log_2(n)}$. On peut donc majorer le nombre d'itérations par l'entier directement supérieur à $\log_2(n)$.

On dit que la complexité est logarithmique et on la note simplement : $O(\log(n))$

Algorithmique(II)

Tris de tableaux

A l'ère de l'open data, la difficulté n'est plus guère de trouver de l'information mais de trier les informations disponibles pour réellement trouver celles que l'on cherche. Il existe de nombreux algorithmes de tri, certains très généraux et d'autres plus spécifiques selon les préconditions du problème. Donnons une spécification générale et traitons de deux exemples : le tri par insertion et le tri par sélection.

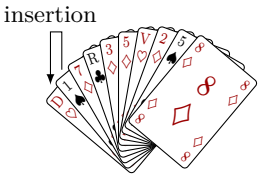
Spécification :

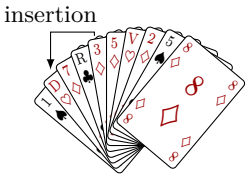
- ⋆ **Entrée / Sortie :** Tableau T de longueur n.
- ⋆ **Rôle :** Trier les valeur dans T par ordre croissant.
- ⋆ **Précondition :** Les valeurs de T sont toutes comparables.
- ⋆ **Postcondition :** Si $0 \leqslant i \leqslant j \leqslant n-1$ alors $T[i] \leqslant T[j]$.
  Et les valeurs de T sont les mêmes (dans un autre ordre) en entrée et en sortie.

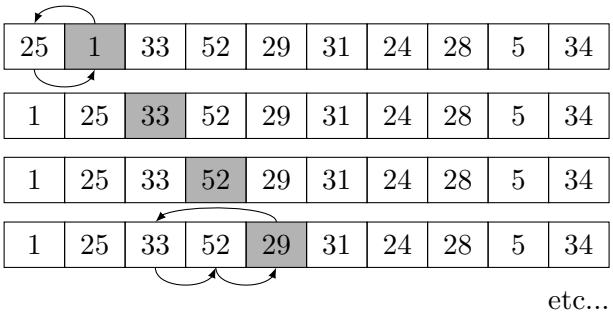
1 - Tri par insertion

a) Algorithme

C'est un des tris les plus simples. On prend une carte à la fois dans l'ordre de la main (de gauche à droite) et on l'insère à sa place parmi les cartes que l'on a déjà rangées (à gauche).








```
1 pour i allant de 1 à n − 1 faire
2 |   insert ← Tableau[i]
3 |   j ← i
4 |   tant que j > 0 et Tableau[j − 1] > insert faire
5 |   |   Tableau[j] ← Tableau[j − 1]
6 |   |   j ← j − 1
7 |   Tableau[j] ← insert
```

b) Correction

Correction partielle

Reformulons la première postcondition ($0 \leqslant p \leqslant k \leqslant n-1 \Rightarrow T[p] \leqslant T[k]$).

Un invariant de la boucle `pour` serait : ($0 \leqslant p \leqslant k \leqslant i \Rightarrow T[p] \leqslant T[k]$).

Si le tableau est vide, il ne se passe rien. S'il ne contient qu'un élément, l'invariant est vrai pour $i = 0$ car il ne concerne qu'un élément et il ne se passe rien d'autre. Supposons donc qu'il

soit vrai avant le début d'une itération, pour $i \geqslant 0$, et démontrons le, après l'itération, pour $i' = i + 1$.

On a donc $(0 \leqslant p \leqslant k \leqslant i \Rightarrow T[p] \leqslant T[k])$.

Ligne 2, on mémorise $T[i']$ dans la variable insert

Si $T[i'] > T[i]$, la boucle `tant que` ne fait rien, donc les lignes 3 et 7 non plus, et l'invariant est vérifié.

Sinon, dès la première ligne de la boucle `tant que`, $T[i'] \leftarrow T[i]$ donc l'invariant est vérifié, et chaque autre itération de cette boucle conserve l'invariant.

A la sortie, toutes les valeurs d'indice supérieur à j sont plus grandes que insert et les autres plus petites, donc la ligne 7 conserve l'invariant.

