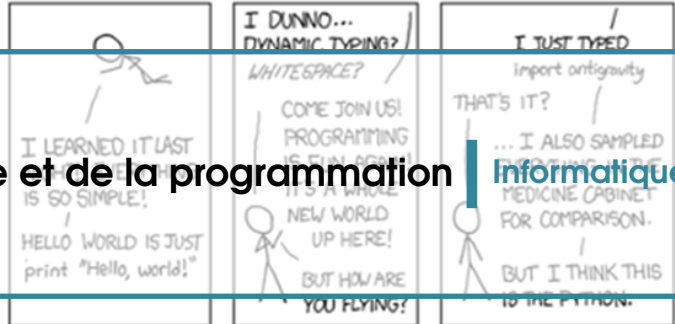
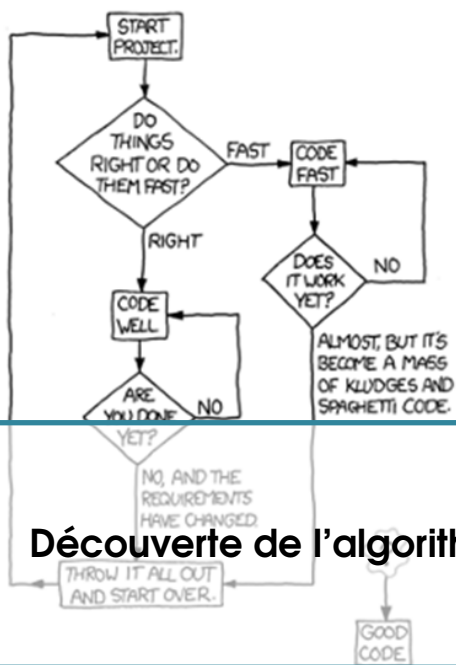


## HOW TO WRITE GOOD CODE:



## Découverte de l'algorithmique et de la programmation Informatique

## Chapitre 7

## Complexité algorithmique

Savoirs et compétences :

- Complexité.

1	Mise en évidence du problème	2
2	Complexité des algorithmes	2
3	Profiling des algorithmes	6

## 1 Mise en évidence du problème

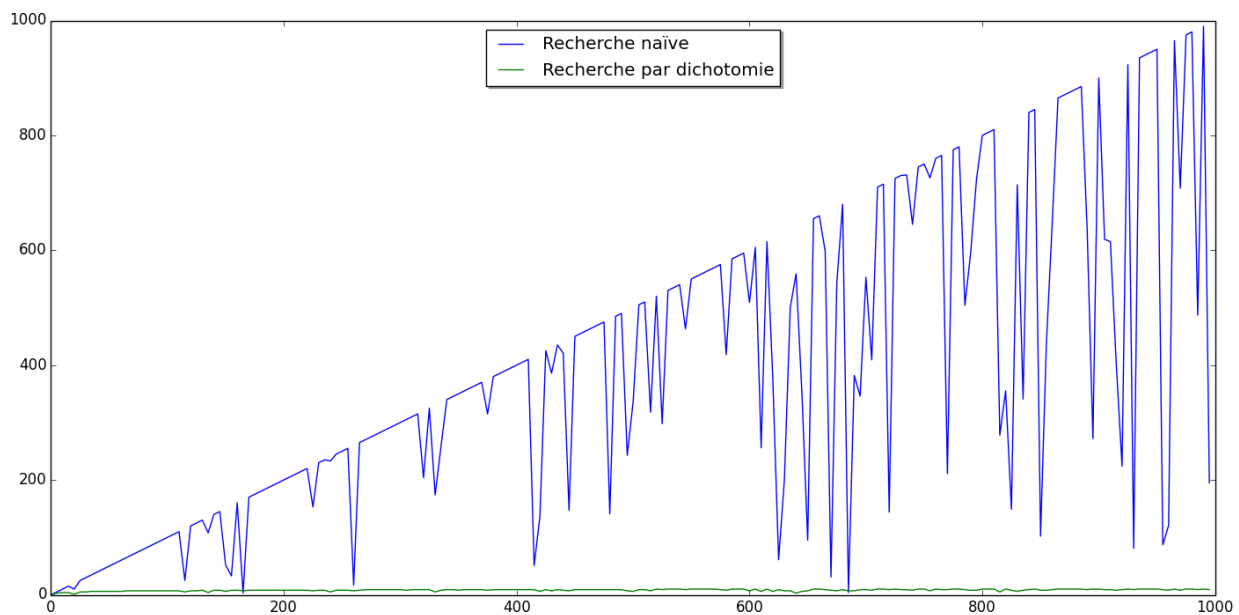
Prenons l'exemple de la recherche d'un élément dans une liste :

```
def is_number_in_list(nb, tab):  
    """Renvoie True si le nombre nb est dans ✓  
        la liste de nombres tab  
    Entrées :  
        * nb, int -- nombre entier  
        * tab, list -- liste de nombres entiers  
    """  
    for i in range(len(tab)):  
        if tab[i]==nb:  
            return True  
    return False
```

