Langage et Algorithmique	Chap 03	La complexité d'un algorithme

3.1 Pourquoi parler de complexité ?

Comment mesurer l'efficacité d'un algorithme

Lorsqu'on conçoit un algorithme, une question naturelle se pose : **est-il efficace ?**
Deux algorithmes peuvent résoudre exactement le même problème, mais avec des temps d'exécution très différents dès que la taille des données augmente.
Quand on dit qu'un algorithme est **rapide** ou **lent**, on veut une mesure qui ne dépende pas :

- du modèle de processeur,
- du langage,
- de l'ordinateur utilisé,
- de l'optimisation du compilateur/interpréteur.

On cherche donc une mesure **abstraite**, basée sur le **nombre d'opérations** effectuées, en fonction de la **taille des données** (notée en général n).

Exemple 1 — Comparer deux algorithmes

On veut chercher un nombre dans une liste de n valeurs **déjà triée** :

- **Recherche séquentielle** : on regarde 1 à 1 (n comparaisons).
- **Recherche dichotomique** : on coupe en deux à chaque étape (environ $\log_2(n)$ comparaisons).

Même si une machine est 10 fois plus rapide, $\log_2(n)$ reste **beaucoup** plus petit que n quand n devient grand. On a donc accès à une mesure **objective** de l'efficacité d'un algorithme, qui ne dépend pas du temps d'éxécution.

3.2 Quest-ce que la complexité ?

Définition 1 — Complexité temporelle

La **complexité temporelle** d'un algorithme (et donc d'un programme) mesure le **nombre d'opérations** qu'il effectue en fonction de la taille de l'entrée n.
Elle permet d'estimer le **temps d'exécution** de l'algorithme, indépendamment de la machine ou du langage utilisé.

Définition 2 — Complexité spatiale

La **complexité spatiale** d'un algorithme mesure la **quantité de mémoire supplémentaire** utilisée en fonction de la taille de l'entrée n.
Elle prend en compte :

- les variables temporaires ;
- les structures de données auxiliaires ;
- la mémoire utilisée par les appels récursifs.

Temps contre mémoire

Un algorithme peut être :

- rapide mais gourmand en mémoire ;
- lent mais peu coûteux en mémoire ;
- ou chercher un compromis entre les deux.

Le choix d'un algorithme dépend donc à la fois du temps disponible et de la mémoire utilisable.
Dans le reste de ce chapitre, nous évoquons uniquement la complexité **temporelle**.

3.3 Taille de l'entrée et modèle de coût

Définition 3 — Taille d'entrée

Pour pouvoir avoir une mesure de la complexité d'un algorithme, il faut être en mesure de décrire "combien de données" il reçoit, on définit pour cela la **taille de l'entrée** (souvent notée n).

Exemple 2 — Tailles d'entrée

- Pour une liste : la taille de l'entrée $n =$ est le **nombre d'éléments**,
- Pour une chaîne de caractère : la taille de l'entrée $n =$ est le **nombre de caractères**,
- Pour une matrice (grille) de dimension $n \times n$: la taille de l'entrée $n =$ est souvent donné par **le nombre de cases**, donc n^2.

Définition 4 — Modèle d'opérations élémentaires

Afin de pouvoir comparer deux algorithmes différents de manière objective, on se donne un **modèle de coût**.
Cela consiste à décider quelles actions sont considérées comme **élémentaires**, c'est-à-dire ayant un coût constant, puis à compter combien de fois ces opérations sont exécutées en fonction de la taille de l'entrée. Les plus classiques sont :

- une affectation (`x = ...`),
- un accès à un élément (`tab[i]`),
- une comparaison (`<`, `>`, `==`),
- une opération arithmétique simple ($+, -, \times$),
- un test de condition (`if`).

Exercice 1 — Identifier la taille d'entrée

Pour chaque situation, proposer une taille d'entrée n pertinente.