Pour la seconde postcondition, on peut constater qu'à chaque itération, en incluant la variable insert, aucune valeur du tableau n'est perdue, car on affecte uniquement sur des doublons. De plus, la dernière affectation provoque un doublon de la variable insert dans le tableau. Donc la seconde postcondition est un invariant de la boucle `pour`.

Terminaison

Une boucle `pour` termine toujours.

Pour la boucle `tant que` le variant est simplement `j` qui décroit strictement (de 1 à chaque itération) et est minoré par 0 (condition de la boucle).

Donc cet algorithme termine.

c) Complexité

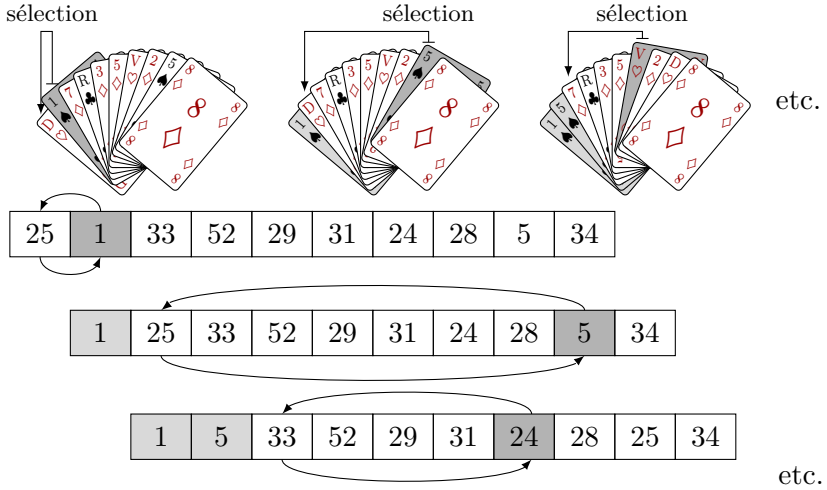
- ⋆ La taille des données à traiter est la taille n du tableau.
- ⋆ L'opération significative est la modification du tableau : Tableau$[i] \leftarrow$ ...
- ⋆ Cette opération apparaît directement une fois dans la boucle `pour` donc $n - 1$ fois.
  Elle apparaît également une fois dans la boucle `tant que` :
  - — Le pire cas est donc celui qui retarde au maximum la sortie de cette boucle, c'est à dire quand toutes les valeurs déjà triées sont plus grandes que la suivante.
    Pire cas : Tableau trié à l'envers.
  - — Dans ce cas il y a i affectations dans le tableau.
  - — Au total il y en a donc $1 + 2 + 3 + \ldots + (n - 1)$.

  Chacun des $(n - 1)$ termes de cette somme est inférieur à n.
  On peut donc la majorer : $1+2+3+\ldots+(n-1) \leqslant n+n+n+\ldots+n = n(n-1) = O(n^2)$.
  (Un peu de mathématiques montre qu'elle vaut $\frac{n(n-1)}{2}$ mais toujours $O(n^2)$.)
- ⋆ Bilan : $n - 1 + O(n^2) = O(n^2)$. La complexité est quadratique.

2 - Tri par sélection

a) Algorithme

Le tri par sélection consiste à sélectionner la carte que l'on souhaite ranger au lieu de prendre la première venue. Les cartes sont choisies de la plus petite à la plus grande et positionnées directement à leurs places définitives (par échange, sans décalage).




```
1 pour i allant de 0 à tailleTableau − 1 faire
2     imin ← i
3     pour j allant de i + 1 à tailleTableau − 1 faire
4         si T[imin] > T[j] alors
5             imin ← j
6     T[imin], T[i] ← T[i], T[imin]
```

b) Preuve de correction

Correction partielle

Les éléments du tableau ne changent pas : on ne fait que des permutations simples.

Pour l'autre partie de la postcondition, on peut montrer que jusqu'à i le tableau est trié et que tous les éléments suivants sont plus grands.

L'invariant : $(0 \leqslant p \leqslant k \leqslant i < m \leqslant n-1 \Rightarrow T[p] \leqslant T[k] \leqslant T[m])$.