■ **Exemple** À partir de l'algorithme précédent, évaluer :

- le nombre de tour de boucles dans le pire des cas;
- le nombre de tour de boucles dans le meilleur des cas.

```
def is_number_in_list_dicho(nb, tab):  
    """  
    Recherche d'un nombre par dichotomie dans ✓  
    un tableau trié.  
    Renvoie l'index si le nombre nb est dans ✓  
    la liste de nombres tab.  
    Renvoie None sinon.  
    Entrées :  
        * nb, int -- nombre entier  
        * tab, list -- liste de nombres entiers ✓  
        triés  
    """  
    g, d = 0, len(tab)-1  
    while g <= d:  
        m = (g + d) // 2  
        if tab[m] == nb:  
            return m  
        if tab[m] < nb:  
            g = m+1  
        else:  
            d = m-1  
    return None
```



Évolution du nombre d'opérations pour rechercher un nombre dans une liste de 1 à 1000 nombres

## 2 Complexité des algorithmes

### 2.1 Présentation

Il existe souvent plusieurs façons de programmer un algorithme. Si le nombre d'opérations à effectuer est peu important et les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu. En revanche, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution et l'occupation mémoire.

#### Définition Complexité en temps

La complexité en temps donne le nombre d'opérations effectuées lors de l'exécution d'un programme. On appelle  $C_o$  le coût en temps d'une opération  $o$ .

#### Définition Complexité en mémoire (ou en espace)

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

R

On distingue la complexité dans le pire des cas, la complexité dans le meilleur des cas, ou la complexité en moyenne. En effet, pour un même algorithme, suivant les données à manipuler, le résultat sera déterminé plus ou moins rapidement.

Généralement, on s'intéresse au cas le plus défavorable à savoir, la complexité dans le pire des cas.

**Définition Complexité algorithmique – Notation  $\mathcal{O}$**  [Bournez] Soient  $f$  et  $g$  deux fonctions  $f, g : \mathbb{N} \rightarrow \mathbb{R}_*^+$ . On note  $f(n) = \mathcal{O}(g(n))$  lorsqu'il existe  $c \in \mathbb{R}^+$  et  $n_0 \in \mathbb{N}$  tels que pour tout  $n \geq n_0$ ,

$$f(n) \leq c \cdot g(n)$$

Intuitivement, cela signifie que  $f$  est inférieur à  $g$  à une constante multiplicative près pour les données suffisamment grandes.

■ **Exemple** Par ordre de complexité croissante on a :

- $\mathcal{O}(1)$  : algorithme s'exécutant en temps constant, quelle que soit la taille des données;
- $\mathcal{O}(\log(n))$  : algorithme rapide (complexité logarithmique) (Exemple : recherche par dichotomie dans un tableau trié);
- $\mathcal{O}(n)$  : algorithme linéaire;
- $\mathcal{O}(n \cdot \log(n))$  : algorithme quasi-linéaire;
- $\mathcal{O}(n^2)$  : complexité quadratique;
- $\mathcal{O}(n^3)$  : complexité cubique;
- $\mathcal{O}(2^n)$  : complexité exponentielle.

## 2.2 Coût temporel d'un algorithme et d'une opération

**Résultat** On considère que le coût élémentaire  $C_e$  correspond au coût d'une affectation, d'une comparaison ou de l'évaluation d'une opération arithmétique.

