Rappels & Complément
Algorithmique, Python

MP⋆ - Lycée Carnot - Dijon

11 et 18 septembre 2017

Preuve d'algorithme

- **Une preuve de terminaison** pour les boucles conditionnelles (`while`). Cela se fait via un *variant de boucle* : en général une suite entière minorée strictement décroissante.
- **La détermination d'*invariants de boucle*** : propriétés décrivant l'état du système (ou une partie de celui-ci) à un instant donné d'une boucle, qui se prouve par récurrence.
- **La correction de l'algorithme** : on peut alors connaître l'état du système en fin d'exécution et donc vérifier que la quantité renvoyée est la bonne, ou que le traitement effectué est le bon.

Preuve d'algorithme

- **Une preuve de terminaison** pour les boucles conditionnelles (`while`). Cela se fait via un *variant de boucle* : en général une suite entière minorée strictement décroissante.
- **La détermination d'*invariants de boucle*** : propriétés décrivant l'état du système (ou une partie de celui-ci) à un instant donné d'une boucle, qui se prouve par récurrence.
- **La correction de l'algorithme** : on peut alors connaître l'état du système en fin d'exécution et donc vérifier que la quantité renvoyée est la bonne, ou que le traitement effectué est le bon.

Preuve d'algorithme

- **Une preuve de terminaison** pour les boucles conditionnelles (`while`). Cela se fait via un *variant de boucle* : en général une suite entière minorée strictement décroissante.

- **La détermination d'*invariants de boucle*** : propriétés décrivant l'état du système (ou une partie de celui-ci) à un instant donné d'une boucle, qui se prouve par récurrence.

- **La correction de l'algorithme** : on peut alors connaître l'état du système en fin d'exécution et donc vérifier que la quantité renvoyée est la bonne, ou que le traitement effectué est le bon.

Preuve d'algorithme

- **Une preuve de terminaison** pour les boucles conditionnelles (`while`). Cela se fait via un *variant de boucle* : en général une suite entière minorée strictement décroissante.

- **La détermination d'*invariants de boucle*** : propriétés décrivant l'état du système (ou une partie de celui-ci) à un instant donné d'une boucle, qui se prouve par récurrence.

- **La correction de l'algorithme** : on peut alors connaître l'état du système en fin d'exécution et donc vérifier que la quantité renvoyée est la bonne, ou que le traitement effectué est le bon.

Exemple : puissance naïve

- Écrire une fonction `puiss(x, n)` renvoyant x^n et prouver sa correction.

Exemple : exponentiation rapide

- Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$: il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

L'idée est alors de décomposer n en base 2 : si n s'écrit

$$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

$$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.
Par exemple,

- $19 = 1 + 2 + 2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires),
- $39 = 1 + 2 + 2^2 + 2^5$ et

$$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

(8 multiplications nécessaires).

Écrire une fonction `exprap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

Exemple : exponentiation rapide

- Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$: il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

  L'idée est alors de décomposer n en base 2 : si n s'écrit

  $$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

  avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

  $$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

  le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.

  Par exemple,

  - $19 = 1 + 2 + 2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires),
  - $39 = 1 + 2 + 2^2 + 2^5$ et

    $$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

    (8 multiplications nécessaires).

  Écrire une fonction `exprap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

Exemple : exponentiation rapide

- Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$:

  il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

  L'idée est alors de décomposer n en base 2 : si n s'écrit

  $$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

  avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

  $$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

  le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.
  Par exemple,

  - $19 = 1 + 2 + 2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires),
  - $39 = 1 + 2 + 2^2 + 2^5$ et

    $$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

    (8 multiplications nécessaires).

Écrire une fonction `exprap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

Exemple : exponentiation rapide

- Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$:

  il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

  L'idée est alors de décomposer n en base 2 : si n s'écrit

  $$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

  avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

  $$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

  le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.
  Par exemple,

  - $19 = 1 + 2 + 2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires),
  - $39 = 1 + 2 + 2^2 + 2^5$ et

    $$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

    (8 multiplications nécessaires).

Écrire une fonction `exprap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

Exemple : exponentiation rapide

- Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$:

  il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

  L'idée est alors de décomposer n en base 2 : si n s'écrit

  $$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

  avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

  $$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

  le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.
  Par exemple,

  - $19 = 1 + 2 + 2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires),
  - $39 = 1 + 2 + 2^2 + 2^5$ et

    $$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

    (8 multiplications nécessaires).

