

Institut Supérieur du Numérique (SupNum)



Structures de données et complexité algorithmique

Semestre : S3



Dr. KHADHRAOUI Mayara

2025/2026

PLAN

- Introduction, rappels
- Notions de complexité
- Algorithmes de tri
- Pointeurs et structures en C++
- Liste chaînées / Piles / Files
- Arbres binaires
- Tables de hachage

Introduction

- ◆ Dans le domaine de l'algorithmique, il est courant de trouver **plusieurs algorithmes** capables de résoudre le même problème.
- ◆ Cependant, tous **ne sont pas équivalents**: certains sont plus rapides, d'autres consomment moins de mémoire, ou réagissent mieux à de grandes quantités de données.
- ◆ Face à cette diversité, il devient essentiel de pouvoir **comparer ces algorithmes** de manière objective.
- ◆ C'est là qu'intervient la notion de **complexité**, un outil fondamental pour évaluer les performances et faire des choix éclairés en fonction du contexte.

COMPLEXITE

Définition :

La complexité mesure l'efficacité d'un algorithme — c'est-à-dire la quantité de temps et de mémoire qu'il utilise pour résoudre un problème.

Pourquoi l'évaluer ?

- Réduire les coûts de traitement :
 -  En temps (exécution plus rapide)
 -  En ressources (mémoire optimisée)
- Permet de s'attaquer à des problèmes plus grands.

◆ Deux types principaux de complexité :

- 1 Complexité temporelle → Temps d'exécution
- 2 Complexité spatiale → Mémoire utilisée

COMPLEXITE

- On suppose que chaque opération élémentaire coûte $1\mu s$

Taille/ Comporte- ment	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$0,01\ s$	10^{14} siècles
1000	$10\mu s$	$1000\mu s$	$0,01\ s$	$1\ s$	astronomique
10000	$13\mu s$	$0,01s$	$\frac{1}{7}s$	$1,7\ min$	astronomique

👉 Ce tableau montre que plus la complexité d'un algorithme croît rapidement (de $\log n$ à 2^n), plus le temps d'exécution explose quand la taille des données augmente.

COMPLEXITE

Notation à utiliser

- A : un algorithme
- D : l'ensemble auquel appartient les données d'entrée de A
- $N_A(d)$: désigne le nombre d'instructions élémentaires effectués par A pour calculer sa sortie lorsque la donnée d est donnée en entrée
- Pour chaque opération élémentaire i du processeur le temps nécessaire est t_i
- On prend $t_+ = 1, t_- = 1, t_x = 2$

Notation à utiliser

Objectif :

Présenter les notations employées pour analyser la complexité d'un algorithme.

- A : un algorithme
- D : l'ensemble des données d'entrée possibles de l'algorithme A
- $N_a(d)$: le nombre d'instructions élémentaires exécutées par A pour calculer le résultat à partir de la donnée d
- Chaque opération élémentaire i du processeur nécessite un temps t_i
- Le temps total d'exécution dépend :
 - du nombre d'instructions effectuées ($N_a(d)$)
 - du temps d'exécution de chaque instruction (t_i)

Exemple de temps élémentaires

Exemple de quelques opérations :

- Addition : $t_+ = 1$ unité
- Soustraction : $t_- = 1$ unité
- Multiplication : $t_x = 2$ unités
- Ces notations permettent de quantifier le coût d'un algorithme
- On distingue :
 - le coût en nombre d'opérations
 - le coût en temps total d'exécution
- Elles sont utilisées pour étudier la complexité temporelle d'un algorithme.

COMPLEXITE

Quelques règles

Règle 1 : Structure séquentielle

- Si l'on considère l'algorithme $T_1; T_2$; où T_1 et T_2 sont des traitements, alors on a les relations suivantes.
 - $N_{T_1;T_2}(d) = N_{T_1}(d) + N_{T_2}(d)$

Par exemple :

- $I \leftarrow I + 1$
- $I \leftarrow I + 2$

La première instruction réalise une affectation et une addition. Pareil pour la deuxième instruction. Alors l'algorithme réalise 2 affectations et 2 additions.