■ **Exemple**

```
def foo() :
    a=20
    a<=100
    a+a
```

Chacune de ces 3 opérations expressions ont le même coût temporel  $C_e$ . On a donc  $C_{foo}(n) = 3C_e$ .

`foo` prendra (a priori) toujours le même temps d'exécution. On a donc  $C_{foo}(n) = \mathcal{O}(1)$ . On parle de complexité constante.

**Résultat** Pour une séquence de deux instructions de coûts respectifs  $C_1$  et  $C_2$ , le coût total est de la séquence est de  $C_1 + C_2$ .

■ **Exemple**

Soit la fonction suivante.

```
def foo() :
    a=20
    print(a)
```

Le coût temporel correspond à l'addition du coût élémentaire de l'affectation ajouté au coût de l'affichage. On a donc  $C_{foo}(n) = C_e + C_{print}$ . Le coût du `print` est variable en fonction, par exemple, de la longueur de la chaîne à afficher.

`foo` prendra (a priori) toujours le même temps d'exécution. On a donc  $C_{foo}(n) = \mathcal{O}(1)$ . On parle de complexité constante.

**Résultat** Le coût d'un test `if test : inst_1 else : inst_2` est inférieur ou égal au maximum du coût de l'instruction 1 et du coût de l'instruction 2 additionné au coût du test (coût élémentaire).

■ **Exemple**

Soit le programme suivant (sans application réelle) :

```
def foo(x) :
    if x < 0 :
        x = x + 1
        x = x + 2
    else :
        x = x + 1
```

La comparaison a un coût élémentaire  $C_e$ . Dans le « pire » des cas, on réalise deux additions et deux affectations. Le coût temporel total est donc  $C_{\text{foo}}(n) = C_e + \max(4C_e, 2C_e) = 5C_e$ . `foo` prendra (a priori) toujours le même temps d'exécution. On a donc  $C_{\text{foo}}(n) = \mathcal{O}(1)$ . On parle de complexité constante.

**Résultat** Le coût d'une boucle `for i in range(n) : inst` est égal à :  $n$  fois le coût de l'instruction `inst` si elle est indépendante de la valeur de  $i$ .

### ■ Exemple

Soit la fonction suivante :

```
def foo(n) :
    res = 0
    for i in range(n) :
        res = res + i
    return res
```

On peut considérer que l'incrément de  $i$  a un coût  $C_e$ .  $C_{\text{foo}}(n) = C_e + n \times 3C_e = \mathcal{O}(n)$ . La durée de l'algorithme croît linéairement avec la valeur de  $n$ .

**Résultat** Soit la boucle `while cond : inst`, la condition `cond` faisant intervenir un variant de boucle. Il est donc possible de connaître le nombre  $n$  d'itérations de la boucle.

**Résultat** Dans la pratique, on cherche toujours à **majorer** le coût temporel d'un algorithme. En conséquences :

- il est inutile de compter exactement le nombre d'opérations ;
- il « suffit » de se placer dans le pire des cas et de compter le nombre de fois qu'est réalisée l'opération se réalisant le plus de fois.

## 2.3 Exemple

### ■ Exemple Calcul de factorielle

```
def factorielle(n) :
    if n == 0 :
        return 1
    else :
        i = 1
        res = 1
        while i <= n :
            res = res * i
            i = i + 1
        return res
```

**Complexité en mémoire**  $C_M$  : lors de l'exécution du programme, il sera nécessaire de stocker les variables suivantes :  $n$ ,  $res$ ,  $i$ .

La complexité en mémoire est donc constante :  $C_M = \mathcal{O}(1)$ .

#### Complexité en temps $C_T$ (1)

Compte-tenu d'un des résultats précédents, dans le pire des cas, on est dans le `else`. La boucle `while` est réalisée  $n$  fois ; donc on peut directement que  $C_T(n) = \mathcal{O}(n)$ .

#### Complexité en temps $C_T$ (2) Une justification plus exhaustive serait la suivante.

La première comparaison a un coût élémentaire  $C_e$ .

Pour  $n = 0$  le coût du retour est  $C_r$ .

Pour  $n \neq 0$  :

- les deux affectations ont un coût respectif  $C_e$  ;
- la boucle tant que sera réalisée  $n$  fois. Pour chaque itération,
  - la multiplication ainsi que l'affectation ont chacun un coût  $C_e$  ;
  - l'incrément et l'affectation ont chacun un coût  $C_e$  ;
- le coût du retour est  $C_r$ .

