Complexité algorithmique

Algorithmique

Questions fondamentales

Les trois questions fondamentales à se poser à propos de tout algorithme sont les suivantes :

— donne-t-il un résultat ou ne s'arrête-t-il jamais ? (Notion de terminaison) ;

— donne-t-il le résultat attendu ou calcule-t-il autre chose ? (Notion de correction) ;

— donne-t-il le résultat en un temps raisonnable ou faut-il attendre très longtemps ? (Notion de complexité).

Principe

Dans ce chapitre, nous nous intéressons au **coût en temps** d'un algorithme (et non au coût en espace mémoire). La question centrale est : combien de temps un algorithme met-il à s'exécuter ?

Ce temps dépend de la **taille des données** fournies en entrée. On peut facilement concevoir qu'un programme de recherche n'a pas le même temps d'exécution selon que l'on cherche un nom parmi les trente élèves d'une classe ou dans l'annuaire de toute une ville.

Les ordinateurs n'ayant pas tous la même vitesse d'exécution, il convient d'évaluer le coût **indépendamment de la machine**. On raisonne donc en terme de **nombre d'opérations élémentaires** : chaque opération simple (comparaison, affectation, opération arithmétique, accès à un élément, etc.) compte pour une étape de calcul.

Meilleur, pire et cas moyen

Pour une même taille de données, le temps d'exécution peut varier selon la configuration des données. On distingue :

— le **meilleur des cas** : temps minimal pour toutes les entrées de taille n ;

— le **cas moyen** : temps moyen pour toutes les entrées de taille n ;

— le **pire des cas** : temps maximal pour toutes les entrées de taille n.

Nous nous intéresserons principalement au **pire des cas**, car il fournit une garantie : l'algorithme ne sera jamais plus lent que cette borne.

Exemple. Considérons la recherche d'un élément dans une liste non triée de n éléments. Dans le meilleur des cas, l'élément est en première position (une seule comparaison). Dans le pire des cas, il est en dernière position ou absent (n comparaisons).

La notation O

La complexité ne retient qu'un **ordre de grandeur** du nombre d'opérations, en négligeant les constantes et les termes de moindre degré. On utilise la notation O (lire « grand O »), qui donne une borne supérieure du taux de croissance.

Règles pratiques :

— On ne garde que le terme dominant (celui qui croît le plus vite).

— On supprime les coefficients multiplicatifs.

Exemples :

— $4n + 3$ opérations $\to O(n)$ (on ignore le +3 et le coefficient 4)

— $2n^2 + 7n + 4$ opérations $\to O(n^2)$ (le terme n^2 domine)

— $\frac{n}{3}$ opérations $\to O(n)$ (le coefficient $\frac{1}{3}$ est ignoré)

Autrement dit, des algorithmes en n, $4n + 3$ ou $\frac{n}{3}$ opérations ont tous la même classe de complexité : $O(n)$.

Classes de complexité

Il est possible de catégoriser les algorithmes par classes de complexité. Voici les principales, rangées par ordre croissant de temps d'exécution :

Notation	Nom	Exemple typique
$O(1)$	Complexité constante	Accès à un élément d'une liste par indice
$O(\log n)$	Complexité logarithmique	Recherche dichotomique dans une liste triée
$O(n)$	Complexité linéaire	Parcours d'une liste, recherche séquentielle
$O(n \log n)$	Complexité quasi-linéaire	Tri fusion, le tri `sort()` de Python
$O(n^2)$	Complexité quadratique	Tri par sélection, tri par insertion
$O(2^n)$	Complexité exponentielle	Certains algorithmes sur les sous-ensembles
$O(n!)$	Complexité factorielle	Problème du voyageur de commerce (force brute)

Le logarithme $\log n$**.** Dans ce chapitre, $\log n$ désigne le logarithme de base 2 de n : c'est à peu près le nombre de fois qu'on peut diviser n par 2 avant d'atteindre 1. Par exemple, $\log 16 = 4$ car $16 = 2^4$, $\log 1\,024 = 10$ car $1\,024 = 2^{10}$, et $\log 1\,000\,000 \approx 20$. Un algorithme qui divise la taille de son problème par 2 à chaque étape effectue donc environ $\log n$ étapes.

Conséquences pratiques

Le tableau ci-dessous donne le temps approximatif d'exécution en fonction de la taille des données, en supposant qu'une opération prend 1 ns (10^{-9} seconde), soit un milliard d'opérations par seconde.

Taille (n)	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3 ns	10 ns	33 ns	100 ns	1 µs	3,6 ms
100	7 ns	100 ns	0,7 µs	10 µs	> âge de l'Univers	> âge de l'Univers
1 000	10 ns	1 µs	10 µs	1 ms	> âge de l'Univers	> âge de l'Univers
10 000	13 ns	10 µs	0,13 ms	0,1 s	> âge de l'Univers	> âge de l'Univers
100 000	17 ns	0,1 ms	1,7 ms	10 s	> âge de l'Univers	> âge de l'Univers
1 000 000	20 ns	1 ms	20 ms	17 min	> âge de l'Univers	> âge de l'Univers

L'âge de l'Univers est d'environ 14 milliards d'années. Avec seulement 100 éléments, un algorithme en 2^n demanderait environ 40 000 milliards d'années, soit près de 3 000 fois l'âge de l'Univers.

À retenir. Multiplier la taille des données par 10 :

— multiplie le temps par **10** si la complexité est linéaire $O(n)$;

— multiplie le temps par **100** si la complexité est quadratique $O(n^2)$;

— ajoute une constante au temps si la complexité est logarithmique $O(\log n)$;

— élève le temps à la puissance 10 si la complexité est exponentielle $O(2^n)$.