Règle 1 : Structure séquentielle



Principe :

Lorsqu'on exécute deux traitements T_1 et T_2 de manière séquentielle, le nombre total d'instructions est la somme des instructions de chacun.

Formule : $N_{\{T_1; T_2\}}(d) = N_{\{T_1\}}(d) + N_{\{T_2\}}(d)$

◆ Exemple :

$i \leftarrow i + 1$

$i \leftarrow i + 2$

- Chaque instruction réalise une addition et une affectation.

Règle 1 : Structure séquentielle

- L'ensemble des deux instructions réalise donc :
 - 2 additions
 - 2 affectations



Conclusion :

Dans une structure séquentielle, les coûts s'additionnent.

COMPLEXITE

Quelques règles

Règle 2 : Structure alternative

- Si l'on considère l'algorithme :

Si B Alors T_1 Sinon T_2

Alors on a la relation suivante :

- $N_{\text{Si B Alors } T_1 \text{ Sinon } T_2}(d) = N_B(d) + N_{T_1}(d)$ si B était vrai. Ou bien
- $N_{\text{Si B Alors } T_1 \text{ Sinon } T_2}(d) = N_B(d) + N_{T_2}(d)$ si B était faux.

Par exemple :

- $I \leftarrow 2$
- **Si $(I > 3)$ Alors $I \leftarrow I + 3$ Sinon $I \leftarrow 4$**

L'algorithme effectuera une comparaison et une affectation.

Règle 2 : Structure alternative — Principe et Formule



Principe :

La structure alternative permet à un algorithme de choisir entre deux traitements selon que la condition est vraie ou fausse.

On considère l'algorithme :

Si B Alors T_1 Sinon T_2

Formules :

- Si B est vraie : $N_{\{\text{SiBAlors}T_1\text{Sinon}T_2\}}(d) = N_B(d) + N_{\{T_1\}}(d)$
- Si B est fausse : $N_{\{\text{SiBAlors}T_1\text{Sinon}T_2\}}(d) = N_B(d) + N_{\{T_2\}}(d)$

Règle 2 : Structure alternative — Principe et Formule

 Interprétation :

- On évalue la condition B (comparaison)
- Puis on exécute seulement une des deux branches
- Le coût dépend du résultat logique de la condition

Règle 2 : Structure alternative — Exemple et Interprétation

◆ Exemple d'algorithme :

$i \leftarrow 2$

Si $(i > 3)$ Alors $i \leftarrow i + 3$ Sinon $i \leftarrow 4$

🧩 Analyse pas à pas :

1. Affectation initiale : $i \leftarrow 2$
2. Comparaison : $i > 3 \rightarrow$ faux
3. Exécution de la branche 'Sinon' : $i \leftarrow 4$

⚙️ Opérations effectuées :

- 1 affectation
- 1 comparaison
- 1 affectation supplémentaire (selon la condition)

Règle 2 : Structure alternative — Exemple et Interprétation

Conclusion :

- L'algorithme effectue une seule des deux actions possibles.
- On compte le coût du test + le coût de la branche exécutée.
- Cette structure exprime une décision conditionnelle dans un algorithme.

COMPLEXITE

Quelques règles

Règle 3 : Structure répétitive

- Si l'on considère l'algorithme :
Tant que B Faire S

Alors on a la relation suivante :

- $N_{\text{TantqueBFaireS}}(d) = \sum_{i=1}^k [N_B(d) + N_{S(i)}(d)] + N_B(d)$

Où k le nombre d'itérations de la boucle et $S(i)$ représente le traitement effectué lors de l'itération i . D'une manière générale, le nombre d'instruction effectué dans une boucle outre que Tantque est donné par

$$\sum_{i=1}^k N(i)$$

Où $N(i)$ est le nombre d'instruction réalisées lors de l'itération i . On y rajoute l'initialisation des indices, leur incrémentation et le test de fin.