En conséquence, la complexité en temps s'élève à :

$$C_T(n) = C_e + \max(C_r; C_e + C_e + n(4C_e) + C_r)$$

Ainsi  $C_T(n) = C_e(3 + 4n) + C_r$  et  $C_T(n) \underset{+\infty}{\sim} 4C_e n$  lorsque  $n$  tend vers l'infini. On parle d'une complexité algorithmique linéaire, notée  $\mathcal{O}(n)$ .

Il est fréquent que la complexité en temps soit améliorée au prix d'une augmentation de la complexité en espace, et vice-versa. La complexité dépend notamment :

- de la puissance de la machine sur laquelle l'algorithme est exécuté;
- du langage et compilateur / interpréteur utilisé pour coder l'algorithme;
- du style du programmeur.

## 2.4 D'autres exemples

### 2.4.1 Recherche d'un maximum

■ **Exemple** Soit une liste de nombre entiers désordonnés. Comment déterminer le plus grand nombre de la liste ?  
Intuitivement, une solution est de parcourir la liste d'éléments et de déterminer le plus grand élément par comparaisons successives.

```
def cherche_max(L : list) -> int :
    maxi = L[0]
    for i in range(1, len(L)):
        if tab[i] > maxi :
            maxi = tab[i]
    return maxi
```

Dans ce cas, le coût temporel est :  $C_T(n) = C_e + n(2C_e)$ . Ici encore, la complexité de cet algorithme est linéaire car  $C_T(n) \underset{+\infty}{\sim} 2C_e n$ .

Par ailleurs, en comptant le nombre de comparaisons, on observe aussi qu'il y a environ  $n$  comparaisons. On a donc  $C_T(n) = \mathcal{O}(n)$ .

### 2.4.2 Tri d'une liste

#### ■ Exemple Algorithme naïf

Soit une liste de nombre entiers désordonnés. Comment les trier par ordre croissant ?

Une méthode dite naïve pourrait être la suivante :

- trouver le plus petit élément du tableau. Notons  $min$  son indice;
- on permute alors le  $min^e$  élément avec le premier élément;
- ...
- on trouve le plus petit élément du tableau compris entre l'indice  $i$  et  $N$ ;
- on permute alors le  $min^e$  élément avec le  $i^e$  élément.

#### ■ Pseudo Code

**Data:**  $tab$  tableau d'entiers désordonnés de taille  $n$

**Result:**  $tab$  tableau d'entiers ordonnés

```
for i = 1 to n - 1 do
    min ← i
    for j = i + 1 to n do
        if tab[j] < tab[min] then
            min ← j
        end
    end
    tmp ← tab[i]
    tab[i] ← tab[min]
    tab[min] ← tmp
end
```

Ici les bornes de la boucle imbriquée dépendent de l'indice  $i$ . Ainsi :

- au rang 1,  $C_1 = C_e + (n-1)(2C_e) + 3C_e$ ;
- au rang 2,  $C_2 = C_e + (n-2)(2C_e) + 3C_e$ ;
- au rang  $i$ ,  $C_i = C_e + (n-i)(2C_e) + 3C_e$ .

Le coût temporel peut donc s'exprimer ainsi :

$$\begin{aligned} C_T(n) &= \sum_{i=0}^n (C_e + (n-i)(2C_e) + 3C_e) = C_e \sum_{i=0}^n (1 + 2n - 2i + 3) \\ &= C_e \sum_{i=0}^n (4 + 2n - 2i) = C_e \left( 4n + 2n^2 - 2 \frac{n(n+1)}{2} \right) = C_e (3n + n^2) \end{aligned}$$

Dans ce cas,  $C_T(n) \sim C_e n^2$ . On parle de complexité quadratique. Lorsque la taille du tableau double, le temps de calcul est multiplié par 4.

