

---

# MPSI - Informatique Tronc Commun

## Cours 7 - Ordres de complexité

---

### I Première approche de la complexité : opération par opération

**1 - Idéalement** on voudrait compter combien d'opérations réalise une fonction quand elle est appelée. Pour ça, il faut commencer par décider quelles sont les opérations qu'on considère comme étant **élémentaires** (s'exécutant rapidement, avec une durée qui ne dépend pas de la taille de l'entrée).

**2 - Les opérations sur les booléens sont élémentaires.** Tant les opérations binaires (**and**, **or**) comme l'opération unaire **not**.

**3 - Les opérations sur les flottants sont élémentaires :**  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $**$ ,  $==$

**4 - Les opérations sur les entiers sont élémentaires :**  $+$ ,  $-$ ,  $*$ ,  $//$ ,  $==$ . À condition que les entiers en question et le résultat ne soient ni trop grands ni trop petits. Typiquement, pour des entiers entre  $-2^{63}$  et  $2^{63} - 1$  on considèrera ces opérations comme élémentaires.

**5 - L'assignation à une variable, ainsi que le début et la fin d'une fonction sont des opérations élémentaires :**

- Assigner à une variable une valeur déjà calculée, est une opération élémentaire
- Démarrer un appel de fonction (assigner aux paramètres les valeurs des arguments) est élémentaire
- Finir un appel de fonction (renvoyer le résultat) est une opération élémentaire

**6 - Exemple pour une fonction très simple** Si on appelle  $n$  la taille de la liste en entrée, chaque appel à la fonction suivante

```
def est_triee(liste):
    for i in range(len(liste) - 1):
        if liste[i] > liste[i+1]:
            return False
    return True
```

effectue au plus  $6 \times (n - 1) + 5$  opérations élémentaires, si  $n \neq 0$ .

**7 - Problème** Pour des fonctions qui soient pas extrêmement simples, compter les opérations une par une est pénible... et pas nécessaire : ce qui nous intéresse est de comprendre **comment** la complexité augmente lorsque la taille de l'entrée augmente.

**8 - Solutions** Il y a au moins deux façons plus simples de s'intéresser à la complexité :

1. En ne s'intéressant qu'à un type d'opérations en particulier (par exemple, combien d'additions au plus réalise chaque appel à cette fonction)... qui soit représentatif du coût de la fonction.
2. En utilisant les notations de Landau (suite de ce cours).

### II Les notations de Landau

**9 - Définition (Grand O)** Soit  $f : \mathbb{N} \rightarrow \mathbb{N}^*$  une fonction. On définit l'ensemble de fonctions  $O(f)$  par

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{N}^* \mid \exists k > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq kf(n)\}$$

**10 - Définition (Grand Oméga)** Soit  $f : \mathbb{N} \rightarrow \mathbb{N}^*$  une fonction. On définit l'ensemble de fonctions  $\Omega(f)$  par

$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{N}^* \mid \exists k > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, kf(n) \leq g(n)\}$$

**11 - Définition (Grand Theta)** Soit  $f : \mathbb{N} \rightarrow \mathbb{N}^*$  une fonction. On définit l'ensemble de fonctions  $\Theta(f)$  par

$$\Theta(f) = O(f) \cap \Omega(f)$$

**12 - Abus de notation usuels** Soient  $f : \mathbb{N} \rightarrow \mathbb{N}^*$ ,  $g : \mathbb{N} \rightarrow \mathbb{N}^*$  et  $h : \mathbb{N} \rightarrow \mathbb{N}^*$  trois fonctions.

On se permettra souvent de noter  $f(n) = O(g(n))$  au lieu de  $f \in O(g)$ , tout comme on se permettra d'énoncer des règles de calcul de la forme

$$O(f(n)) + O(g(n)) = O(h(n))$$

en lieu et place de

$$\forall f_1 \in O(f), \forall g_1 \in O(g), (f_1 + g_1) \in O(h)$$

De même pour les  $\Theta$  et les  $\Omega$ .

#### II.1 Règles de calcul

**13 - Préservation par multiplication par une constante**

Si  $k > 0$  est une constante (qui ne dépend pas de  $n$ , donc), et  $f(n) = O(g(n))$ , alors

$$kf(n) = O(g(n))$$

De même pour  $\Theta$  et  $\Omega$ .

