Algorithmique

Complexité temporelle

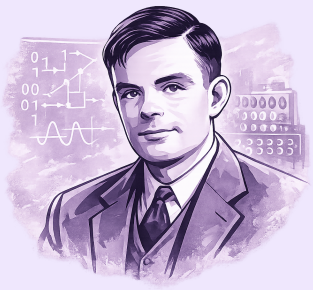
Cours

Pourquoi mesurer la complexité ?

Deux algorithmes peuvent résoudre le même problème tout en étant très différents en termes de vitesse. Sur de petites données, la différence est imperceptible ; sur de grands volumes, elle peut être spectaculaire : quelques millisecondes contre plusieurs heures !

La **complexité temporelle** d'un algorithme mesure le **nombre d'opérations** qu'il effectue en fonction de la **taille des données en entrée** (notée n). Elle permet de comparer des algorithmes indépendamment du langage de programmation ou de la machine utilisée.

Alan Turing (1912–1954) a posé les bases théoriques de ce domaine en introduisant en 1936 la **machine de Turing** : un modèle abstrait et universel de calcul. Ce modèle sert encore aujourd'hui de référence pour définir rigoureusement ce qu'est un algorithme et pour mesurer son coût.



Les opérations élémentaires

Opération élémentaire

Une **opération élémentaire** est une instruction basique supposée s'exécuter en un **temps constant**, quelle que soit la taille des données. On compte notamment :

- une **affectation** : `x = 5`
- une **opération arithmétique** : addition, soustraction, multiplication, division
- une **comparaison** : `a < b`, `a == b`
- un **accès à un élément** : `t[i]`
- un **return** ou un **print**

Convention — lecture d'une variable simple

On ne compte **pas séparément** la lecture d'une variable simple (`a`, `x`, `compteur`…) : elle est considérée comme incluse dans l'opération qui l'utilise. Seul l'accès **indexé** `t[i]` est explicitement compté, car il implique un calcul d'adresse mémoire.

Ainsi, dans `x = a + b`, on compte 2 opérations (1 addition + 1 affectation), et non 4.

On appelle **fonction de coût** le nombre total d'opérations élémentaires d'un algorithme, noté $T(n)$ où n est la taille des données en entrée. Voici un premier exemple annoté :

Exemple — séquence d'instructions

```
def calcul(a, b):
    x = a + b      # 1 addition  + 1 affectation = 2 opérations
    y = x * 2      # 1 multiplication + 1 affectation = 2 opérations
    z = x - y      # 1 soustraction + 1 affectation = 2 opérations
    return z       # 1 opération
```

On calcule $T = 2 + 2 + 2 + 1 = \mathbf{7}$. Cette valeur est **indépendante des valeurs** de `a` et `b`, et ne dépend d'aucun paramètre n : la complexité est **constante**.

Exercice 1 — Compter les opérations

Compter le nombre d'opérations élémentaires effectuées par la fonction suivante. On précisera le détail du calcul pour chaque ligne.

Code à analyser

```
def f(a, b, c):
    p = a * b
    q = p + c
    r = q * q - p
    return r
```

Réponse

.....

.....

.....

.....

Les boucles

Lorsqu'un algorithme contient une boucle, le coût total dépend du **nombre d'itérations** et du **coût du corps** de la boucle.

Coût d'une boucle

Si le corps d'une boucle coûte c opérations et que la boucle s'exécute n fois, le coût total de la boucle est $c \times n$ opérations.

Exemple — calcul de la somme d'une liste

```
def somme(t):                    # t est une liste de longueur n
    s = 0                        # 1 opération
    for i in range(len(t)):
        s = s + t[i]             # 1 accès + 1 addition + 1 affectation = 3
        ↪ opérations
    return s                     # 1 opération
```

La boucle s'exécute n fois. On calcule :

$$T(n) = 1 + 3n + 1 = 3n + 2$$

Quand n est grand, c'est le terme $3n$ qui domine. On dit que la complexité est **linéaire** en n : doubler la taille des données double le temps d'exécution.

Exercice 2 — Boucle simple

On considère la fonction suivante, où `t` est une liste de longueur n :

Code à analyser

```
def maximum(t):
    m = t[0]
    for i in range(1, len(t)):
        if t[i] > m:
            m = t[i]
    return m
```

1. Combien d'opérations la ligne `m = t[0]` effectue-t-elle ?

   .....