### 2.4.3 Diviser pour régner – recherche dichotomique

■ **Exemple**

```
def recherche_dichotomique(x, a):
    g, d = 0, len(a)-1
    while g <= d:
        m = (g + d) // 2
        if a[m] == x:
            return m
        elif a[m] < x:
            g = m+1
        else:
            d = m-1
    return None
```

On peut montrer que la suite  $d - g$  décroît strictement (car  $d$  décroît et  $g$  croît). Dans ce cas, la difficulté consiste en déterminer le nombre de fois que sera exécutée la boucle `while`. On note  $C_w = C_e + \max(C_e + C_r; 2C_e + 2C_e; 3C_e + C_e) = C_e + \max(C_e + C_r; 4C_e)$  le coût d'une itération de la boucle `while`.

Au cours de l'algorithme, on va devoir diviser en 2 la taille le tableau jusqu'à ce qu'on trouve (ou pas) l'élément recherché. On cherche donc combien de fois  $m$  on peut diviser par 2 la taille du tableau  $n$  :

$$\frac{n}{2^m} \geq 1 \iff n \geq 2^m \iff \ln(n) \geq m \ln(2)$$

On parlera ici de complexité logarithmique.

**Résultat** Pour une opération ayant un temps d'exécution de  $10^{-9} s$ , on peut calculer le temps d'exécution en fonction du nombre de données et de la complexité de l'algorithme :

Données	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$
100	$2 \cdot 10^{-9} s$	$0,1 \cdot 10^{-6} s$	$0,2 \cdot 10^{-6} s$	$10 \cdot 10^{-6} s$	$1,26765 \cdot 10^{21} s$
1 000	$3 \cdot 10^{-9} s$	$1 \cdot 10^{-6} s$	$3 \cdot 10^{-6} s$	$0,001 s$	$1,0715 \cdot 10^{292} s$
10 000	$4 \cdot 10^{-9} s$	$10 \cdot 10^{-6} s$	$40 \cdot 10^{-6} s$	$0,1 s$	$+\infty$

## 3 Profiling des algorithmes

Afin d'évaluer la performance des algorithmes, il existe des fonctionnalités permettant de compter le temps consacré à chacune des fonctions ou à chacune des instructions utilisées dans un programme <http://docs.python.org/2/library/profile.html>.

■ **Exemple** Voici un exemple du crible d'Eratosthène.

```
def crible(n):
    tab=[]
    for i in range(2,n):
        tab.append(i)
    # Liste en comprehension tab=[x for x in range(2,n)]
    for i in range(0,len(tab)):
        for j in range(len(tab)-1,i,-1):
            if (tab[j]%tab[i]==0):
```

```
        tab.remove(tab[j])
    return tab

import cProfile
cProfile.run('crible(10000)')
```

cProfile renvoie alors le message suivant :

```
28770 function calls in 1.957 seconds
```

```
Ordered by: standard name
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1  0.000  0.000  1.957  1.957  <string>:1(<module>)
    1  0.420  0.420  1.957  1.957  eratosthene.py:4(crible)
    1  0.000  0.000  1.957  1.957  {built-in method exec}
9999  0.015  0.000  0.015  0.000  {built-in method len}
9998  0.016  0.000  0.016  0.000  {method 'append' of 'list' objects}
    1  0.000  0.000  0.000  0.000  {method 'disable' of '_lsprof.Profiler' objects}
8769  1.505  0.000  1.505  0.000  {method 'remove' of 'list' objects}
```

On alors le bilan du temps passé à effectuer chacune des opérations. Ainsi pour améliorer notablement l'algorithme, le plus intéressant serait d'optimiser la méthode `remove`.

## Références

- [1] François Denis <http://pageperso.lif.univ-mrs.fr/~francois.denis/algoL2/chap1.pdf>
- [2] Alain Soyeur <http://asoyeur.free.fr/>
- [3] François Morain, Cours de l'Ecole Polytechnique, <http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Morain/TC/X2004/Poly/www-poly009.html>.
- [4] Renaud Kerivent et Pascal Monasse, La programmation pour ... , Cours de l'École des Ponts ParisTech - 2012/2013 <http://imagine.enpc.fr/~monasse/Info>.
- [5] Olivier Bournez, Cours INFO 561 de l'Ecole Polytechnique, Algorithmes et programmation, <http://www.enseignement.polytechnique.fr/informatique/INF561/uploads/Main/poly-good.pdf>.
- [6] Wack et Al., *L'informatique pour tous en classes préparatoires aux grandes écoles*, Editions Eyrolles.