Complexité algorithmique

Le calcul de la complexité d'un algorithme permet de mesurer sa performance. Il existe deux types de complexité :

- complexité spatiale : permet de quantifier l'utilisation de la mémoire
- complexité temporelle : permet de quantifier la vitesse d'exécution

1 Complexité temporelle

L'objectif d'un calcul de complexité algorithmique temporelle est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Réaliser un calcul de complexité en temps revient à compter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme.
Puisqu'il s'agit seulement de comparer des algorithmes, les règles de ce calcul doivent être indépendantes :

- du langage de programmation utilisé ;
- du processeur de l'ordinateur sur lequel sera exécuté le code ;
- de l'éventuel compilateur employé.

Par soucis de simplicité, on fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 « unité » de temps.
Exemple : a = b * 3 : 1 multiplication + 1 affectation = 2 « unités »

La complexité en temps d'un algorithme sera exprimé par une fonction, notée T (pour Time), qui dépend :

- Du nombre de données n passées en paramètres : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter.
- de la donnée en elle même, de la façon dont sont réparties les différentes valeurs qui la constituent.
  par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès le début si le premier élément est « le bon ». Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Cette remarque nous conduit à préciser un peu la définition de la complexité en temps. Il est en effet possible distinguer différentes complexités en temps :

- la complexité dans le meilleur des cas
- la complexité dans le pire des cas
- la complexité moyenne
- etc

On calculera le plus souvent la **complexité dans le pire des cas**, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

2 Ordre de grandeur

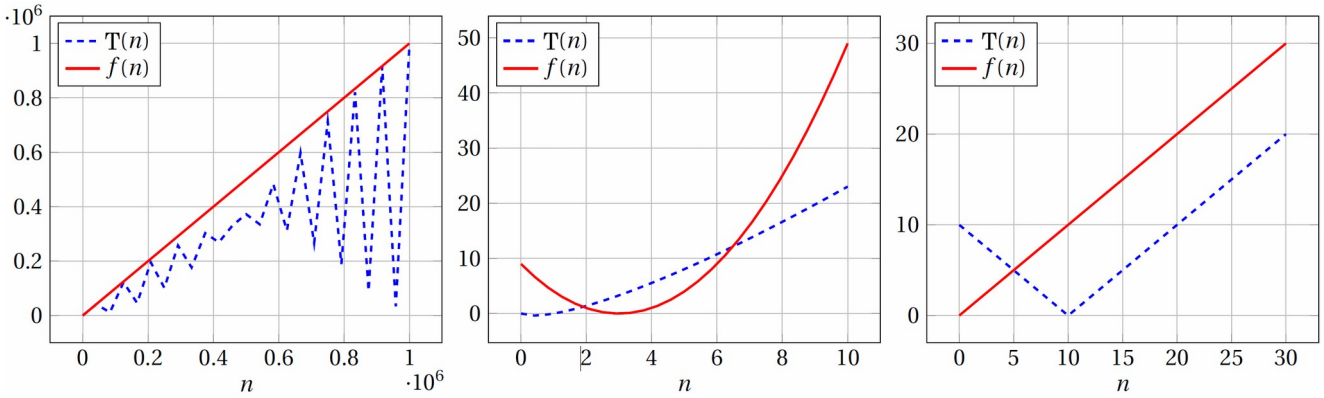
Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction T, mais seulement l'ordre de grandeur asymptotique, noté O (« grand O »).

Une fonction $T(n)$ est en $O(f(n))$ (« en grand O de $f(n)$ ») si :

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, \forall n \in \mathbb{R}^+, n \geq n_0 \Rightarrow |T(n)| \leq c|f(n)|$$

Autrement dit : $T(n)$ est en $O(f(n))$ s'il existe un seuil n_0 à partir duquel la fonction T est toujours dominée par la fonction f, à une constante multiplicative fixée c près.

Exemples :
$T_1(n) = 7 = O(1)$
$T_2(n) = 12n + 5 = O(n)$
$T_3(n) = 4n^2 + 2n + 6 = O(n^2)$
$T_4(n) = 2 + (n-1) \times 5 = O(n)$



3 Classes de complexité

$O(1)$	constante	$O(n^2)$	quadratique
$O(\log(n))$	logarithmique	$O(n^3)$	cubique
$O(n)$	linéaire	$O(2^n)$	exponentielle
$O(n.\log(n))$	quasi-linéaire	$O(n!)$	factorielle

4 Exemple de calcul de complexité

```
def factorielle(n):
    fact = 1
    i = 2

    while i <= n:
        #  i <= n
        fact = fact * i
        i = i + 1
    return fact
```

```
.
affectation : 1
affectation : 1
.
itérations : au plus x(n - 1)
    comparaison : 1
    multiplication + affectation : 2
    addition + affectation : 2
```

On effectue donc $(n-1) \times (1+2+2) = 5n-5$ opérations. La complexité est donc $O(n)$.