Règle 3 : Structure répétitive

- La structure répétitive (ou boucle) permet d'exécuter plusieurs fois une suite d'instructions tant qu'une condition est vraie.
- Algorithme général : Tant que B Faire S
 - B : condition logique (booléenne)
 - S : séquence d'instructions exécutée à chaque itération
- Cette structure permet de répéter un traitement jusqu'à ce qu'une condition ne soit plus satisfaite.

Relation mathématique et explication

- Le nombre total d'instructions exécutées par la boucle est :
$$N\_TantqueB\_FaireS(d) = \sum [N\_B(d) + N\_S(i)(d)] + N\_B(d)$$
- Où :
 - k : nombre d'itérations
 - $N\_B(d)$: instructions pour évaluer la condition B
 - $N\_S(i)(d)$: instructions du corps de la boucle à l'itération i
- En général : Nombre total = $\sum N(i)$
 - On ajoute aussi l'initialisation, l'incrémentation et le test final.
- Cette relation permet d'évaluer la complexité d'un algorithme contenant une boucle.

COMPLEXITE

Exemple

- $I \leftarrow 0$
- Tant que $I < N$ Faire $I \leftarrow I + 1$

Combien d'instructions si $N = 10$?

Pour $N = 10$, le corps de l'algorithme s'exécute 10 fois, le test est effectué 11 fois. L'algorithme effectue donc 10 additions, 10 affectations (+1 pour l'initialisation), et 11 comparaisons.

Exemple – Structure répétitive : Boucle Tant que

Algorithme :

$i \leftarrow 0$

Tant que $i < N$ Faire

$i \leftarrow i + 1$

- La variable i commence à 0.
- La boucle s'exécute tant que la condition $i < N$ est vraie.
- À chaque itération, i augmente de 1.
- Quand i atteint N , la boucle s'arrête.

Analyse du comportement de la boucle

- La condition $i < N$ est testée à chaque itération.
- Le corps de la boucle s'exécute un certain nombre de fois.
- Exemple : pour $N = 10$, la boucle s'exécute 10 fois (de $i = 0$ à $i = 9$).
- Après la 10^e exécution, la condition devient fausse et la boucle s'arrête.

Combien d'instructions si $N = 10$?

- Le corps de la boucle s'exécute 10 fois.
 - Le test $i < N$ est effectué 11 fois (une fois de plus à la fin).
 - L'algorithme effectue donc :
 - 1 affectation initiale ($i \leftarrow 0$)
 - 10 additions ($i + 1$)
 - 10 affectations ($i \leftarrow i + 1$)
 - 11 comparaisons ($i < N$)
- L'algorithme exécute 10 tours de boucle mais réalise 11 tests au total.

Combien d'instructions si $N = 10$?

→ En **analyse de complexité**, on compte parfois *chaque instruction* pour estimer le coût total.

→ Donc il faut bien noter :

Une boucle qui s'exécute N fois effectue souvent $N + 1$ tests de condition.

COMPLEXITE

Analyse exacte d'algorithme

Calculer exactement le coût: pas facile !

- Rappelons que la complexité = fonction de la taille des données
- Mais cette fonction peut être très difficile à calculer de façon exacte
- Il vaut mieux parler de **bornes** (supérieure et inférieure) pour cette fonction
- $\Rightarrow$ Notations de Landau: O , Ω , Θ
- **Intérêts :**
 - les constantes sont ignorées
 - on ne garde que le **terme dominant** de la fonction

Analyse exacte d'un algorithme

- La complexité d'un algorithme est une fonction de la taille des données.

👉 Cela signifie que **le temps ou les ressources nécessaires** pour exécuter un algorithme **dépendent de la taille des données qu'on lui donne en entrée.**

Exemple simple :

Si tu as un algorithme qui trie une liste de nombres :

Trier 10 nombres prendra peu de temps.

Trier 1 million de nombres prendra beaucoup plus de temps.

👉 **Plus la taille des données augmente, plus l'algorithme met du temps ou consomme de la mémoire.**

Analyse exacte d'un algorithme

En informatique, on exprime cela avec la complexité algorithmique, par exemple :

$O(n) \rightarrow$ le temps augmente proportionnellement au nombre d'éléments.

