Complexité algorithmique

Résumé

Deux programmes peuvent résoudre le même problème, mais l'un met une seconde et l'autre une heure. Pourquoi ? Parce qu'ils n'utilisent pas le même algorithme, et certains algorithmes font beaucoup plus de travail que d'autres. La **complexité** est l'outil qui permet de mesurer ce travail : elle exprime comment le temps d'exécution évolue quand la taille des données augmente. Grâce à elle, on peut comparer des algorithmes *avant* de les programmer, et choisir le plus efficace. C'est l'une des notions les plus importantes en informatique.

Prérequis : Les boucles, Recherche dans une liste, Tris naïfs.

① Mémo

Définition 1 (Qu'est-ce que la complexité ?)

La complexité mesure les ressources (temps, mémoire) nécessaires à l'exécution d'un algorithme en fonction de la taille n des données.

- **Complexité temporelle** : nombre d'opérations élémentaires.
- **Complexité spatiale** : mémoire utilisée.
- On s'intéresse au **pire cas** (worst case), noté avec la notation O (« grand O »).

Propriété 1 (Classes de complexité courantes)

$O(1)$ Constante : accès à un élément par indice, opération arithmétique.
$O(\log n)$ Logarithmique : dichotomie.
$O(n)$ Linéaire : parcours d'une liste, recherche séquentielle.
$O(n \log n)$ Quasi-linéaire : tri fusion, tri rapide (en moyenne).
$O(n^2)$ Quadratique : tri par sélection, tri par insertion (pire cas), double boucle imbriquée.
$O(2^n)$ Exponentielle : force brute sur les sous-ensembles.

Comment déterminer la complexité ?

1. Identifier la variable n (taille des données).
2. Compter les boucles : une boucle simple de 0 à n donne $O(n)$, deux boucles imbriquées donnent $O(n^2)$.
3. Une boucle qui divise n par 2 à chaque itération donne $O(\log n)$.
4. On ne garde que le terme dominant : $O(3n^2 + 5n + 7) = O(n^2)$.

5. Les constantes multiplicatives sont ignorées : $O(5n) = O(n)$.

Propriété 2 (Ordres de grandeur concrets (pour $n = 10^6$))

$O(\log n)$ ≈ 20 opérations : instantané.
$O(n)$ 10^6 opérations : une fraction de seconde.
$O(n \log n)$ $\approx 2 \times 10^7$ opérations : quelques secondes.
$O(n^2)$ 10^{12} opérations : plusieurs heures.
$O(2^n)$ Incalculable pour $n > 40$.

⚠ Erreurs classiques

Code erroné	Code correct	Explication
« Deux boucles imbriquées, donc $O(n^2)$ » systématiquement	Analyser les bornes réelles de chaque boucle	Si la boucle interne fait un nombre constant d'itérations (ex. : `range(5)`), la complexité reste $O(n)$. Seules des boucles *dépendant de n* contribuent.
$O(3n^2 + 5n) = O(3n^2)$	$O(3n^2 + 5n) = O(n^2)$	La notation O absorbe les constantes multiplicatives : $O(3n^2) = O(n^2)$. On ne garde que le terme dominant sans coefficient.
« Le tri par insertion est en $O(n^2)$ » sans nuance	Préciser : $O(n^2)$ dans le **pire cas**, $O(n)$ dans le meilleur cas	Certains algorithmes ont des complexités différentes selon les cas. Il faut toujours préciser : meilleur, moyen ou pire.
Oublier la complexité spatiale de la récursion	Chaque appel récursif consomme de la mémoire sur la pile	Une récursion de profondeur n utilise $O(n)$ en espace même si elle est en $O(n)$ en temps. C'est un coût caché.