C'est pourquoi les algorithmes exponentiels sont considérés comme impraticables pour des données de grande taille.

Exemple guidé : compter les opérations

Recherche séquentielle

```python
def recherche(L, x):
    """Cherche x dans la liste L. Renvoie True si trouvé, False sinon."""
    for i in range(len(L)):
        if L[i] == x:
            return True
    return False
```

Comptons les opérations dans le **pire des cas** (l'élément n'est pas dans la liste, de longueur n) :

— La boucle `for` s'exécute n fois.

— À chaque tour : un accès à `L[i]` (une opération) et une comparaison `== x` (une opération), soit 2 opérations par tour.

— Total : $2n$ opérations.

La complexité au pire des cas est donc $O(n)$: **linéaire**.

Notes au-dessus d'un seuil

```python
def nb_au_dessus(notes, seuil):
    """Renvoie le nombre de notes strictement supérieures à seuil."""
    compteur = 0
    for note in notes:
```

```
        if note > seuil:
            compteur = compteur + 1
    return compteur

print(nb_au_dessus([12, 8, 15, 10, 17], 10))
```

```
3
```

La boucle s'exécute n fois. À chaque tour : une comparaison, puis, quand la note dépasse le seuil, une addition et une affectation. La boucle effectue donc entre n et $3n$ opérations selon les notes : dans tous les cas, la complexité est $O(n)$.

Exemple : organiser un tournoi

Dans un tournoi, chaque équipe rencontre une fois chacune des autres. La fonction suivante construit la liste de tous les matchs.

```
def rencontres(equipes):
    matchs = []
    for i in range(len(equipes)):
        for j in range(i):
            matchs.append((equipes[j], equipes[i]))
    return matchs

print(rencontres(["Lyon", "Paris", "Lille", "Brest"]))
```

```
[('Lyon', 'Paris'), ('Lyon', 'Lille'), ('Paris', 'Lille'),
 ('Lyon', 'Brest'), ('Paris', 'Brest'), ('Lille', 'Brest')]
```

Que fait cette fonction ? Pour chaque équipe d'indice `i`, la boucle interne lui fait rencontrer les équipes placées avant elle, d'indices 0 à `i - 1`. Chaque match n'apparaît donc qu'une seule fois : avec quatre équipes, on obtient six matchs.

Complexité. La boucle externe s'exécute n fois. Pour chaque valeur de i, la boucle interne s'exécute i fois. Le nombre total d'itérations de la boucle interne est donc :

$$0 + 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

On retrouve ce nombre autrement : chacune des n équipes rencontre les $n-1$ autres, ce qui fait $n(n-1)$ rencontres vues du côté des équipes ; or chaque match concerne deux équipes, il est donc compté deux fois. La complexité est $O(n^2)$: **quadratique**. Avec 20 équipes, il faut 190 matchs ; avec 40 équipes, 780 matchs, soit environ quatre fois plus.

Exercices

Exercice 1 Pour chaque expression, donner la complexité de référence correspondante :

1. $5n^2 + 2n - 7 = O(\ldots)$

2. $2n^3 + 4n^2 + 1 = O(\ldots)$
3. $\frac{1}{2}n^5 + 4n^3 + 6n = O(\ldots)$
4. $7 + 2n\log(n) = O(\ldots)$
5. $5n^4 + 2^n + n\log(n) = O(\ldots)$
6. $3^n + n! + 4n^3 = O(\ldots)$

Exercice 2 Déterminer la complexité au pire des cas de chacune des fonctions suivantes.

a) Calcul du produit des éléments d'une liste :

```
def produit(L):
    p = 1
    for x in L:
        p = p * x
    return p
```

b) Recherche de deux éléments de somme nulle :

```
def somme_nulle(L):
    n = len(L)
    for i in range(n):
        for j in range(i + 1, n):
            if L[i] + L[j] == 0:
                return True
    return False
```

c) Nombre de chiffres de l'écriture binaire d'un entier $n \geq 1$:

```
def nb_bits(n):
    compteur = 1
    while n > 1:
        n = n // 2
        compteur += 1
    return compteur
```

Exercice 3 On souhaite mesurer expérimentalement la complexité de la fonction `somme_nulle` de l'exercice 2b.

1. Utiliser le module `time` pour mesurer le temps d'exécution sur des listes de tailles 250, 500, 1000 et 2000 (sans paire de somme nulle, pour être dans le pire des cas).
2. Vérifier que lorsque la taille est multipliée par 2, le temps est approximativement multiplié par 4 (caractéristique d'une complexité quadratique).

```
import time

def somme_nulle(L):
    n = len(L)
    for i in range(n):
        for j in range(i + 1, n):
            if L[i] + L[j] == 0:
```

```
                return True
    return False

for taille in [250, 500, 1000, 2000]:
    L = list(range(1, taille + 1))  # nombres positifs : aucune somme nulle
    debut = time.perf_counter()
    somme_nulle(L)
    fin = time.perf_counter()
    print(f"n = {taille:>4} : {fin - debut:.4f} s")
```

Exercice 4 La fonction suivante renvoie le plus grand écart entre deux éléments d'une liste. Sa complexité est $O(n^2)$.

```
def ecart_max(L):
    n = len(L)
    ecart = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(L[i] - L[j]) > ecart:
                ecart = abs(L[i] - L[j])
    return ecart
```

Proposer un algorithme plus efficace, en $O(n)$, pour résoudre le même problème.

Exercice 5 Un algorithme en $O(n^2)$ met 0,2 seconde à traiter 2 000 éléments.

1. Combien de temps mettra-t-il approximativement pour 20 000 éléments ?
2. Quelle taille de données peut-il traiter en moins d'une minute ?