  Écrire une fonction `exprap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

Exemple : exponentiation rapide

```
def exprap(x, n):
    """calcul de x puissance n par expon. rapide"""
    puis = 1
    xpuis2 = x
    m = n
    while m != 0:
        if m % 2 == 1:
            puis *= xpuis2
        xpuis2 *= xpuis2
        m //= 2
    return puis
```

- **Terminaison** : `m` strictement décroissant minoré.
- **Invariant de sortie** : à la fin du i^e tour de boucle, `m` contient $\left\lfloor \frac{n}{2^i} \right\rfloor$, `xpuis2` contient x^{2^i}, et `puis` contient

$$x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^{i-1}}\right)^{b_{i-1}} = x^{b_0 + b_1 2 + \cdots + b_{i-1} 2^{i-1}},$$

où $b_{i-1}, \ldots, b_0$ sont les i derniers chiffres de l'écriture en base 2 de n.

- **Correction** : dernière étape : lorsque $m = 0$, on a bien obtenu tous les chiffres de l'écriture en base 2 de n et on renvoie bien x^n.

Exemple : exponentiation rapide

```
def exprap(x, n):
    """calcul de x puissance n par expon. rapide"""
    puis = 1
    xpuis2 = x
    m = n
    while m != 0:
        if m % 2 == 1:
            puis *= xpuis2
        xpuis2 *= xpuis2
        m //= 2
    return puis
```

- **Terminaison** : `m` strictement décroissant minoré.
- **Invariant de sortie** : à la fin du i^e tour de boucle, `m` contient $\left\lfloor \frac{n}{2^i} \right\rfloor$, `xpuis2` contient x^{2^i}, et `puis` contient

$$x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^{i-1}}\right)^{b_{i-1}} = x^{b_0 + b_1 2 + \cdots + b_{i-1} 2^{i-1}}$$

  où $b_{i-1}, \ldots, b_0$ sont les i derniers chiffres de l'écriture en base 2 de n.
- **Correction** : dernière étape : lorsque $m = 0$, on a bien obtenu tous les chiffres de l'écriture en base 2 de n et on renvoie bien x^n.

Exemple : exponentiation rapide

```
def exprap(x, n):
    """calcul de x puissance n par expon. rapide"""
    puis = 1
    xpuis2 = x
    m = n
    while m != 0:
        if m % 2 == 1:
            puis *= xpuis2
        xpuis2 *= xpuis2
        m //= 2
    return puis
```

- **Terminaison** : `m` strictement décroissant minoré.
- **Invariant de sortie** : à la fin du i^{e} tour de boucle, `m` contient $\left\lfloor \frac{n}{2^i} \right\rfloor$, `xpuis2` contient x^{2^i}, et `puis` contient

$$x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^{i-1}}\right)^{b_{i-1}} = x^{b_0 + b_1 2 + \cdots + b_{i-1} 2^{i-1}}$$

où $b_{i-1}, \ldots, b_0$ sont les i derniers chiffres de l'écriture en base 2 de n.

- **Correction** : dernière étape : lorsque $m = 0$, on a bien obtenu tous les chiffres de l'écriture en base 2 de n et on renvoie bien x^n.

Exemple : exponentiation rapide

```
def exprap(x, n):
    """calcul de x puissance n par expon. rapide"""
    puis = 1
    xpuis2 = x
    m = n
    while m != 0:
        if m % 2 == 1:
            puis *= xpuis2
        xpuis2 *= xpuis2
        m //= 2
    return puis
```

- **Terminaison** : `m` strictement décroissant minoré.
- **Invariant de sortie** : à la fin du i^{e} tour de boucle, `m` contient $\left\lfloor \frac{n}{2^i} \right\rfloor$, `xpuis2` contient x^{2^i}, et `puis` contient

$$x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^{i-1}}\right)^{b_{i-1}} = x^{b_0 + b_1 2 + \cdots + b_{i-1} 2^{i-1}}$$

  où $b_{i-1}, \ldots, b_0$ sont les i derniers chiffres de l'écriture en base 2 de n.
- **Correction** : dernière étape : lorsque $m = 0$, on a bien obtenu tous les chiffres de l'écriture en base 2 de n et on renvoie bien x^n.