$O(n^2) \rightarrow$ le temps augmente comme le carré du nombre d'éléments.

$O(\log n) \rightarrow$ le temps augmente lentement même si les données augmentent.

$O(n)$ – Croissance linéaire

Principe : Le temps augmente **proportionnellement** au nombre d'éléments.

Exemple :

- Si ton programme lit **10** éléments, il prend **10** secondes.
- Si tu doubles à **20** éléments, il prendra environ **20** secondes.



Image suggérée : Un ouvrier qui classe des papiers un à un.

$O(n^2)$ – Croissance quadratique

Principe : Le temps augmente comme le carré du nombre d'éléments.

Exemple :

- Si chaque élève doit comparer sa réponse avec tous les autres.
 - 10 élèves $\rightarrow$ 100 comparaisons ; 20 élèves $\rightarrow$ 400 comparaisons.
-  Image suggérée : Des élèves qui se comparent tous entre eux.

$O(\log n)$ – Croissance logarithmique

Principe : Le temps augmente très lentement même si les données augmentent beaucoup.

Exemple :

- Comme quand tu cherches un mot dans un dictionnaire en divisant les pages en deux à chaque fois.
- 1000 pages → seulement une dizaine d'étapes pour trouver le mot.

 Image suggérée : Une personne qui cherche un mot dans un dictionnaire.

Analyse exacte d'un algorithme

En résumé :

Plus les **données en entrée** sont grandes, plus l'**algorithme peut devenir lent ou gourmand**. C'est ce que veut dire "la complexité est une fonction de la taille des données".

- La complexité mesure le coût en temps ou en mémoire.
- Mais calculer cette fonction exactement est souvent très difficile.

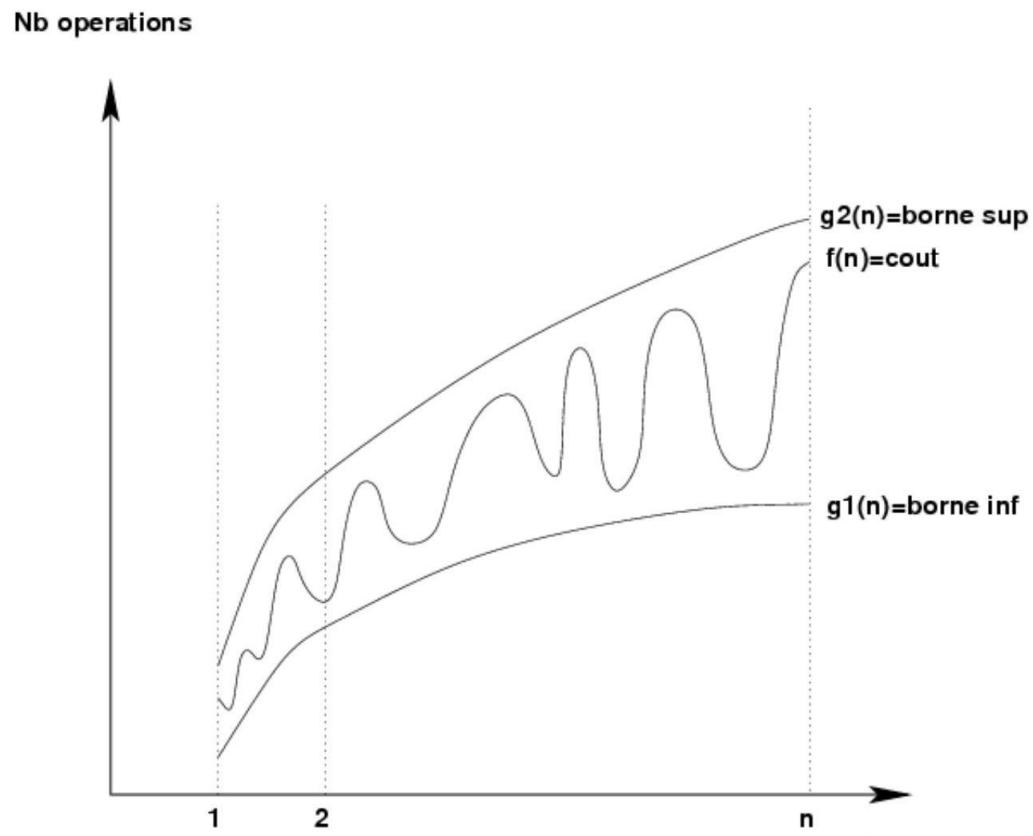
 Objectif : comprendre pourquoi on préfère utiliser des bornes.

