# Méthodes de Conception d'Algorithmes

C. TRABELSI & E. MARTINS & M. FRANÇOIS

## TD1 - - Analyse de la complexité d'un algorithme

## Avant propos

*Comme nous l'avons vu au premier semestre, une multitude d'algorithmes peuvent résoudre un même problème. Mais ces derniers sont tous différents et n'auront donc pas forcément la même performance face aux problèmes auxquels ils sont confrontés. Un algorithme par exemple peut prendre deux à deux mille fois plus de temps qu'un autre pour arriver au même résultat, ou cent fois plus de mémoire, etc. Il est donc très important de pouvoir comparer des algorithmes entre eux pour savoir lesquels choisir. Pour cela, des outils ont été introduits et permettent de mesurer la complexité d'un algorithme. Nous utiliserons la notation de Landau (méthode de comparaison asymptotique).*

### 1 Rappel : complexité et notation de Landau

**Complexité en temps et en mémoire :** afin de pouvoir comparer des algorithmes entre eux en terme d'efficacité, on étudie leur complexité c'est-à-dire l'évolution de leur coût proportionnellement à la quantité d'information à traiter.

- Un algorithme de complexité en temps linéaire $(i.e\ O(n))$, mettra deux fois plus de temps à traiter deux fois plus d'informations.
- Un algorithme de complexité en temps quadratique $(i.e\ O(n^2))$, mettra quatre fois plus de temps à traiter deux fois plus d'informations.

**Mesure du temps d'exécution :** on mesure le temps d'exécution d'un algorithme en nombre d'instructions (opération élémentaire : addition, affectation, etc.) ou en nombre de cycles d'horloge (c'est l'unité de temps pour le processeur ; une opération nécessite $n$ cycles d'horloge). Ces deux notations ne sont pas dépendantes de la puissance du processeur (contrairement au temps d'exécution en seconde). On a : $t_{secondes} = nb_{cycles}/freq_{proc}$.

**Remarque :** *ce qui nous intéresse c'est le comportement asymptotique (à l'infini) de l'algorithme. Nous ne serons donc pas pointilleux sur le nombre **exacte** du nombre d'instructions / du nombre de cycles. On considérera en première approximation, que toutes les instructions coûtent le **même** nombre de cycles.*

**NB d'instructions = NB d'affectations + NB d'opérations arithmétiques + NB de comparaisons pour traiter $n$ informations**

**Mesure du coût en mémoire :** on mesure le coût en mémoire d'un algorithme par la quantité de mémoire vive (en octet) utilisée au maximum durant l'exécution de ce dernier. De la même manière que pour le coût en temps, c'est la forme de l'évolution de coût (constant, linéaire, etc.) qui est intéressant.