Complexité

- Temporelle
- Spatiale

- Empirique
- Théorique

Complexité

- Temporelle
- Spatiale

- Empirique
- Théorique

Complexité

- Temporelle
- Spatiale

- Empirique
- Théorique

Complexité

- Temporelle
- Spatiale

- Empirique
- Théorique

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Opérations élémentaires

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

Notations

Notations O, Ω et Θ

On suppose que $f(n) \geqslant 0$ et $g(n) \geqslant 0$ pour tout entier n.

- On note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que

$$\forall\, n \geqslant n_0, \quad f(n) \leqslant c \times g(n)$$

  Cela signifie que f croît au plus aussi vite que g.

- On note $f = \Omega(g)$ lorsqu'il existe un rang n_0 et une constante $d > 0$ tels que

$$\forall\, n \geqslant n_0, \quad f(n) \geqslant d \times g(n)$$

  Cela signifie que f croît au moins aussi vite que g.

- On note $f = \Theta(g)$ lorsque $f = O(g)$ et $f = \Omega(g)$ c'est-à-dire lorsqu'il existe un rang n_0 et des constantes $c, d > 0$ tels que

$$\forall\, n \geqslant n_0, \quad c \times g(n) \leqslant f(n) \leqslant d \times g(n)$$

  Cela signifie que f et g sont du même ordre.

Notations

Notations O, Ω et Θ

On suppose que $f(n) \geqslant 0$ et $g(n) \geqslant 0$ pour tout entier n.

- On note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que
$$\forall\, n \geqslant n_0, \quad f(n) \leqslant c \times g(n)$$
Cela signifie que f croît au plus aussi vite que g.

- On note $f = \Omega(g)$ lorsqu'il existe un rang n_0 et une constante $d > 0$ tels que
$$\forall\, n \geqslant n_0, \quad f(n) \geqslant d \times g(n)$$
Cela signifie que f croît au moins aussi vite que g.

- On note $f = \Theta(g)$ lorsque $f = O(g)$ et $f = \Omega(g)$ c'est-à-dire lorsqu'il existe un rang n_0 et des constantes $c, d > 0$ tels que
$$\forall\, n \geqslant n_0, \quad c \times g(n) \leqslant f(n) \leqslant d \times g(n)$$
Cela signifie que f et g sont du même ordre.

Notations

Notations O, Ω et Θ

On suppose que $f(n) \geqslant 0$ et $g(n) \geqslant 0$ pour tout entier n.

- On note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que
$$\forall\, n \geqslant n_0, \quad f(n) \leqslant c \times g(n)$$
Cela signifie que f croît au plus aussi vite que g.

- On note $f = \Omega(g)$ lorsqu'il existe un rang n_0 et une constante $d > 0$ tels que
$$\forall\, n \geqslant n_0, \quad f(n) \geqslant d \times g(n)$$
Cela signifie que f croît au moins aussi vite que g.

- On note $f = \Theta(g)$ lorsque $f = O(g)$ et $f = \Omega(g)$ c'est-à-dire lorsqu'il existe un rang n_0 et des constantes $c, d > 0$ tels que
$$\forall\, n \geqslant n_0, \quad c \times g(n) \leqslant f(n) \leqslant d \times g(n)$$
Cela signifie que f et g sont du même ordre.

Comparatif

Chaque valeur est multipliée par 10^{-6} s comme majorant du temps d'exécution d'une opération élémentaire.

	n	10	50	100	500	1000
log.	$\ln n$	2 μs	4 μs	4,6 μs	6 μs	7 μs
	$\sqrt{n}$	3 μs	7 μs	10 μs	20 μs	30 μs
lin.	n	10 μs	50 μs	100 μs	0,5 ms	1 ms
semi-lin.	$n \ln n$	20 μs	200 μs	500 μs	3 ms	7 ms
quad.	n^2	100 μs	2,5 ms	10 ms	0,25 s	1s
polyn.	n^3	1 ms	0,1 s	1 s	2 min	16 min
exp.	2^n	1 ms	36 a	$4 \cdot 10^6$ a	10^{137} a	$3 \cdot 10^{287}$ a
	$n!$	4 s	10^{51} a	$3 \cdot 10^{144}$ a	$3 \cdot 10^{1121}$ a	$3 \cdot 10^{2554}$ a

Ordres de grandeurs :

- **Âge de l'univers** : $13,8 \times 10^9$ années
- **Nombre d'atomes dans l'univers** : 10^{80}.