Approche par bornes

- Au lieu d'une mesure exacte, on parle de bornes :
 - Borne supérieure
 - Borne inférieure
 - Ces bornes permettent de mieux caractériser la croissance de la complexité.
- 👉 On cherche donc une manière plus simple d'évaluer la performance.

COMPLEXITE

Bornes inférieure et supérieure



Une idée : utiliser des bornes

- Au lieu de calculer le coût exact, on estime :
 - Une borne **supérieure** (le pire des cas)
 - Une borne **inférieure** (le meilleur des cas)
- Ces bornes donnent une idée générale du comportement de l'algorithme sans entrer dans les détails compliqués.

👉 On utilise pour cela des notations spéciales appelées notations de **Landau**.

Notation O (Grand O)

Notation O (Grand O)

- Représente une borne supérieure : **le pire des cas**.
- L'algorithme ne dépasse pas une certaine croissance.
- Exemple : $T(n) = 3n^2 + 2n + 1 \rightarrow O(n^2)$.

Notation Ω (Grand oméga)

- Représente une borne inférieure : **le meilleur des cas**.
- L'algorithme prend au moins autant de temps que $f(n)$.
- Exemple : $T(n) = 3n^2 + 2n + 1 \rightarrow \Omega(n^2)$.

Notation Θ (Thêta)

- Représente une borne exacte (**supérieure et inférieure**).
- L'algorithme croît exactement comme $f(n)$.
- Exemple : $T(n) = 3n^2 + 2n + 1 \rightarrow \Theta(n^2)$.

Les notations de Landau : O , Ω , Θ

- Ces notations permettent de simplifier la fonction de complexité.
 - On ignore les constantes (par exemple $2n$ ou $5n$ deviennent $O(n)$).
 - On garde seulement le terme le plus important (le terme dominant).
-  Cela permet de comparer facilement deux algorithmes et de savoir lequel est plus efficace.

Règle 1 : Les constantes sont ignorées

- Le temps d'exécution est proportionnel, les constantes ne changent pas la croissance.

Exemples :

– $T(n) = 5n + 10 \rightarrow O(n)$

– $T(n) = 0.001n^2 + 3n + 20 \rightarrow O(n^2)$

💡 Les coefficients et les additions constantes n'affectent pas l'ordre de grandeur.

Règle 2 : On garde seulement le terme dominant

- Quand il y a plusieurs termes, seul celui qui croît le plus vite avec n compte.

Exemples :

$$- T(n) = \mathbf{n^3} + 100n^2 + 50n + 10 \rightarrow O(n^3)$$

$$- T(n) = \mathbf{2n^2} + 10n \log n + 500 \rightarrow O(n^2)$$



Le terme dominant détermine la complexité **asymptotique**.

Règle 2 : On garde seulement le terme dominant

Définition :

La **complexité asymptotique** décrit comment le temps ou la mémoire d'un algorithme évoluent lorsque la taille du problème devient très **grande**.

Objectif :

Ignorer les **détails mineurs** (constantes, petits termes) pour se concentrer sur la tendance générale de la croissance.

Exemple :

Si $T(n) = 3n^2 + 5n + 2$, alors pour de grandes valeurs de n , le terme dominant est $3n^2$, donc la **complexité asymptotique** est $O(n^2)$.

Règle 2 : On garde seulement le terme dominant



En résumé :

« Asymptotique » signifie observer le comportement à long terme de l'algorithme quand **n devient très grand**.