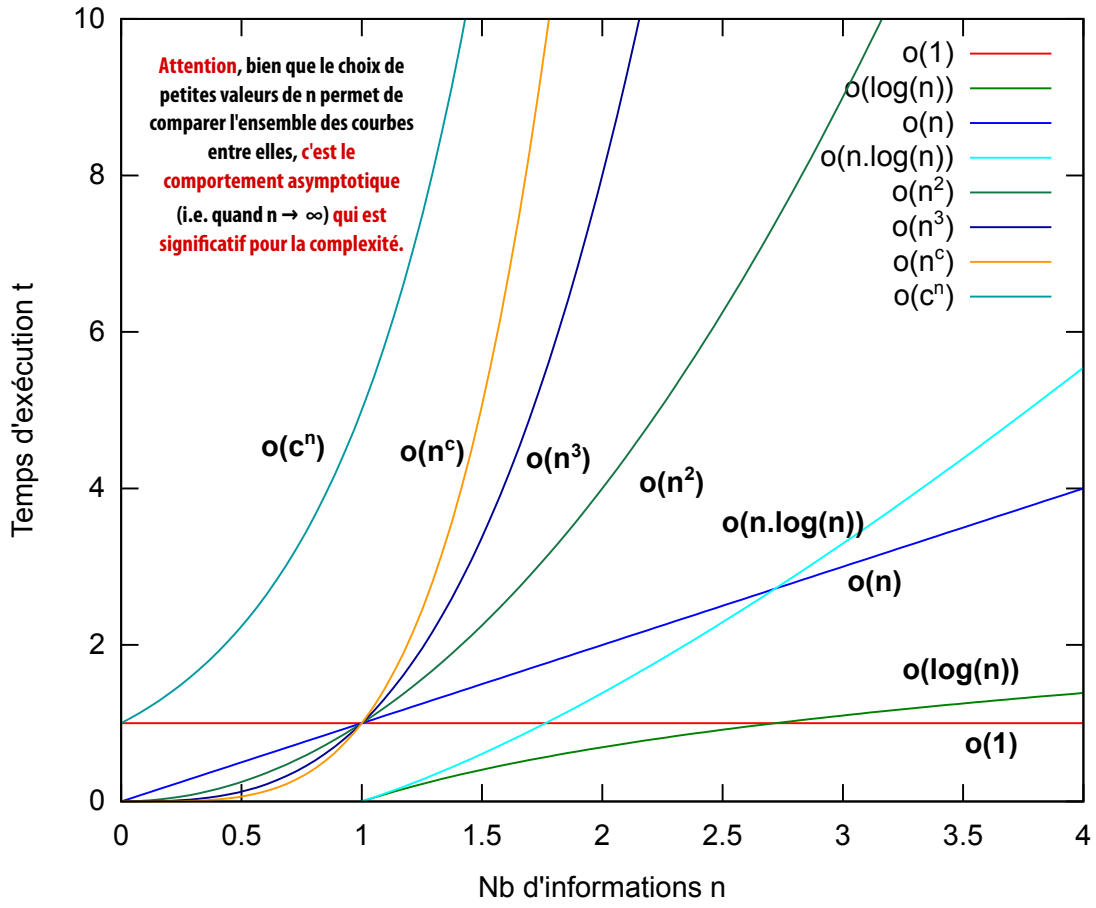
**Coût mémoire = NB d'octets réservés pour traiter $n$ informations**

**Notation de Landau :** la notation de Landau est une notation mathématique signifiant qu'une fonction contient une composante prépondérante qui définie son comportement asymptotique (à l'infini). Par exemples :
$f(x) = 3x + 2 = O(x)$, quand $x \to \infty$ ou
$g(x) = 5x^2 - 2x + 4 = O(x^2)$, quand $x \to \infty$

Basé sur cette typologie on retrouve donc un certain nombre de classes d'algorithmes récapitulées dans le tableau suivant :

| | Notation de Landau | Appelation |
|---|---|---|
| - coûteux | $O(1)$ | constante |
| | $O(log(n))$ | logarithmique |
| | $O(n)$ | linéaire |
| | $O(nlog(n))$ | quasi-linéaire (linéarithmique) |
| | $O(n^2)$ | quadratique |
| | $O(n^c)$ | polynomiale |
| | $O(c^n)$ | exponentielle |
| + coûteux | $O(n!)$ | factorielle |

La figure suivante donne un aperçu de la rapidité de croissance de ces différentes courbes et illustre bien la notion de coût rattaché à chacune.



## 2 Mesure de complexité : cas pratiques

Pour vous familiariser avec la mesure de complexité, vous allez reprendre les algorithmes que nous avons vu au premier semestre et les analyser. Pour simplifier, nous supposerons que toutes les opérations ou les méta-actions nécessitent le **même temps** d'exécution. Par exemples :