Comparatif

Échelle de croissances comparées :

$$\ln n \ll \sqrt{n} \ll n \ll n \ln n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

n maximum pour un temps d'exécution d'une seconde :

$\ln n$	$\sqrt{n}$	n	$n \ln n$	n^2	n^3	2^n	$n!$
$\simeq 10^{400\ 000}$	10^{12}	10^6	87848	10^3	100	20	10

On retiendra qu'à partir de n^3, le temps d'exécution n'est pas raisonnable.

Comparatif

Échelle de croissances comparées :

$$\ln n \ll \sqrt{n} \ll n \ll n \ln n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

n maximum pour un temps d'exécution d'une seconde :

$\ln n$	$\sqrt{n}$	n	$n \ln n$	n^2	n^3	2^n	$n!$
$\simeq 10^{400\ 000}$	10^{12}	10^6	87848	10^3	100	20	10

On retiendra qu'à partir de n^3, le temps d'exécution n'est pas raisonnable.

Comparatif

Échelle de croissances comparées :

$$\ln n \ll \sqrt{n} \ll n \ll n \ln n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

n maximum pour un temps d'exécution d'une seconde :

$\ln n$	$\sqrt{n}$	n	$n \ln n$	n^2	n^3	2^n	$n!$
$\simeq 10^{400\ 000}$	10^{12}	10^6	87848	10^3	100	20	10

On retiendra qu'à partir de n^3, le temps d'exécution n'est pas raisonnable.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Exercice

- `puiss(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(n) = \Theta(n)$ (linéaire)

  Complexité spatiale : $\Theta(1)$

- `exporap(x,n)`

  Complexité temporelle : $\Theta(1) + \Theta(k) = \Theta(\log n)$ où k nombre de chiffre de l'écriture binaire de n : logarithmique.

  Complexité spatiale : $\Theta(1)$

Mais... L'entier n est codé en binaire dans la machine... Raisonnable de calculer la complexité en fonction de k ?

`puiss` devient exponentiel et `exporap` linéaire.

Structures de données

Il convient dans la pratique de l'algorithmique de choisir de représenter ses données en utilisant une structure appropriée, dépendant de ce qu l'on a besoin d'en faire et de ce que cela coûte.

On rencontre deux types de structures : celles pour lesquelles un emplacement dans la mémoire (RAM) aura été alloué dès la création et ne changera pas, dites **statiques** et celles dont l'allocation mémoire et la taille peuvent varier au cours de l'exécution d'un algorithme, dites **dynamiques**. Lorsque l'on peut modifier les éléments de cette structure, elle dite **mutable**.

Par exemple, en python, les tuples sont statiques, non mutables, et les listes sont dynamiques et mutables.

Structures de données

Il convient dans la pratique de l'algorithmique de choisir de représenter ses données en utilisant une structure appropriée, dépendant de ce qu l'on a besoin d'en faire et de ce que cela coûte.

On rencontre deux types de structures : celles pour lesquelles un emplacement dans la mémoire (RAM) aura été alloué dès la création et ne changera pas, dites **statiques** et celles dont l'allocation mémoire et la taille peuvent varier au cours de l'exécution d'un algorithme, dites **dynamiques**. Lorsque l'on peut modifier les éléments de cette structure, elle dite **mutable**.

Par exemple, en python, les tuples sont statiques, non mutables, et les listes sont dynamiques et mutables.

Structures de données

Il convient dans la pratique de l'algorithmique de choisir de représenter ses données en utilisant une structure appropriée, dépendant de ce qu l'on a besoin d'en faire et de ce que cela coûte.

On rencontre deux types de structures : celles pour lesquelles un emplacement dans la mémoire (RAM) aura été alloué dès la création et ne changera pas, dites **statiques** et celles dont l'allocation mémoire et la taille peuvent varier au cours de l'exécution d'un algorithme, dites **dynamiques**. Lorsque l'on peut modifier les éléments de cette structure, elle dite **mutable**.

Par exemple, en python, les tuples sont statiques, non mutables, et les listes sont dynamiques et mutables.