1. Un algorithme qui assure la correction orthographique d'un livre

   ---

2. Un algorithme qui inverse la couleur d'une image de 1920×1080 pixels.

   ---

3. Un algorithme qui travaille sur un arbre binaire parfait de hauteur h.

3.4 Compter des opérations

Du code vers une fonction de coût

Pour étudier la complexité d'un algorithme, on cherche à estimer le **nombre d'opérations élémentaires** qu'il effectue en fonction de la taille de l'entrée, notée n.
On associe ainsi à l'algorithme une fonction $T(n)$, qui représente le nombre d'opérations nécessaires pour traiter une entrée de taille n. Dans un second temps, on simplifie cette fonction afin de ne conserver que son **ordre de grandeur** lorsque n devient grand.

Définition 5 — Complexité linéaire

On dit qu'un algorithme a une **complexité linéaire** lorsque le nombre d'opérations est proportionnel à la taille de l'entrée n.
Autrement dit, si n double, le nombre d'opérations double également. Ce type de comportement apparaît typiquement lorsqu'on parcourt une fois l'ensemble des données.

Exemple 3 — Boucle simple

Considérons :

```
s = 0
for i in range(n):
    s = s + 1
```

On effectue :

- une initialisation (`s = 0`),
- puis n fois l'instruction `s = s + 1`.

Donc $T(n) = n + 1$ ce qui **proportionnel** à n, la complexité du programme précédent est donc **linéaire**.

Exercice 2 — Compter et exprimer $T(n)$

On considère :

```
c = 0
for i in range(n):
    c = c + 1
    c = c + 1
```

1. Combien de fois la variable `c` est-elle incrémentée au sein d'un tour de boucle ?

---

2. Donner une expression du nombre total d'incrémentations en fonction de n.

---

3. Quelle est donc la complexité du programme précédent ?

Définition 6 — Croissance quadratique

On dit qu'un algorithme a une **complexité quadratique** lorsque le nombre d'opérations est proportionnel au carré de la taille de l'entrée, c'est-à-dire à n^2.
Ce comportement apparaît généralement lorsqu'on utilise deux boucles imbriquées parcourant chacune n éléments.

Exemple 4 — Boucles imbriquées

```
c = 0
for i in range(n):
    for j in range(n):
        c = c + 1
```

La ligne `c = c + 1` s'exécute $n \times n = n^2$ fois.
Donc $T(n) = n^2 + 1$, donc le programme est de complexité quadratique car $T(n)$ est de l'ordre de n^2.

Exercice 3 — Boucles imbriquées “triangle”

On considère :

```
c = 0
for i in range(n):
    for j in range(i):
        c = c + 1
```

1. Pour $n = 5$, compléter le tableau :

i	nombre d’itérations de la boucle en j
0	
1	
2	
3	
4	

2. En déduire le total d’incrémentations de c pour $n = 5$.

---

3. Donner une expression en fonction de n.

---

4. Conclure quant à la complexité du programme.

3.5 Meilleur cas, pire cas, cas moyen

Définition 7 — Meilleur cas, pire cas et cas moyen

Pour une taille d'entrée donnée n, le nombre d'opérations effectuées par un algorithme peut varier selon la configuration des données.
On distingue alors :

- le **meilleur cas** : nombre minimal d'opérations possibles pour une entrée de taille n ;
- le **pire cas** : nombre maximal d'opérations possibles pour une entrée de taille n ;
- le **cas moyen** : nombre moyen d'opérations sur l'ensemble des entrées possibles de taille n.

Ces trois mesures sont des fonctions du type $T(n)$, mais correspondent à des situations différentes.

Interprétation

- Le **meilleur cas** correspond à une situation particulièrement favorable, mais rarement représentative.
- Le **pire cas** donne une garantie : l'algorithme ne fera jamais plus d'opérations que cette valeur.
- Le **cas moyen** est souvent plus réaliste, mais plus difficile à définir car il dépend d'hypothèses sur les données.

Exemple 5 — Recherche séquentielle

On cherche une valeur `x` dans une liste `tab` de taille n.