2. Combien d'itérations la boucle effectue-t-elle (en fonction de n) ?

   .....

3. Dans le **pire cas**, combien d'opérations sont effectuées à chaque itération ? (On suppose que la condition est toujours vraie.)

   .....

4. Exprimer le coût total en fonction de n.

   .....

Les boucles imbriquées

Lorsqu'une boucle contient elle-même une autre boucle, les coûts se **multiplient**.

Boucles imbriquées

Si une boucle externe s'exécute n fois et qu'elle contient une boucle interne s'exécutant elle aussi n fois, le corps de la boucle interne est exécuté $n \times n = n^2$ fois au total.

Exemple — vérifier si une liste contient des doublons

```
def contient_doublon(t):          # t est une liste de longueur n
    for i in range(len(t)):
        for j in range(len(t)):
            if i != j and t[i] == t[j]:      # 3 opérations
                return True
    return False
```

La boucle externe s'exécute n fois, la boucle interne aussi : le corps est exécuté n^2 fois. On a $T(n) = 3n^2$. On dit que la complexité est **quadratique**.

Exercice 3 — Boucles imbriquées

On considère la fonction suivante, où n est un entier :

Code à analyser

```
def afficher_paires(n):
    for i in range(n):
        for j in range(n):
            print(i, j)      # 1 opération
```

1. Combien de fois le corps de la boucle interne est-il exécuté ?

.....

2. Donner le coût total en fonction de n.

.....

3. Même question si la boucle interne était `for j in range(i+1, n)`.
   Indice : pour $i = 0$ on fait $n - 1$ tours, pour $i = 1$ on fait $n - 2$ tours, etc.

.....

Les tests

Les structures conditionnelles (`if / else`) introduisent une variabilité dans le nombre d'opérations : selon les données, une branche ou l'autre est exécutée.

Convention : le pire cas

Pour les structures conditionnelles, on adopte la convention du **pire cas** : on retient la branche qui coûte le **plus d'opérations**. On obtient ainsi une **borne supérieure** garantie sur le temps d'exécution.

Exemple — valeur absolue

```
def valeur_absolue(x):
    if x >= 0:          # 1 comparaison
        return x        # branche 1 : 1 opération
    else:
        return -x       # branche 2 : 1 opposé + 1 return = 2 opérations
```

On retient le **pire cas** : la branche `else`, qui coûte 2 opérations. On a $T = 1 + 2 = 3$ (coût constant, ne dépend pas de n).

Exercice 4 — Tests dans une boucle

On considère la fonction suivante, où `t` est une liste de longueur n :

Code à analyser

```
def compter_positifs(t):
    compteur = 0
    for i in range(len(t)):
        if t[i] > 0:
            compteur = compteur + 1
    return compteur
```

1. Combien d'opérations coûte le test `if t[i] > 0` (accès + comparaison) ?
2. Dans le pire cas, quel est le coût complet d'une itération (test + branche la plus chère) ?
3. Donner le coût total dans le pire cas, en fonction de n.
4. Quelle est la complexité de cet algorithme ?

Ordres de grandeur

Dans les sections précédentes, on a calculé $T(n)$: une formule exacte du nombre d'opérations. En pratique, on ne s'intéresse qu'à son **ordre de grandeur** lorsque n devient grand, noté $O(f(n))$.

De $T(n)$ à O — règles de simplification

- On **ignore les constantes multiplicatives** : $3n$ et $7n$ donnent tous deux $O(n)$.
- On **ne conserve que le terme dominant** : $n^2 + 3n + 5$ donne $O(n^2)$.

On note $O(f(n))$ (lire « grand O de $f(n)$ ») la classe de complexité obtenue.