Tableaux

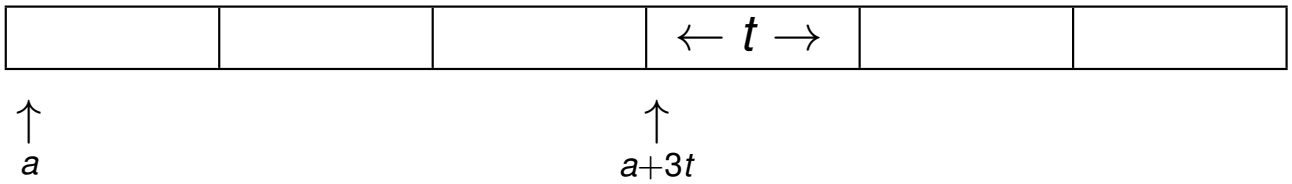
Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.

- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Tableaux

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.

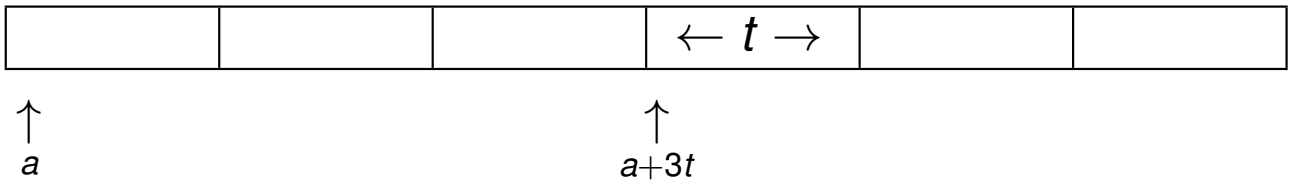


- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Tableaux

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.

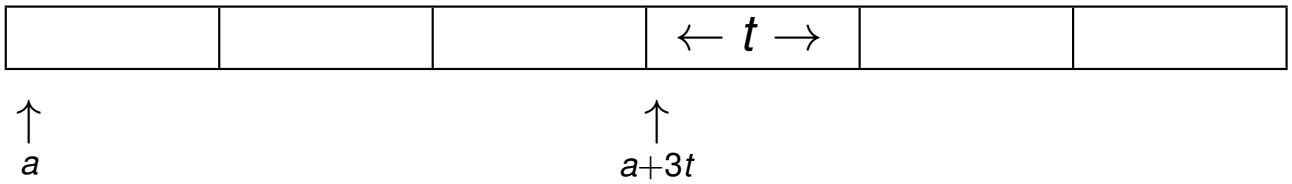


- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Tableaux

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.

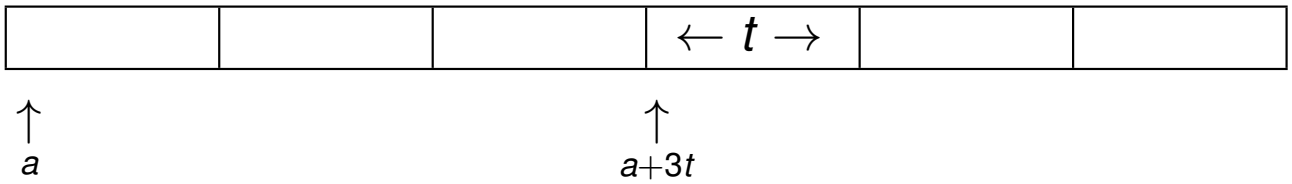


- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Tableaux

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.

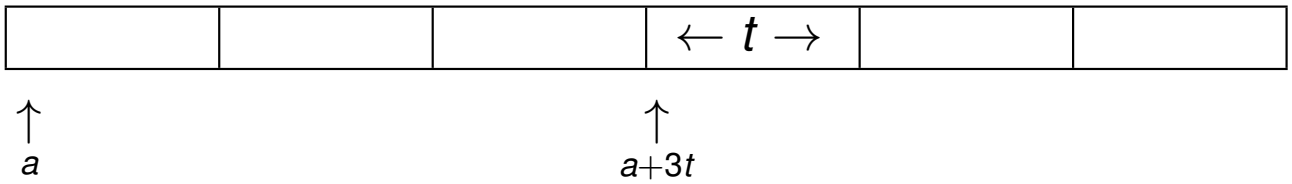


- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Tableaux

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a + kt$.



- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.