On a déjà discuté de la recherche d'extremum. C'est ce que fait la boucle `pour j`

La boucle `pour i` se contente donc de mettre en i la plus petite valeur parmi celles d'indice supérieur. On peut donc vérifier l'invariant.

Terminaison

Deux boucles `pour` donc termine.

c) Complexité

Il y a toujours n itérations de la boucle `pour i` donc autant d'échanges de valeurs dans le tableau. À chaque fois il y a $n-1-i$ itérations de la boucle `pour i` donc autant de comparaisons de valeurs du tableau.

Au total : $n + (n-1-0) + (n-1-1) + (n-1-2) + \ldots + 1 = O(n^2)$

La complexité est quadratique.

La différence par rapport au tri par insertion est le nombre de modifications de valeurs dans le tableau. Si on ne compte que ces opérations (en admettant qu'elles soient lourdes par rapport aux comparaisons) alors le coût n'est plus que linéaire !

Algorithmique(III)

Exemple d'algorithme d'apprentissage

1 - Un mot sur l'Intelligence Artificielle

On parle d'intelligence artificielle, dès lors qu'une machine est en mesure de rivaliser avec l'homme face à un problème réputé nécessiter son intelligence. Mais comme l'intelligence humaine elle-même est difficile à définir, la frontière entre l'IA et les autres algorithmes est imprécise.

On peut citer quelques chantiers du domaine de l'IA...

- ⋆ Raisonner.
- ⋆ Planifier.
- ⋆ Restreindre un espace de recherche.
- ⋆ Parler, lire.
- ⋆ Comprendre une langue.
- ⋆ Apprendre.

On peut également citer quelques dates clés :

1948 Turing parle de Machine Intelligence.

En 1997, Deep Blue bat Kasparov (aux échecs).

En 2005, des voitures autonomes sont construites.

2014 Le test de Turing est passé.

En 2015 : IA obtient un score d'enfant de 4 ans à un test de QI.

En 2016-2017 : IA bat les meilleurs joueurs mondiaux au Go & Poker.

Le poumon actuel de l'IA est l'apprentissage. L'idée est de fournir des données assez riches, sur lesquelles des algorithmes, éventuellement simples, se basent pour évoluer vers le traitement de problèmes sophistiqués. Le risque est de fournir des données biaisées ; le but peut être d'abstraire (remplacer les données par une loi), de généraliser (à des données qu'on n'a pas vues), de compresser, d'oublier...

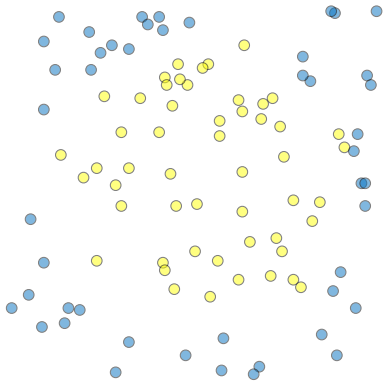
2 - Exemple de problème

On dispose d'un jeu de données : des points dans le plan.

À chaque donnée est associée une classe : chaque point est soit jaune soit bleu.

Le problème est de déterminer une classe pour un nouveau point : jaune ou bleu ?

La réponse que l'on souhaite apporter : jaune s'il est avec les jaunes (au centre) et bleu s'il est avec les bleus (à l'extérieur)... ce qui demanderait à être précisé à la frontière !



On va traduire « être avec les jaunes » par « avoir plus de voisins jaunes » et on va se demander ce que veut dire être voisin. Le point « le plus » voisin peut être défini comme le plus proche, c'est à dire qui minimise une distance. On a donc besoin de mesurer la distance entre deux points.

3 - Mesure

Le cours de mathématiques de seconde donne un calcul de distance entre deux points dont on connait les coordonnées dans un repère orthonormé : $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

On pourrait également définir une distance entre deux phrases, deux situations dans un jeu d'échecs, ou deux images. La difficulté vient essentiellement du fait qu'il n'y a plus seulement deux caractéristiques (deux dimensions) mais potentiellement des millions (autant que de pixels dans une image par exemple).