Avantage : simplifie la comparaison entre algorithmes

COMPLEXITE

Notations de Landau

Idée: $f(n)$ est le coût de l'algorithme, $g(n)$ est une fonction plus simple à manipuler

$O()$: définition intuitive

- Soit deux fonctions $f(n)$ et $g(n)$
- $f(n) = O(g(n))$ veut dire que $c \times g(n)$ est une **borne supérieure** pour $f(n)$
- c est une constante (indépendante de n)
- On s'intéresse à des **grandes valeurs de n** $\Rightarrow$ on supposera que n dépasse une valeur n_0

$\Rightarrow g()$ **domine** $f()$ à une constante multiplicative c près

L'idée générale

$f(n)$ = le **coût réel** de l'algorithme quand la taille des données est n .

$g(n)$ = une **fonction plus simple** qu'on utilise pour **approximer le comportement** de $f(n)$.

Exemple 1 : un algorithme simple

♦ Étape 1 : calcul du coût

Il y a une boucle qui s'exécute **n fois**.

À chaque tour, on fait une opération (afficher un nombre).

👉 donc le coût total :

$$f(n) = n$$

♦ Étape 2 : fonction de comparaison

On prend une fonction simple

$$g(n) = n$$

♦ Étape 3 : relation Big O

On a clairement :

$$f(n) = n = 1 \times g(n)$$

Donc :

$$f(n) = O(n)$$

👉 Le temps d'exécution augmente proportionnellement à la taille du problème.

Exemple 2 : deux boucles imbriquées

◆ Étape 1 : coût

La première boucle tourne **n fois**.

La deuxième boucle tourne **n fois** pour chaque tour de la première.

👉 donc le coût total :

$$f(n) = n \times n = n^2$$

◆ Étape 2 :

$$g(n) = n^2$$

Alors :

$$f(n) = O(n^2)$$

● **Interprétation** : si on double la taille des données, le temps est multiplié par 4.

Exemple 3 : un cas plus “réaliste”

Supposons que le coût d'un algorithme soit :

$$f(n) = 3n^2 + 5n + 10$$

et qu'on choisit :

$$g(n) = n^2$$

Pour des grandes valeurs de n , le terme $3n^2$ **domine** les autres.
Les termes $5n$ et 10 deviennent négligeables.

Donc :

$$f(n) \leq 4n^2 \text{ pour } n > 10 \Rightarrow f(n) \leq 4 * g(n) \text{ pour } n > 10$$

ce qui signifie :