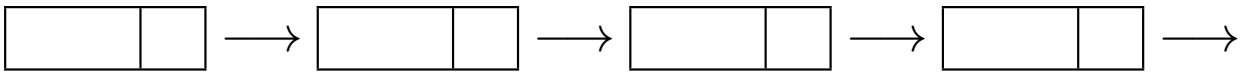
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^e élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



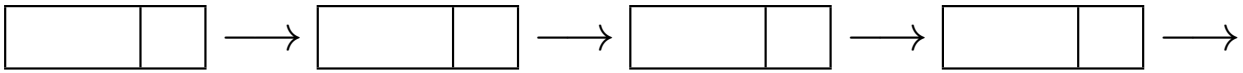
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^e élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



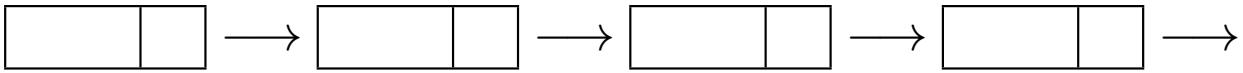
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



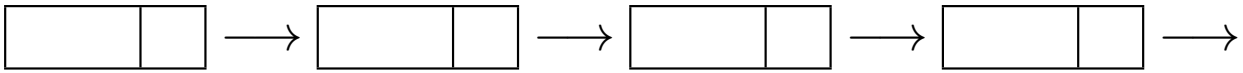
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



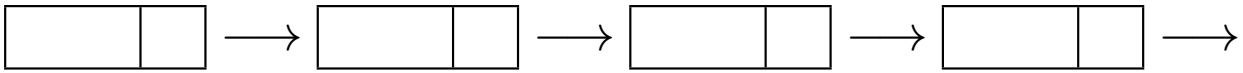
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



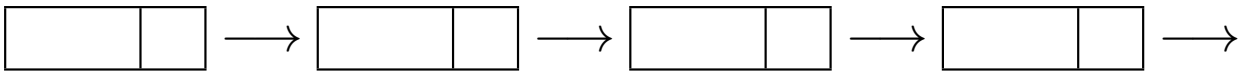
- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Listes

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste.

Les listes Python

Python permet de travailler avec des objets de type `list` qui est une structure de données dynamique mutable : ce n'est cependant ni un tableau ni une liste au sens informatique du terme.

Le principe est le suivant : lorsque l'on crée une liste ou lorsque l'on ajoute des éléments à une liste, on réserve une place plus grande en mémoire. Plus exactement, la liste est « allongée » lorsque l'on passe certains palier qui sont :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, \ldots$$

D'où viennent ces valeurs ?

Pour le savoir, il faut fouiller de le code source de CPython, consultable librement :

Les listes Python

Python permet de travailler avec des objets de type `list` qui est une structure de données dynamique mutable : ce n'est cependant ni un tableau ni une liste au sens informatique du terme.

Le principe est le suivant : lorsque l'on crée une liste ou lorsque l'on ajoute des éléments à une liste, on réserve une place plus grande en mémoire. Plus exactement, la liste est « allongée » lorsque l'on passe certains palier qui sont :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, \ldots$$

D'où viennent ces valeurs ?

Pour le savoir, il faut fouiller de le code source de CPython, consultable librement :

Les listes Python

Python permet de travailler avec des objets de type `list` qui est une structure de données dynamique mutable : ce n'est cependant ni un tableau ni une liste au sens informatique du terme.

Le principe est le suivant : lorsque l'on crée une liste ou lorsque l'on ajoute des éléments à une liste, on réserve une place plus grande en mémoire. Plus exactement, la liste est « allongée » lorsque l'on passe certains palier qui sont :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, \ldots$$

D'où viennent ces valeurs ?

Pour le savoir, il faut fouiller de le code source de CPython, consultable librement :

Les listes Python

Python permet de travailler avec des objets de type `list` qui est une structure de données dynamique mutable : ce n'est cependant ni un tableau ni une liste au sens informatique du terme.

Le principe est le suivant : lorsque l'on crée une liste ou lorsque l'on ajoute des éléments à une liste, on réserve une place plus grande en mémoire. Plus exactement, la liste est « allongée » lorsque l'on passe certains palier qui sont :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, \ldots$$

D'où viennent ces valeurs ?