② Exemples

Programme 1 · O(1) — Complexité constante

```
1 def premier_element(L):
2     return L[0]
```

Programme 2 · O(n) — Complexité linéaire

```python
def somme(L):
    total = 0
    for x in L:       # n itérations
        total += x    # O(1) par itération
    return total
# Total : n × O(1) = O(n)
```

Programme 3 · O(n²) — Complexité quadratique

```python
def doublons(L):
    """Vérifie s'il y a des doublons (version naïve)."""
    n = len(L)
    for i in range(n):             # n itérations
        for j in range(i + 1, n):  # jusqu'à n-1 itérations
            if L[i] == L[j]:
                return True
    return False
# Total : n(n-1)/2 comparaisons = O(n²)
```

Programme 4 · O(log n) — Complexité logarithmique

```python
def puissance_rapide(x, n):
    """Calcule x^n par exponentiation rapide."""
    resultat = 1
    while n > 0:
        if n % 2 == 1:
            resultat *= x
        x *= x
        n //= 2        # on divise n par 2 à chaque tour
    return resultat
# Nombre d'itérations : log(n)
```

Programme 5 · Comparer deux approches

```python
def contient_doublon_naif(L):       # O(n²)
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[i] == L[j]:
                return True
    return False

def contient_doublon_tri(L):        # O(n log n)
    L_triee = sorted(L)             # O(n log n)
    for i in range(len(L) - 1):     # O(n)
        if L_triee[i] == L_triee[i+1]:
            return True
```

```
13      return False
```

③ Exercices

Exercice 1 *Déterminer la complexité* Pour chaque fonction, déterminer la complexité en $O(\cdot)$ en justifiant :

```
def f1(n):
    s = 0
    for i in range(n):
        s += i
    return s
```

```
def f2(n):
    s = 0
    while n > 1:
        n = n // 2
        s += 1
    return s
```

```
def f3(L):
    s = 0
    for x in L:
        for y in L:
            if x == y:
                s += 1
    return s
```

Exercice 2 *Comparaison d'algorithmes* On dispose de deux algorithmes A et B pour résoudre un même problème de taille n.

- A effectue $100n$ opérations.
- B effectue n^2 opérations.

1. Pour quelle valeur de n les deux algorithmes effectuent-ils le même nombre d'opérations ?
2. Pour $n = 50$, lequel est le plus rapide ?
3. Pour $n = 1000$, lequel est le plus rapide ?
4. Justifier pourquoi A est asymptotiquement meilleur que B.

Exercice 3 *Optimiser un algorithme* La fonction suivante calcule le nombre d'éléments communs à deux listes.

```
def communs(L1, L2):
    compteur = 0
    for x in L1:
        for y in L2:
            if x == y:
                compteur += 1
    return compteur
```

1. Quelle est la complexité de cette fonction si `len(L1) = n` et `len(L2) = m` ?
2. Proposer une version plus efficace utilisant un ensemble (`set`).

Solutions des exercices

Corrigé de l'exercice 1.

1. `f1` : une seule boucle de 0 à $n-1$, soit n itérations. Complexité : $O(n)$.
2. `f2` : on divise n par 2 à chaque itération. Le nombre d'itérations est $\lfloor \log_2(n) \rfloor$. Complexité : $O(\log n)$.
3. `f3` : deux boucles imbriquées parcourant chacune la liste de taille n. Complexité : $O(n^2)$.

Corrigé de l'exercice 2.

1. $100n = n^2 \iff n^2 - 100n = 0 \iff n(n-100) = 0$. Comme $n > 0$, on a $n = 100$.
2. Pour $n = 50$: A fait 5 000 opérations, B fait 2 500. B est plus rapide.
3. Pour $n = 1000$: A fait 100 000 opérations, B fait 1 000 000. A est dix fois plus rapide.
4. A est en $O(n)$ et B en $O(n^2)$. Pour tout $n > 100$, A est plus rapide, et l'écart grandit avec n.
   La constante 100 de A est négligeable face à la croissance quadratique de B.

Corrigé de l'exercice 3.

1. La double boucle parcourt tous les couples (x, y). Complexité : $O(n \times m)$.

2. ```python
   def communs_v2(L1, L2):
       ensemble = set(L2)      # O(m)
       compteur = 0
       for x in L1:             # O(n)
           if x in ensemble:    # O(1) en moyenne (table de hachage)
               compteur += 1
       return compteur
   ```

   Complexité : $O(n + m)$ au lieu de $O(n \times m)$. Le test d'appartenance à un `set` est en $O(1)$ en moyenne grâce au hachage.
   ```