- `var = 0` est une affectation : son coût $\Longrightarrow 1$
- `var2 = var1 + 2` comporte une addition et une affectation : son coût $\Longrightarrow 1+1=2$
- `var1 <= var2` est une comparaison (méta-action) : son coût $\Longrightarrow 1$
- `printf("%d", var)` est une méta-action : son coût $\Longrightarrow 1$.
- **`for`** `(i = 0; i < n; i++)` comporte une unique affectation, un test qui est répété $n+1$ fois ($n$ tours de boucle, plus un dernier avant de sortir) et une incrémentation (addition + affectation) répétée $n$ fois : coût total $\Longrightarrow 1+(n+1)\times 1+(1+1)\times n = 1+n+1+2n = 3n+2$

### 2.1 Algorithme d'échange

Affecter `var1` à `varTemp`
Affecter `var2` à `var1`
Affecter `varTemp` à `var2`

► **1.** Calculer la complexité en temps de l'algorithme :

| | Valeur calculée | Landau | Appellation |
|---|---|---|---|
| dans le meilleur des cas | | | |
| dans le pire des cas | | | |
| en moyenne | | | |

► **2.** Considérant des variables de type **`int`**, dans le pire des cas, quelle quantité de mémoire (en octet), cet algorithme utilise-t-il ?

### 2.2 Algorithme de parcours (d'un tableau de **`taille`** $n$ cases)

Affecter 0 à `compteur`
**tant que** *`(compteur < n)`* **faire**
  Afficher `TAB[compteur]`
  Affecter `compteur+1` à `compteur`
**fin**

► **3.** Calculer la complexité en temps de l'algorithme :

| | Valeur calculée | Landau | Appellation |
|---|---|---|---|
| dans le meilleur des cas | | | |
| dans le pire des cas | | | |
| en moyenne | | | |

► **4.** Considérant des variables de type **`int`**, dans le pire des cas, quelle quantité de mémoire (en octet), cet algorithme utilise-t-il ?

### 2.3 Algorithme de recherche d'un extremum

```
Affectez TAB[0] à minimum
pour compteur allant de 1 à n-1 inclus faire
    si (TAB[compteur] < minimum) alors
        Affecter TAB[compteur] à minimum
    fin
fin
Retournez minimum
```

► **5.** Calculer la complexité en temps de l'algorithme :

| | Valeur calculée | Landau | Appellation |
|---|---|---|---|
| dans le meilleur des cas | | | |
| dans le pire des cas | | | |
| en moyenne | | | |

► **6.** Considérant des variables de type **`int`**, dans le pire des cas, quelle quantité de mémoire (en octet), cet algorithme utilise-t-il ?

### 2.4 Algorithme de recherche de l'indice d'un extremum

```
Affectez TAB[0] à minimum
Affecter 0 à indice
pour compteur allant de 1 à n-1 inclus faire
    si (TAB[compteur] < minimum) alors
        Affecter TAB[compteur] à minimum
        Affecter compteur à indice
    fin
fin
Retournez indice
```

► **7.** Calculer la complexité en temps de l'algorithme :

| | Valeur calculée | Landau | Appellation |
|---|---|---|---|
| dans le meilleur des cas | | | |
| dans le pire des cas | | | |
| en moyenne | | | |

► **8.** Considérant des variables de type **`int`**, dans le pire des cas, quelle quantité de mémoire (en octet), cet algorithme utilise-t-il ?

### 2.5 Algorithme de tri à bulles

```
répéter
    Affecter vrai à estTrié
    pour indice allant de 0 à n-2 inclus faire
        si (TAB[indice] > TAB[indice+1]) alors
            Échanger TAB[indice] et TAB[indice+1]
            Affecter faux à estTrié
        fin
    fin
tant que estTrié est faux;
```

► **9.** Calculer la complexité en temps de l'algorithme :

| | Valeur calculée | Landau | Appellation |
|---|---|---|---|
| dans le meilleur des cas | | | |
| dans le pire des cas | | | |
| en moyenne | | | |

► **10.** Considérant des variables de type **`int`**, dans le pire des cas, quelle quantité approximative de mémoire (en octet), cet algorithme utilise-t-il ?