Pour le savoir, il faut fouiller de le code source de CPython, consultable librement :

Les listes Python

```
/* This over-allocates proportional to the list size, making room
* for additional growth.  The over-allocation is mild, but is
* enough to give linear-time amortized behavior over a long
* sequence of appends() in the presence of a poorly-performing
* system realloc().
* The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
*/
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}
```

Le `>> 3` signifie un décalage de 3 bits vers la droite, donc une division entière par 2^3 et `newsize < 9 ? 3 : 6` vaut 3 si `newsize < 9` et 6 sinon.

Autrement dit, si n est la taille de la liste, la place allouée dans la mémoire sera

$$n + \left\lfloor \frac{n}{8} \right\rfloor + \begin{cases} 3 & \text{si } n < 9 \\ 6 & \text{sinon} \end{cases}$$

Les listes Python

```
/* This over-allocates proportional to the list size, making room
* for additional growth.  The over-allocation is mild, but is
* enough to give linear-time amortized behavior over a long
* sequence of appends() in the presence of a poorly-performing
* system realloc().
* The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
*/
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}
```

Le `>> 3` signifie un décalage de 3 bits vers la droite, donc une division entière par 2^3 et `newsize < 9 ? 3 : 6` vaut 3 si `newsize < 9` et 6 sinon.

Autrement dit, si n est la taille de la liste, la place allouée dans la mémoire sera

$$n + \left\lfloor \frac{n}{8} \right\rfloor + \begin{cases} 3 & \text{si } n < 9 \\ 6 & \text{sinon} \end{cases}$$

Les listes Python

```
/* This over-allocates proportional to the list size, making room
* for additional growth.  The over-allocation is mild, but is
* enough to give linear-time amortized behavior over a long
* sequence of appends() in the presence of a poorly-performing
* system realloc().
* The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
*/
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}
```

Le `>> 3` signifie un décalage de 3 bits vers la droite, donc une division entière par 2^3 et `newsize < 9 ? 3 : 6` vaut 3 si `newsize < 9` et 6 sinon.

Autrement dit, si n est la taille de la liste, la place allouée dans la mémoire sera

$$n + \left\lfloor \frac{n}{8} \right\rfloor + \begin{cases} 3 & \text{si } n < 9 \\ 6 & \text{sinon} \end{cases}$$

Gestion de l'allocation mémoire lors de l'utilisation de listes python

Lu sur `http://www.laurentluce.com/posts/python-list-implementation/`

...

```
>>> L = [1,2,3]

>>> L[2]
3

>>> L[1] = 0 # Mutable

>>> L
[1, 0, 3]
```

```
>>> L[1:3] # Slicing
[0, 3]

>>> len(L)
3

>>> L + [1,2,3] # Concaténation
[1, 0, 3, 1, 2, 3]

>>> 3 in L # Appartenance
True
```

```
>>> L = [1,2,3]

>>> L[2]
3

>>> L[1] = 0 # Mutable

>>> L
[1, 0, 3]
```

```
>>> L[1:3] # Slicing
[0, 3]

>>> len(L)
3

>>> L + [1,2,3] # Concaténation
[1, 0, 3, 1, 2, 3]

>>> 3 in L # Appartenance
True
```

```
>>> tuple(L) # Casting
(1, 0, 3)

>>> list(t)
[1, 2.65, True, 'azerty']

>>> str(L)
'[1, 0, 3]'

>>> list("Bonjour")
['B', 'o', 'n', 'j', 'o', 'u', 'r']
```

```
>>> L.append(5) # Ajouter un élément à la fin

>>> L
[1, 0, 3, 5]

>>> a = L.pop() # Récupérer le dernier élément

>>> a, L
(5, [1, 0, 3])
```

```
>>> tuple(L) # Casting
(1, 0, 3)

>>> list(t)
[1, 2.65, True, 'azerty']

>>> str(L)
'[1, 0, 3]'

>>> list("Bonjour")
['B', 'o', 'n', 'j', 'o', 'u', 'r']
```

```
>>> L.append(5) # Ajouter un élément à la fin

>>> L
[1, 0, 3, 5]

>>> a = L.pop() # Récupérer le dernier élément

>>> a, L
(5, [1, 0, 3])
```

```
>>> L2 = L

>>> id(L), id(L2)
(140352491149648, 140352491149648)

>>> id(L) == id(L2)
True
```

```
>>> L[1] = 12

>>> L
[1, 12, 0]

>>> L2
[1, 12, 0]
```

```
>>> L2 = L

>>> id(L), id(L2)
(140352491149648, 140352491149648)

>>> id(L) == id(L2)
True
```

```
>>> L[1] = 12

>>> L
[1, 12, 0]

>>> L2
[1, 12, 0]
```

```
>>> L3 = L.copy() # ou bien L[:] ou encore list(L)

>>> id(L) == id(L3)
False

>>> L[1] = 36

>>> L
[1, 36, 0]

>>> L3
[1, 12, 0]
```

```
>>> LL = [[1], [2], [3]]

>>> LL2 = LL.copy() # copie superficielle

>>> LL2
[[1], [2], [3]]
```

```
>>> LL2[0][0] = 0

>>> LL2, LL
([[0], [2], [3]], [[0], [2], [3]])
```

```
>>> LL2[0] = [1]

>>> LL, LL2
([[0], [2], [3]], [[1], [2], [3]])
```

```
>>> LL = [[1], [2], [3]]

>>> LL2 = LL.copy() # copie superficielle

>>> LL2
[[1], [2], [3]]
```

```
>>> LL2[0][0] = 0

>>> LL2, LL
([[0], [2], [3]], [[0], [2], [3]])
```

```
>>> LL2[0] = [1]

>>> LL, LL2
([[0], [2], [3]], [[1], [2], [3]])
```

```
>>> LL = [[1], [2], [3]]

>>> LL2 = LL.copy() # copie superficielle

>>> LL2
[[1], [2], [3]]
```

```
>>> LL2[0][0] = 0

>>> LL2, LL
([[0], [2], [3]], [[0], [2], [3]])
```

```
>>> LL2[0] = [1]

>>> LL, LL2
([[0], [2], [3]], [[1], [2], [3]])
```