On peut proposer par exemple la **distance euclidienne** :

```
1 fonction distance(pointA, pointB)
2     c ← 0
3     pour i allant de 0 à taille(pointA) − 1 faire
4         c ← c + (pointB[i] − pointA[i])²
5     renvoyer √c
```

Cet algorithme termine, donne une réponse cohérente en dimension 2 et a une complexité en $O(d)$ où d est la dimension des points.

4 - Les k plus proches voisins

On cherche toujours à savoir si un nouveau point a ou non « plus de voisins jaunes ».

Une première idée pourrait être de définir ce qu'est un voisinage : les points ne dépassant pas une certaine distance du point étudié. Cela sous-entend de choisir un seuil au delà duquel deux points ne sont plus considérés voisins. Ce n'est pas satisfaisant ici, car rien ne garantirait alors qu'un voisinage contienne effectivement des voisins !

La seconde idée, que l'on retiendra ici, est de fixer un nombre k de voisins à atteindre. On va donc déterminer la distance de tous les points à celui étudié et ne retenir que les k plus proches voisins.

a) Algorithme du 1 plus proche voisin

Si on ne retient que le voisin le plus proche :

```
1 fonction un-ppv(x, Data)
2     i ← 0
3     meilleur ← distance(Data[0], x)
4     pour j allant de 1 à taille(Data) − 1 faire
5         candidat ← distance(Data[j], x)
6         si candidat < meilleur alors
7             i ← j
8             meilleur ← candidat
9     renvoyer classe de Data[i]
```

On reconnait le parcours séquentiel d'un tableau pour la recherche d'un extremum, que l'on sait prouver, mais dont on peut préciser la complexité en tenant compte des différents calculs de distances : $O(nd)$ où n est la taille des données et d leur dimension.

Attention cependant : on peut prouver que le voisin retenu est le plus proche, mais pas que sa classe est la bonne ! De manière générale, prouver un algorithme d'IA est rarement possible.

b) Algorithme des k plus proches voisins

Cette fois $k > 1$ et on note V le tableau que l'on va remplir des indices des k plus proches voisins dans *Data* et de leurs distances à x.

Pour l'initialiser on note ∞ une valeur plus grande que les distances potentielles.

```
1  fonction k-ppv(x, k, Data)
2      pour j allant de 0 à k − 1 faire
3          V[j]["indice"] ← −1
4          V[j]["distance"] ← ∞
5      pour i allant de 0 à n − 1 faire
6          distance ← distance(Data[i], x)
7          tant que j < k et distance > V[j]["distance"] faire
8              j ← j + 1
9          si j<k alors
10             pour m allant de k − 1 à j + 1 par pas de −1 faire
11                 V[m] ← V[m − 1]
12             V[j]["distance"] ← distance
13             V[j]["indice"] ← i
14     pour j allant de 0 à k − 1 faire
15         C[j] ← Data[V[j]["indice"]]["classe"]
16     renvoyer vote(C)
```

Que l'on pourrait résumer par :

```
1  fonction k-ppv(x, k, Data)
2      V ← k voisins imaginaires très loin
3      pour chaque voisin potentiel dans Data faire
4          si il est plus proche de x qu'un de V alors
5              On insère sa distance à x et son indice dans V
6              en gardant V trié par distances
7      C ← les classes des voisins d'indices dans V
8      renvoyer vote(C)
```

Reste a définir la fonction *vote()*, par exemple par vote majoritaire. Notons *depouille* le

tableau indexé par les classes et comptabilisant les votes :

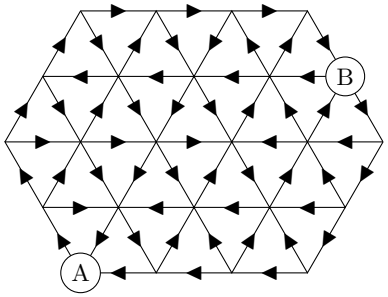
```
1  fonction vote(C)
2      pour chaque classe faire depouille[classe] ← 0
3      pour i allant de 0 à k − 1 faire
4          depouille[C[i]] ← depouille[C[i]] + 1
5      max ← 0
6      pour chaque classe faire
7          effectif ← depouille[classe]
8          si effectif > max alors
9              max ← effectif
10             resultat ← classe
11     renvoyer resultat
```