- meilleur cas : `x` est au début $\Rightarrow$ 1 comparaison ;
- pire cas : `x` est à la fin ou absent $\Rightarrow$ n comparaisons.

Pourquoi privilégier le pire cas ?

En algorithmique, on s'intéresse souvent au pire cas car :

- il garantit un temps maximal d'exécution ;
- il permet de comparer des algorithmes sans hypothèse sur les données ;
- il évite les mauvaises surprises lorsque les données sont défavorables.

En ce qui nous concerne, la complexité est donc le plus souvent exprimée en **pire cas**.

Exercice 4 — Meilleur, pire et cas moyen : tester si une liste est triée

On considère l'algorithme suivant, qui teste si une liste `tab` de taille n est triée dans l'ordre croissant.

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

1. Donner le nombre de comparaison dans le **meilleur cas**.

   ---

2. Donner le nombre de comparaison dans le **pire cas**.

   ---

3. **Cas moyen (modèle simplifié).** On suppose que, pour une liste « au hasard », l'algorithme rencontre en moyenne la première inversion au milieu de la liste.

   Combien de comparaisons cela représente-t-il ?

3.6 Notation $O(\cdot)$: garder l'ordre de grandeur

Définition 8 — Notation $O(\cdot)$

Lorsqu'on étudie la complexité d'un algorithme, on ne cherche pas une expression exacte du nombre d'opérations, mais son **comportement global** lorsque la taille de l'entrée n devient grande.
La notation $O(\cdot)$ permet de décrire cet ordre de grandeur. Dire que $T(n)$ est en $O(f(n))$, ou $T(n) \in O(f(n))$, signifie, de manière informelle, que :

pour n suffisamment grand, le nombre d'opérations $T(n)$ ne dépasse pas une constante multipliée par $f(n)$.

On néglige donc les constantes et les termes de plus bas degré, afin de se concentrer uniquement sur la croissance dominante.

Simplifier des expressions mathématiques

Pour obtenir une forme en $O(\cdot)$, on applique en général :

- on **ignore les constantes** multiplicatives (ex : $3n$ et $100n$ sont du même ordre) ;
- on **ignore les termes de plus bas degré** (ex : $n^2 + 10n + 3$ est dominé par n^2).

Exemple 6 — Simplifications typiques

$$T(n) = 7n + 12 \quad \Rightarrow \quad T(n) \in O(n)$$

$$T(n) = 3n^2 + 2n + 100 \quad \Rightarrow \quad T(n) \in O(n^2)$$

$$T(n) = 5\log_2(n) + 200 \quad \Rightarrow \quad T(n) \in O(\log n)$$

Exercice 5 — Passer en $O(\cdot)$

Pour chaque fonction, donner une forme simplifiée en $O(\cdot)$.

1. $T(n) = 3n\log_2(n) + 50n$

---

2. $T(n) = n^2 + 5n\log_2(n) + 100$

---

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

---

4. $T(n) = n^2 + 10^6 n$

3.7 Ordres de grandeur classiques (à connaître)

Échelle des complexités

Quand n grandit, voici des croissances typiques (de la plus favorable à la moins favorable) :

$$O(1) \ < \ O(\log n) \ < \ O(n) \ < \ O(n \log n) \ < \ O(n^2) \ < \ O(n^3) \ < \ O(2^n)$$

Interprétation :

- $O(1)$: constant (ne dépend pas de n),
- $O(\log n)$: on “divise par 2” à chaque étape (dichotomie),
- $O(n)$: on parcourt une fois l’entrée,
- $O(n^2)$: double boucle sur n,
- $O(2^n)$: croissance exponentielle, qui dépasse très vite les capacités de calcul.

Exercice 6 — Classer des algorithmes

Associer chaque situation à un ordre de grandeur plausible ($O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$).

1. Compter le nombre de valeurs positives dans une liste.

---

2. Comparer toutes les paires d’éléments d’une liste (tester si deux éléments sont égaux).

---

3. Accéder à `tab[0]` dans une liste de taille n.

---

4. Chercher un élément dans une liste triée par dichotomie.