- Certaines opérations peuvent se révéler dangereuses : on évitera a tout pris d'écrire, par exemple `L = [a] * n` mais on préférera `L = [a for _ in range(n)]`. Si `a` n'est pas mutable, cela ne pose pas de problème, mais s'il l'est, c'est très embêtant ! Le même genre de problème peut se présenter lorsque l'on utilise `+=` avec des listes de listes.

- Notons que le module `copy` fournit une fonction `deepcopy` permettant de faire des copies profondes de listes.

- On rappelle également que les listes sont passées par référence lorsqu'elles sont arguments de fonctions : cela signifie que la liste peut être modifiée à l'intérieur d'une fonction dont elle est l'un des arguments.

- Certaines opérations peuvent se révéler dangereuses : on évitera a tout pris d'écrire, par exemple `L = [a] * n` mais on préférera `L = [a for _ in range(n)]`. Si `a` n'est pas mutable, cela ne pose pas de problème, mais s'il l'est, c'est très embêtant ! Le même genre de problème peut se présenter lorsque l'on utilise `+=` avec des listes de listes.

- Notons que le module `copy` fournit une fonction `deepcopy` permettant de faire des copies profondes de listes.

- On rappelle également que les listes sont passées par référence lorsqu'elles sont arguments de fonctions : cela signifie que la liste peut être modifiée à l'intérieur d'une fonction dont elle est l'un des arguments.

- Certaines opérations peuvent se révéler dangereuses : on évitera a tout pris d'écrire, par exemple `L = [a] * n` mais on préférera `L = [a for _ in range(n)]`. Si `a` n'est pas mutable, cela ne pose pas de problème, mais s'il l'est, c'est très embêtant ! Le même genre de problème peut se présenter lorsque l'on utilise `+=` avec des listes de listes.

- Notons que le module `copy` fournit une fonction `deepcopy` permettant de faire des copies profondes de listes.

- On rappelle également que les listes sont passées par référence lorsqu'elles sont arguments de fonctions : cela signifie que la liste peut être modifiée à l'intérieur d'une fonction dont elle est l'un des arguments.

Lu sur `https://wiki.python.org/moin/TimeComplexity` :

Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter.

Operation	Average Case	Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

(1) = These operations rely on the "Amortized" part of "Amortized Worst Case". Individual actions may take surprisingly long, depending on the history of the container.

Utilisation de with

```
with open(nomFichier,'r') as file:
    <...>
    <instructions utilisant file>
    <...>
```

Cela permet d'optimiser la gestion de la mémoire en fermant le fichier automatiquement dès que le traitement est terminé.