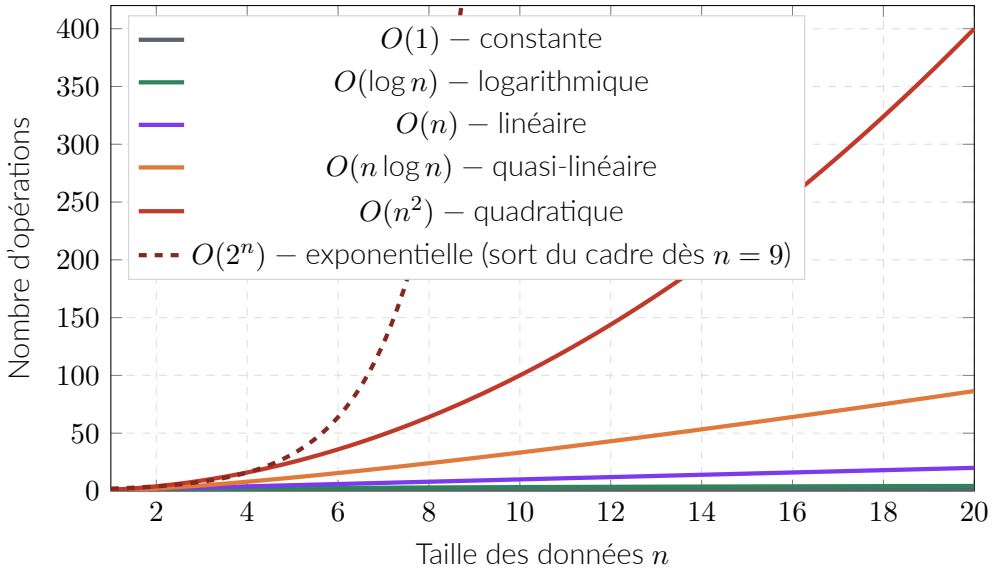
$$
\begin{aligned}
T(n) = 7 \quad &\longrightarrow \quad O(1) \quad \text{(constante)} \\
T(n) = 3n + 2 \quad &\longrightarrow \quad O(n) \quad \text{(linéaire)} \\
T(n) = 4n - 1 \quad &\longrightarrow \quad O(n) \quad \text{(linéaire)} \\
T(n) = 3n^2 + 4n \quad &\longrightarrow \quad O(n^2) \quad \text{(quadratique)}
\end{aligned}
$$

Les principales classes de complexité, du plus rapide au plus lent :

Notation	Nom	Exemple typique
$O(1)$	Constante	Accès à un élément de liste
$O(\log n)$	Logarithmique	Recherche dichotomique
$O(n)$	Linéaire	Recherche séquentielle, calcul de somme
$O(n \log n)$	Quasi-linéaire	Tri fusion
$O(n^2)$	Quadratique	Tri par sélection, tri par insertion
$O(2^n)$	Exponentielle	Exploration exhaustive

Voici le nombre d'opérations pour différentes valeurs de n :

n	$O(1)$	$O(\log_2 n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	1	3	10	100	1 024
100	1	7	100	10 000	$\approx 10^{30}$
1 000	1	10	1 000	1 000 000	hors de portée
10 000	1	13	10 000	100 000 000	hors de portée



⚠ $O(n)$ vs $O(n^2)$ en pratique

Pour $n = 10\,000$, un algorithme $O(n^2)$ effectue **100 millions** d'opérations là où un $O(n)$ en effectue **10 000**. À raison d'un milliard d'opérations par seconde, l'un prend $0{,}01$ ms et l'autre **0,1 seconde**. Pour $n = 10^6$, l'écart devient des **jours** contre des **millisecondes**.

La complexité en espace

Ce cours s'est concentré sur la **complexité temporelle**, qui mesure le nombre d'opérations. Il existe une autre mesure tout aussi importante : la **complexité spatiale** (ou complexité en espace).

Complexité en espace

La **complexité en espace** mesure la quantité de **mémoire** utilisée par un algorithme en fonction de la taille de l'entrée n.

Par exemple, un algorithme qui crée une nouvelle liste de taille n a une complexité en espace de $O(n)$. Un algorithme qui ne crée que quelques variables supplémentaires (indépendamment de n) a une complexité en espace de $O(1)$.

Il existe souvent un **compromis temps/espace** : accélérer un algorithme peut nécessiter d'utiliser plus de mémoire (c'est par exemple le principe de la mémoïsation en programmation dynamique), et inversement.

Type de complexité	Ce qu'elle mesure	Ressource
Temporelle	Nombre d'opérations élémentaires	Processeur (temps CPU)
Spatiale	Quantité de mémoire utilisée	RAM