**14 - Préservation par somme constante**

Si  $I$  est un ensemble fini dont le cardinal est borné par une constante (qui ne dépend pas de  $n$ , donc) et que  $\forall i \in I, f_i(n) = O(g(n))$  alors

$$\sum_{i \in I} f_i(n) = O(g(n))$$

De même pour  $\Theta$  et  $\Omega$ .

**15 - Multiplication de fonctions**

Si  $f(n) = O(h_1(n))$  et  $g(n) = O(h_2(n))$ , alors

$$f(n) \times g(n) = O(h_1(n) \times h_2(n))$$

De même pour  $\Theta$  et  $\Omega$ .

**16 - Dans une somme, on ne garde que le terme dominant :** Si  $f(n) = O(g(n))$ , alors

$$O(f(n) + g(n)) = O(g(n))$$

De même pour  $\Theta$  et  $\Omega$ .

III Les ordres de complexité

Quand on analyse la complexité d’une fonction, ce qui nous intéresse est de savoir dans quel ordre asymptotique se situe cette fonction.

17 - Ordres de complexité usuels Si on appelle  $n$  la taille de l’entrée, les ordres de complexité que l’on va croiser le plus souvent sont :

| Ordre                                  | Nom            |
|----------------------------------------|----------------|
| $\Theta(1)$                            | Temps constant |
| $\Theta(\log(n))$                      | Logarithmique  |
| $\Theta(n)$                            | Linéaire       |
| $\Theta(n \log(n))$                    | Quasi-linéaire |
| $\Theta(n^2)$                          | Quadratique    |
| $\Theta(k^n)$ où $k$ est une constante | Exponentielle  |
| $\Theta(n!)$                           | Factorielle    |

18 - Remarque : on se contentera de  $O$  . Même si en théorie ce qui nous intéresse c’est d’avoir un  $\Theta$ , on va le plus souvent se contenter d’un  $O$ , ce qui peut parfois simplifier les calculs. Si on a un  $O$  et qu’on veut passer à un  $\Theta$  on peut exhiber une famille d’entrées  $(e_n)_{n \in \mathbb{N}}$  pour laquelle la majoration asymptotique trouvée est en effet atteinte.

19 - Remarque Toutes les fonctions logarithme étant égales à constante multiplicative près, et les constantes pouvant être cachées par les  $O$ ,  $\Omega$  et  $\Theta$ , il n’est pas nécessaire de préciser dans quelle base est le logarithme.

20 - Les opérations "élémentaires" des paragraphes 2 à 5 sont en temps constant.

IV Complexité des opérations sur les listes

21 - Opérations sur les listes en temps constant Certaines opérations sur les listes python s’exécutent en temps constant :

- L’accès à une case de la liste. Syntaxe `liste[indice]`
- L’affectation à une case de la liste. Syntaxe `liste[indice] = valeur`
- Le retrait du dernier élément de la liste. Syntaxe `liste.pop()`
- Connaître la taille d’une liste. Syntaxe `len(liste)`

22 - Insertion en fin de liste, si on ne regarde pas de trop près. Insérer un élément en fin de liste (syntaxe `liste.append(valeur)`) ne prend pas toujours le même temps à s’exécuter, mais s’exécute en moyenne en temps constant.

23 - Opérations sur les listes en complexité linéaire Certaines opérations sur les listes ont une complexité linéaire en la liste en question. Soit `liste` une liste de taille  $n$ . Soit `liste2` une liste de taille  $m$ . Soit  $i$  un indice valable dans `liste`. Soit  $x$  un élément de `liste`. Soit  $y$  un élément à ajouter dans `liste`. Soit  $k$  une constante telle que  $k+i-1$  est aussi un indice valable dans `liste`.

- La copie de liste est en  $O(n)$ . Syntaxe `liste.copy()`.
- Le retrait par indice pour un indice autre que le dernier est en  $O(n-i)$ . Syntaxe `liste.pop(i)`.
- L’insertion par indice est en  $O(n)$ . Syntaxe `liste.insert(i, y)`.
- La suppression d’élément par valeur est en  $O(n)$ . Syntaxe `liste.remove(x)`.
- La création d’une tranche de taille  $k$  est en  $O(k)$ . Exemple de syntaxe `liste[i:i+k]`.
- La concaténation de deux listes se fait en temps linéaire en la taille de la deuxième liste. Par exemple `liste + liste2` s’évalue en  $O(m)$ .
- La multiplication `liste*k` s’évalue en  $O(kn)$  et le test `x in liste` s’évalue en  $O(n)$ .