$$f(n) = O(n^2)$$

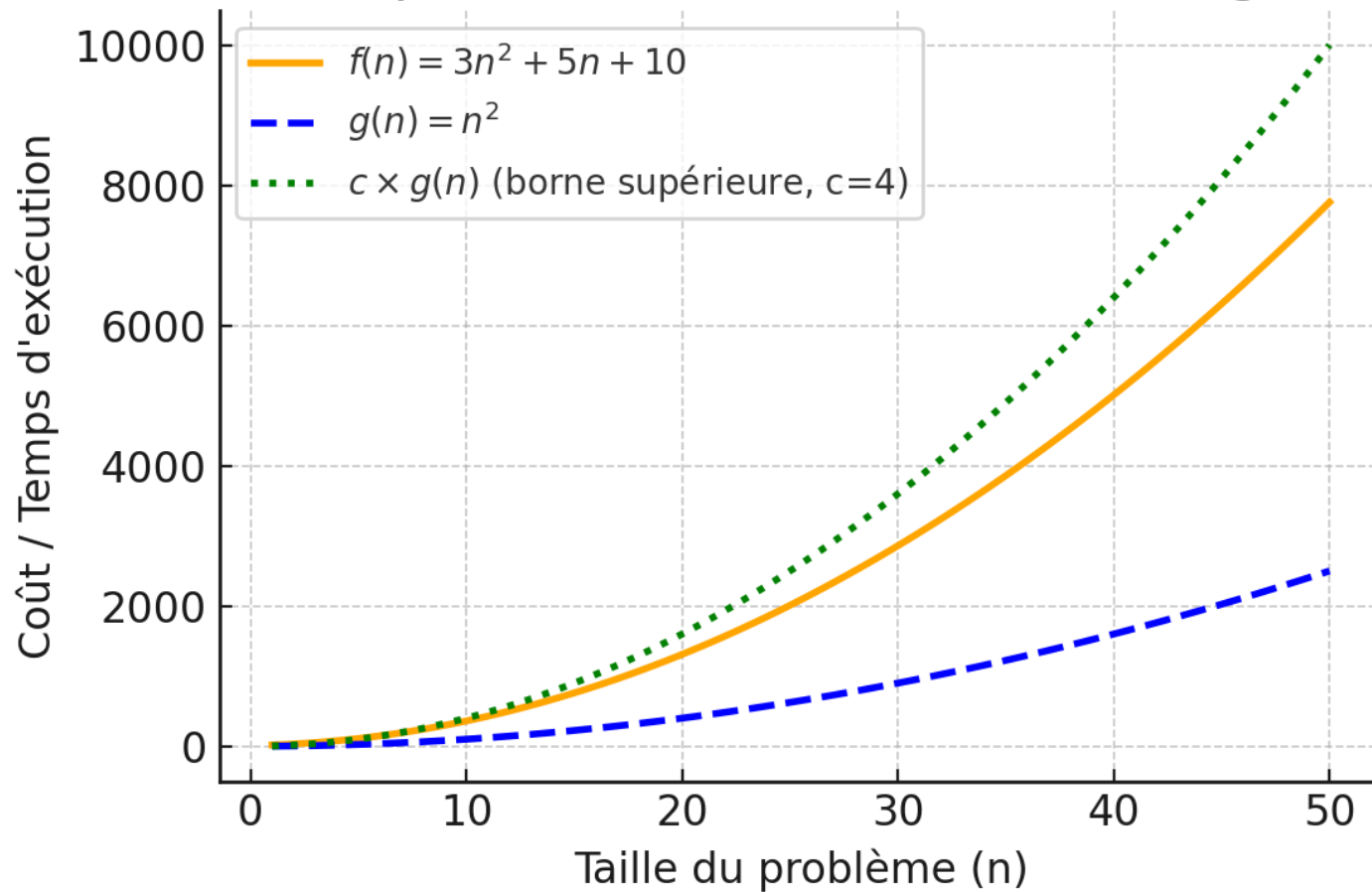
 **Interprétation** : à long terme, $f(n)$ se comporte comme une fonction quadratique.

Résumé

Exemple	$f(n)$ (coût réel)	$g(n)$ (simplifié)	Big O
Une boucle	n	n	$O(n)$
Deux boucles imbriquées	n^2	n^2	$O(n^2)$
Fonction avec plusieurs termes	$3n^2 + 5n + 10$	n^2	$O(n^2)$