Algorithmique(IV)

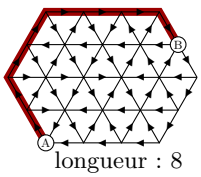
Exemple d'algorithme glouton

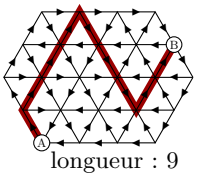
1 - Principe

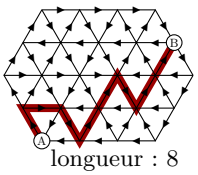
Supposons les routes à sens unique d'un centre-ville, et un automobiliste en A voulant se rendre en B : on souhaite trouver le plus court chemin.

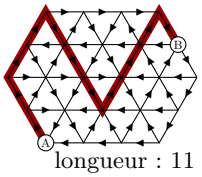


Pour être sûr de trouver le chemin le plus court, il faudrait explorer tous les chemins et retenir le plus court. On peut commencer à la main :


longueur : 8

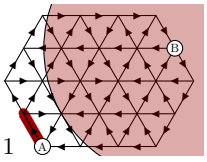

longueur : 9

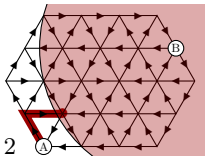

longueur : 8

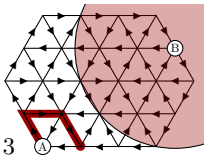

longueur : 11

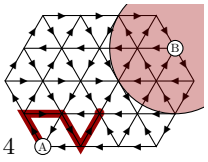
Dresser la liste de tous les chemins possibles risque d'être long (même pour un ordinateur dans des cas pas forcément très compliqués), mais, à la main, on perçoit, au bout de quelques essais, comment tracer le chemin le plus court. Par exemple on ne repasse jamais deux fois au même endroit, et on va le plus possible emprunter un chemin horizontal plutôt que deux obliques...

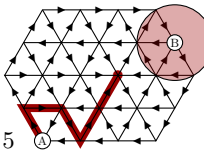
Mais une machine ne « perçoit » rien. Si on ne veut pas d'un algorithme qui passerait en revue tous les chemins, voilà une possibilité : choisir à chaque étape, et indépendamment de ce qui précède ou suit, la route qui nous rapproche le plus du but.

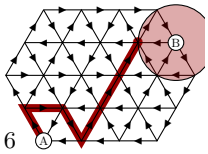

1

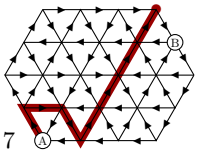

2

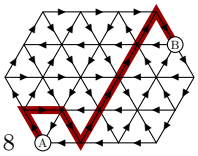

3


4


5

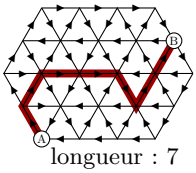

6


7


8

Il n'est pas sûr qu'un tel algorithme apporte une solution (il peut se trouver bloqué dans un cul de sac) ni qu'une solution apportée soit optimale, mais on peut garantir les deux, sous certaines conditions. Comme il a, par ailleurs, une bien meilleure complexité que l'énumération totale des possibilités, il est à privilégier dans les cas favorables.

Ici, ce n'était pas le cas :


longueur : 7

Algorithme glouton
Un algorithme est dit *glouton* si

⋆ il résout un problème pas à pas,

⋆ en faisant à chaque pas le choix du meilleur gain (ou du moindre coût).

Il tente donc de résoudre des problèmes d'*optimisation*.

Dans la spécification, le rôle sera de trouver le plus ou le moins ... et la postcondition concernera le maximum ou le minimum ...

2 - Problème du rendu de monnaie

Énoncé : Comment rendre la monnaie sur une somme S payée avec un montant M en utilisant le moins de pièces possibles.

