# Complexité algorithmique

## 1 Dialogue sur la complexité

*Professeur* : Aujourd'hui, nous allons parler d'un concept très important en informatique : **la complexité algorithmique**. Est-ce que quelqu'un sait ce que cela signifie ?

*Élève* : C'est… la difficulté d'un programme ?

*Professeur* : Pas exactement. La complexité algorithmique mesure **le nombre d'opérations nécessaires pour qu'un algorithme s'exécute**, en fonction de la taille des données. Autrement dit : plus les données augmentent, plus on regarde comment le temps de calcul évolue.

*Élève* : Mais si mon programme marche déjà, pourquoi c'est important ?

*Professeur* : Bonne question ! Imaginez que votre programme trie **10 nombres**. Il fonctionne très bien. Mais si on lui demande de trier **10 millions de nombres**, est-ce qu'il sera toujours rapide ?

*Élève* : Ah… peut-être pas.

*Professeur* : Exactement. Certains algorithmes deviennent **très lents** quand la taille des données augmente. La complexité permet donc de **prévoir les performances** d'un algorithme avant même de l'exécuter.

*Élève* : Comment on mesure ça ?

*Professeur* : On utilise ce qu'on appelle **la notation "Grand O"**. Elle décrit comment le temps d'exécution évolue lorsque la taille de l'entrée augmente.

*Élève* : Vous pouvez donner un exemple ?

*Professeur* : Bien sûr. Prenons un algorithme qui parcourt une liste de **n éléments** pour chercher une valeur. Dans le pire des cas, il doit regarder tous les éléments.

*Élève* : Donc il fait **n opérations** ?

*Professeur* : Exactement. On dit que sa complexité est **O(n)**, ce qui signifie que le temps de calcul est proportionnel au nombre d'éléments.

*Élève* : Et si on a deux boucles ?

*Professeur* : Très bonne remarque. Si vous avez une boucle **dans une autre boucle**, chacune parcourant **n éléments**, le nombre d'opérations peut atteindre **n × n**, soit **$n^2$**.

*Élève* : Donc la complexité est **$O(n^2)$** ?

*Professeur* : Exactement. Et quand **n devient très grand**, la différence entre **O(n)** et **$O(n^2)$** devient énorme.

*Élève* : Vous avez un exemple concret ?

*Professeur* : Imaginez deux programmes qui traitent **1 million de données**.

- Un algorithme en **O(n)** fera environ **1 million d'opérations**.
- Un algorithme en **$O(n^2)$** pourrait en faire **1 000 milliards**.

*Élève* : Ah oui… ça change tout !

*Professeur* : Voilà pourquoi les informaticiens cherchent souvent **le meilleur algorithme possible**, pas seulement un algorithme qui fonctionne.

*Élève* : Mais un algorithme fait-il toujours le même nombre d'opérations ?

*Professeur* : Excellente question ! En réalité, cela dépend souvent **des données qu'on lui donne**. C'est pour cela qu'on distingue **trois types de complexité**.

*Élève* : Lesquels ?

*Professeur* :

- **La complexité dans le meilleur des cas** : c'est la situation la plus favorable pour l'algorithme.
- **La complexité dans le pire des cas** : c'est la situation où l'algorithme doit faire le maximum d'opérations.
- **La complexité en moyenne** : c'est ce qui se passe en général pour des données typiques.

*Élève* : Vous avez un exemple ?

*Professeur* : Prenons la recherche d'un nombre dans une liste.

- **Meilleur cas** : le nombre est **au tout début de la liste**. On le trouve immédiatement.
- **Pire cas** : le nombre est **tout à la fin ou il n'est pas dans la liste**. Il faut parcourir toute la liste.
- **Cas moyen** : en général, on trouve l'élément **vers le milieu**.

*Élève* : Donc le nombre d'opérations peut varier ?

*Professeur* : Exactement. C'est pour cela que les informaticiens s'intéressent souvent **au pire des cas**, car il garantit que l'algorithme ne sera jamais plus lent que ce qu'on a prévu.

*Élève* : Donc même si ça va parfois plus vite, on préfère prévoir le pire ?

*Professeur* : Oui. Cela permet de concevoir des programmes **fiables et prévisibles**, surtout quand on travaille avec de très grandes quantités de données.

*Élève* : Donc l'enjeu, c'est de rendre les programmes plus rapides ?

*Professeur* : Oui, mais pas seulement. Cela permet aussi :

- de **gérer de très grandes quantités de données**,
- d'**économiser de l'énergie et des ressources**,
- et de **concevoir des logiciels efficaces**.

*Élève* : Donc même si deux programmes donnent le même résultat, l'un peut être beaucoup meilleur que l'autre ?

*Professeur* : Exactement. En informatique, **la façon de résoudre un problème est souvent aussi importante que le résultat lui-même**.

*Élève* : Et c'est pour ça qu'on étudie la complexité ?

*Professeur* : Oui. Comprendre la complexité algorithmique permet de **choisir ou concevoir les bons algorithmes** quand les données deviennent très grandes. C'est un des fondements de l'informatique.

## 2 Complexité algorithmique temporelle

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

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

L'objectif d'un calcul de complexité 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.

On fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 « unité » de temps.
Exemple : a = b * 3 On a 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 nombreuses, plus il faudra d'opérations élémentaires pour les traiter.
- De la donnée en elle même, qui va avoir une influence sur la performance de notre algorithme.

On va ainsi devoir distinguer les complexités dans le meilleur des cas, le pire des cas, le cas moyen, etc.
On calculera le plus souvent la **complexité dans le pire des cas**, car elle est la plus pertinente pour un système robuste. Il vaut mieux en effet toujours envisager le pire.

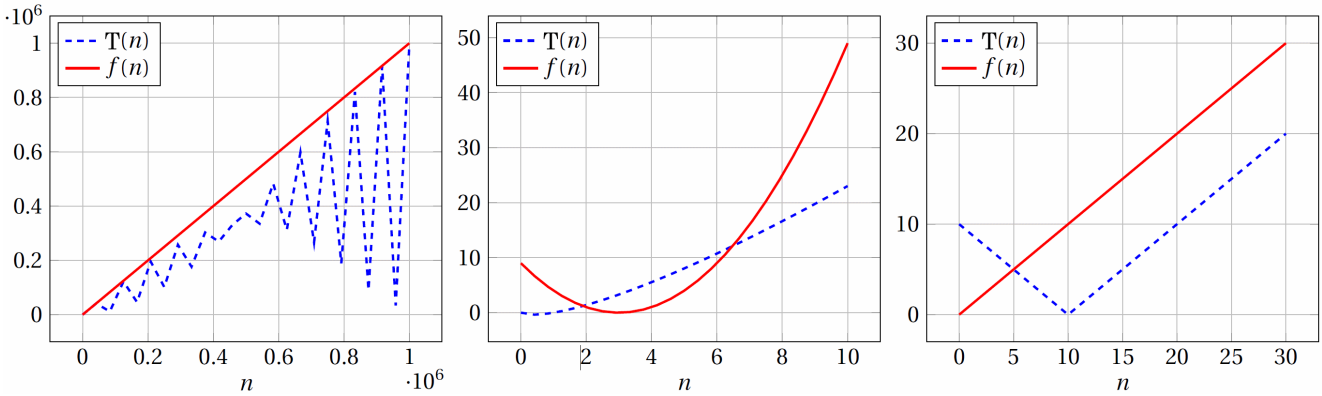
Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction $T$, mais seulement **l'ordre de grandeur asymptotique**, noté $O$ (« grand O »).

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 : $T_1(n)=7=O(1)$ $T_2(n)=12n+5=O(n)$ $T_3(n)=4n^2+2n+6=O(n^2)$



Voici quelques exemple de complexité :

$O(1)$ : L'algorithme est réalisé en **temps constant** par rapport à la donnée d'entrée.

```
def premier_element(liste):
    return liste[0]
```

La taille d'se fait en temps constant, peu importe la taille de la taille de liste.

$O(\log(n))$ : L'algorithme est réalisé en **temps logarithmique** par rapport à la donnée d'entrée.

```
def recherche_dichotomique(l, n):
    low, up = 0, len(l) - 1
    while low <= up:
        med = (low + up)//2
        if l[med] == n :
            return True
        elif l[med] < n :
            low = med + 1
        else:
            up = med - 1
    return False
```

L'algorithme de recherche dichotomique permet de confirmer la présence ou l'absence d'un élément dans une liste triée en temps logarithmique.

$O(n)$ : L'algorithme est réalisé en **temps linéaire** par rapport à la donnée d'entrée.

```
def affiche(liste, n):
    for e in liste :
        if e == n:
            return True
    return False
```

Une recherche d'un élément dans une liste non triée demande de parcourir tous les éléments de la liste dans le pire des cas (si l'élément n'est pas présent dans la liste).

$O(n.\log(n))$ : L'algorithme est réalisé en **temps quasi-linéaire** par rapport à la donnée d'entrée.

```
def vérif_recherche_dichotomique(l):
    for e in l :
        assert recherche_dichotomique(l,e)
```

Une vérification que la recherche dichotomique retrouve bien tous les éléments présents dans la liste va s'exécuter dans le pire des cas en $n \times O(\log n) = O(n.\log n)$

$O(n^2)$ : L'algorithme est réalisé en **temps quadratique** par rapport à la donnée d'entrée.

```
def tri_a_bulle(l):
    for i in range(len(l)-1, 1, -1):
        for j in range(i):
            if l[j] > l[j+1]:
                tmp = l[j]
                l[j] = l[j+1]
                l[j+1] = tmp
```

Le tri à bulle trie une liste en $O(n^2)$.

$O(2^n)$ : L'algorithme est réalisé en **temps exponentielle** par rapport à la donnée d'entrée.

```
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Le calcul de la suite de Fibonacci avec un algorithme naïf effectue environ $2^n$ appels récursifs donc $O(2^n)$.

$O(n!)$ : L'algorithme est réalisé en **temps factorielle** par rapport à la donnée d'entrée.

```
def permutations(tab, d=0):
    if d == len(tab):
        print(tab)
    else:
        for i in range(d, len(tab)):
            tab[d], tab[i] = tab[i], tab[d]
            permutations(tab, d + 1)
            tab[d], tab[i] = tab[i], tab[d]
```

Dans un ensemble à n éléments, il y a n! permutations possibles.