Décontextualisation : Comment écrire une somme égale à un certain entier N avec un minimum de termes pris, avec répétition, parmi un ensemble fini $P = \{p_1, \ldots, p_k\}$

Spécification :

⋆ **Entrée :** Un entier N (à rendre), des entiers dans $P = \{p_1, \ldots, p_k\}$ (pièces).

⋆ **Sortie :** Des entiers $\{m_1, \ldots, m_k\}$ (multiplicité de chaque pièce).

⋆ **Rôle :** Écrire N comme la somme d'un minimum de $P = \{p_1, \ldots, p_k\}$.

⋆ **Précondition :** Tous les entiers en entrée sont strictement positifs.

⋆ **Postcondition :** $m_1p_1 + \ldots + m_kp_k = N$ et $m_1 + ... + m_k$ est minimal pour cette égalité.

Algorithme : Il consiste à utiliser les plus grosses pièces en premier.

```
1  fonction rendre(N, P)
2      trier(P) # par ordre décroissant
3      M ← [0, ..., 0]
4      somme ← 0
5      i ← 0
6      tant que somme < N et i < k faire
7          si somme + P[i] ⩽ N alors
8              M[i] ← M[i] + 1
9              somme ← somme + P[i]
10         sinon
11             i ← i + 1
12     renvoyer M
```

Correction : on peut démontrer qu'il termine toujours (Variant : $N - somme + k - i$).

⋆ Il est optimal pour le système européen : $P = [5, 2, 1]$.
(Par exemple $N = 8$, donne bien $M = [1, 1, 1]$.)

⋆ Il peut donner une solution non optimale :
pour $P = [6, 4, 1]$ et $N = 8$, on obtient $M = [1, 0, 2]$ au lieu de $M = [0, 2, 0]$.

⋆ Il peut ne pas donner de solution correcte :
pour $P = [5, 2]$ et $N = 8$, on obtient $M = [1, 1]$ au lieu de $M = [0, 4]$.

Complexité :

Dans le pire cas, toutes les pièces sont plus grandes que N sauf la dernière qui vaut 1. La boucle fait alors ses $N + k$ itérations qui contiennent une dizaine d'opérations.

Donc : $O(N + k) + Tri(k)$

3 - Problème du sac à dos

Énoncé : (version « Lupin ») Comment remplir un sac, à capacité limitée, avec les objets les plus précieux de tailles variés.

Décontextualisation : Quel sous-ensemble de couples (objets) parmi $O = \{o_1, \ldots, o_k\}$, où $o_i = (v_i, e_i)$ (valeur et encombrement), maximise la somme des v_i tout en ayant une somme de e_i inférieure à C (capacité).

Spécification :

- ⋆ **Entrée :** Ensemble O de couples de nombres positifs et un nombre C.
- ⋆ **Sortie :** Sous-ensemble O' de O.
- ⋆ **Rôle :** Trouver un sous-ensemble O' de O de couples $o_i = (v_i, e_i)$ tel que la somme des v_i soit maximale et celle des e_i inférieure à C.
- ⋆ **Précondition :** Tous les entiers en entrée sont strictement positifs.
- ⋆ **Postcondition :** O' est un sous-ensemble de O.

  La somme par rapport à la seconde composante des couples de O' est inférieure à C.

  Si O'' est un autre tel sous-ensemble de O, la somme par rapport à la première composante des couples de O'' est inférieure à celle de O'.

Algorithme :

- ⋆ Essayer d'énumérer toutes les possibilités revient pour chacun des k objets à choisir si oui ou non, il faut le prendre. Il y a donc 2^k choix possibles. Ce qui donne à cet algorithme une complexité exponentielle !
- ⋆ l'approche gloutonne est tentante, mais comment faire son choix ?
  - — Le plus cher en premier ?
  - — Le moins encombrant en premier ?
  - — ...

  En fait ce problème est dit « NP-difficile » et toute approche gloutonne est vouée à l'échec !

Algorithmique(V)

Concepts et méthodes

1 - Complexité

Complexité

La complexité d'un algorithme est une mesure de sa *consommation de ressources*. Ces ressources peuvent être l'*énergie*, le *temps*, ou encore l'*espace* nécessaire pour arriver au résultat.

Si l'espace (par exemple pour de l'informatique embarqué) ou l'énergie (lire l'article « Numérique : le grand gâchis énergétique » https://lejournal.cnrs.fr/articles/numerique-le-grand-gachis-energetique) sont aussi des mesures fondamentales, nous nous intéresserons ici plus particulièrement au temps.

Pour mesurer ce temps, on pourrait s'armer d'un chronomètre et mesurer le temps d'exécution du programme, mais selon la machine, le langage et son compilateur, le système d'exploitation (etc) utilisé, la mesure pourrait beaucoup fluctuer. On va donc mesurer théoriquement ce temps sur les algorithmes.

a) Principe

⋆ On identifie et on note n la taille des données à traiter.

⋆ On choisit les opérations significatives.

⋆ On compte le nombre de ces opérations en fonction de n.

On appellera *complexité*, ou *coût*, le nombre obtenu.

Quand on travaille avec des données théoriques, on ne peut pas toujours déterminer exactement le nombre d'opérations (à cause des instructions conditionnelles). On peut alors se placer :

⋆ dans le meilleur cas (biaisé par des traitements à part de ces cas),

⋆ dans un cas moyen (très difficile à déterminer),

⋆ dans le pire cas (celui provoquant le plus d'opérations).

En première analyse on traitera toujours le pire cas (après l'avoir identifié).

b) Classe / coût

Soit un algorithme de complexité $c(n)$; et $f(n)$ une autre expression algébrique en n.

On dit que $c(n)$ est un « grand O de $f(n)$ » noté $c(n) = O(f(n))$
s'il existe un coefficient k tel que $c(n) < k \times f(n)$ pour n assez grand.

De manière générale, un algorithme de complexité polynômiale est toujours un $O(n^k)$.

Classe / coût

On appelle classe, l'écriture de la complexité sous la forme $O(f(n))$, pour certaines expressions simples de $f(n)$.

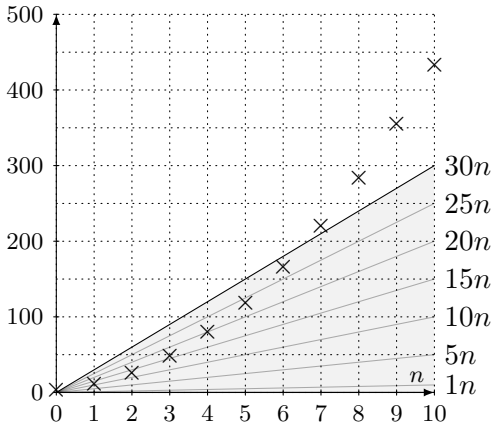
⋆ Coût constant : $O(1)$ ⋆ Coût linéaire : $O(n)$ ⋆ Coût quadratique : $O(n^2)$

Comme un coût constant est dominé par un coût linéaire, lui-même dominé par un coût quadratique, on note les inclusions suivantes des classes :

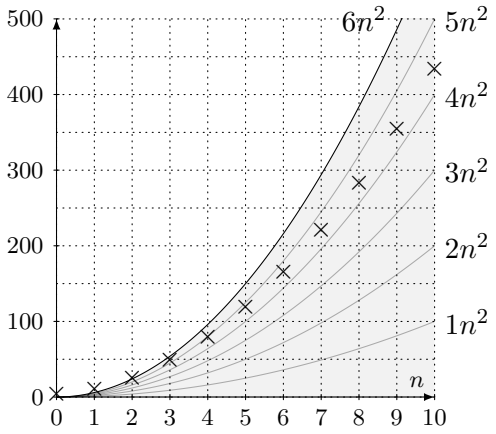
$$O(1) \subset O(n) \subset O(n^2)$$

Une « bonne » estimation de complexité doit donc donner la plus petite classe.

Exemple : $4n^2 + 3n + 4 = O(n^2)$



Le coût n'est pas linéaire.



Le coût est quadratique.

Si $3 < n$ alors $3n < n^2$ et $4 < 9 < n^2$ donc $4n^2 + 3n + 4 < 4n^2 + n^2 + n^2$

Il existe donc $k = 6$ tel que, pour n assez grand ($3 < n$), $4n^2 + 3n + 4 < 6n^2$

Pour aller plus loin :

$$O(1) \subset O(\log(n)) \subset O(\sqrt{n}) \subset O(n) \subset O(n\log(n)) \subset O(n^2) \dots$$

2 - Correction

Correction

La correction d'un algorithme correspond à deux critères :

- ⋆ **La correction partielle** vérifie que l'algorithme produit le bon résultat.
- ⋆ **La terminaison** vérifie qu'il produit un résultat en un temps fini.

La correction partielle d'un algorithme est fortement liée à la spécification de son rôle (le bon résultat n'est peut-être qu'une bonne valeur approchée...) et peut être inaccessible (on ne sait pas la démontrer pour l'algorithme de la suite de Syracuse...).

La terminaison peut ne pas être démontrable (si l'algorithme attend une entrée valide de l'utilisateur qui se trompe tout le temps...), ni même souhaitable (si l'algorithme surveille la température du cœur d'un réacteur nucléaire...).

a) Correction partielle

On démontre la correction partielle en suivant l'état des variables d'une instruction à l'autre.

```
1 fonction avant(x)
2 |  a ← x − 1
3 |  renvoyer a
```

ligne	x	a
1	x	////
2	x	$x-1$

La fonction renvoie bien l'entier précédent.

Les structures conditionnelles engendrent des branches, (on raisonne par disjonction des cas).

```
1 fonction pairAvant(x)
2 |  si x est pair alors
3 |  |  a ← x − 2
4 |  sinon
5 |  |  a ← x − 1
6 |  renvoyer a
```

ligne	x	a	test
1	x	////	
2	$2x'$	////	x pair
3	$2x'$	$2(x'-1)$	

ou

ligne	x	a	test
2	$2x'+1$	////	x impair
5	$2x'+1$	$2x'$	

La fonction renvoie bien l'entier pair précédant x.

Pour les répétitives, (on raisonne par récurrence) :

⋆ on détermine une relation entre les variables intervenant dans la répétitive,

⋆ qui soit vraie avant,

⋆ et qui reste vraie d'une itération à l'autre.

Elle sera alors automatiquement encore vraie après la répétitive.
On dit qu'on a déterminé un **invariant** de la répétitive.
Un invariant « bien choisi » exprime à la sortie de la répétitive la correction partielle.

```
1 fonction multipleAvant(x, k)
2 |  a ← x − 1
3 |  tant que a n'est pas multiple de k faire
4 |  |  a ← a − 1
5 |  renvoyer a
```

Invariant : « il n'y a pas de multiple de k strictement entre a et x »
Ligne 2, il n'y a rien entre a et x donc c'est vrai.
Si c'est vrai ligne 3 (avant une itération),
et que l'on rentre dans la boucle,
alors a lui-même n'est pas multiple de k donc il n'y en a pas entre $a-1$ et x.
Dans ce cas, la ligne 4 met effectivement à jour notre invariant (vrai après l'itération).
Finalement, en sortie, ligne 5, a est bien un multiple de k et notre invariant nous permet d'affirmer que c'est celui qui précède x.

b) Terminaison

Tout algorithme sans appel de fonction ni répétitive termine.
Toute répétitive `pour` itère un nombre fini de fois donc termine.
On cherche donc à prouver la terminaison de répétitives `tant que`.
Pour cela, on cherche à identifier un **variant**, c'est à dire :

⋆ une expression à **valeurs entières** dépendant des variables de la répétitive,

⋆ dont les valeurs sont **strictement décroissantes** au fil des itérations,

⋆ tout en restant **supérieures ou égales à 0**.

Une telle expression finit toujours par valoir 0 et donc ne peut plus être strictement décroissante, ce qui prouve que la boucle se termine.
Pour l'exemple précédent, (avec la précondition $0 < k \leqslant x$) un variant possible est a qui est bien strictement décroissant (de 1 en 1) et positif ou nul (vrai ligne 2, et 0 est multiple de tout nombre k).
Donc l'algorithme termine.