Résumé

Exemple visuel : $f(n)$ et sa borne $O(g(n))$



COMPLEXITE

Notation O

$$f(n) = 3n^2 - 100n + 6$$

1. Les affirmations suivantes sont-elles correctes ?

- $f(n) = O(n^3)$
- $f(n) = O(n^2)$
- $f(n) = O(n)$

2. Quel commentaire vous inspire $f(n) = O(n^3)$?

Rappel : Notation Big O

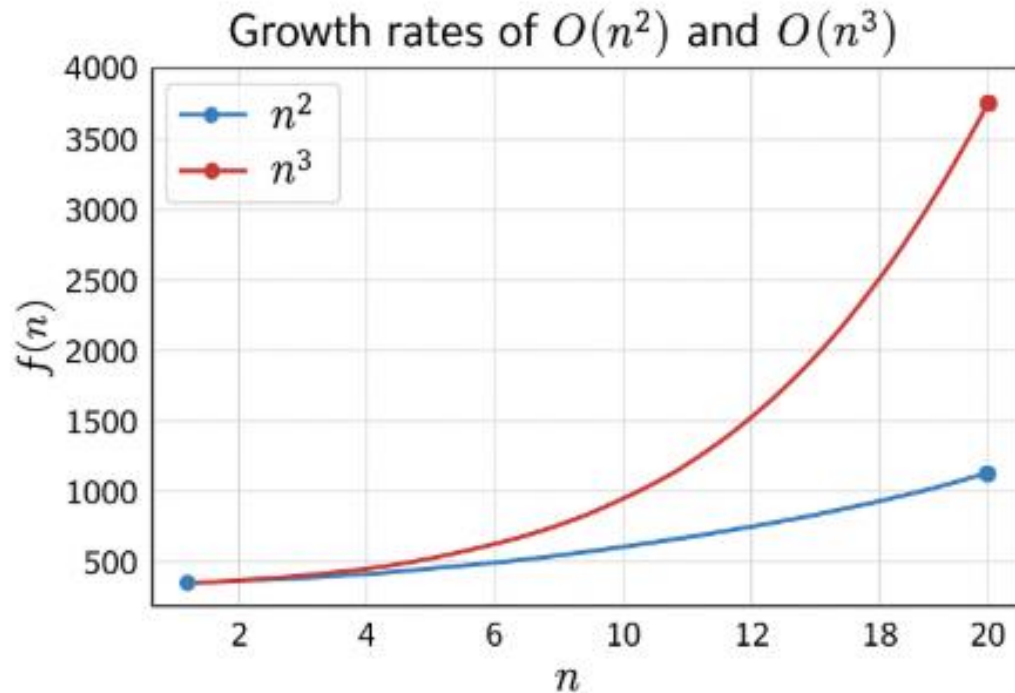
La notation O (Big O) décrit la croissance asymptotique d'une fonction.

Exemple : $f(n) = 3n^2 - 100n + 6$

→ On cherche la borne supérieure de la complexité en fonction de n .

Analyse des affirmations

1. $f(n) = O(n^3) \rightarrow$ ✓ Vrai, car $n^2 \leq n^3$ pour n grand
2. $f(n) = O(n^2) \rightarrow$ ✓ Vrai, le terme dominant est $3n^2$
3. $f(n) = O(n) \rightarrow$ ✗ Faux, car n^2 croît plus vite que n



Analyse des affirmations

1. $f(n) = O(n^3) \rightarrow$ ✓ Vrai, car $n^2 \leq n^3$ pour n grand
2. $f(n) = O(n^2) \rightarrow$ ✓ Vrai, le terme dominant est $3n^2$
3. $f(n) = O(n) \rightarrow$ ✗ Faux, car n^2 croît plus vite que n

👉 Donc, la complexité réelle est $O(n^2)$.

Dire que $f(n) = O(n^3)$ est correct mathématiquement, mais pas précis : on donne une borne trop large.

➡ En algorithmique, on cherche la borne la plus serrée :
ici, $f(n) = O(n^2)$ est la meilleure estimation.

COMPLEXITE

Fonctions

- **Exponentielle** : toute fonction en $O(c^n)$ (c =constante, $c > 1$)
 - **Polynomiale** : toute fonction en $O(n^{c'})$ (c' =constante)
 - **Logarithmique** : toute fonction de la forme $O(\log n)$
-
- Toute fonction exponentielle **domine** toute fonction polynomiale
 - Toute fonction polynomiale **domine** toute fonction logarithmique
 - En ce qui concerne le coût des algorithmes, **on évitera tout ce qui exponentiel**

Les types de fonctions en algorithmique

a) Fonction exponentielle

Forme : $O(c^n)$ avec $c > 1$

Exemple : $2^n, 3^n$, etc.

Cela veut dire que si on augmente légèrement n , le temps explose très vite.

👉 Exemple : si ton algorithme prend 1 seconde pour $n = 10$, il peut prendre des **années** pour $n = 100$.

⚠ C'est **très lent** quand n devient grand.

⚠ On essaie donc de **l'éviter**.

Les types de fonctions en algorithmique

b) Fonction polynomiale

Forme : $O(n^{c'})$ avec c' une **constante** (comme 2, 3, etc.)

Exemple : n^2 , n^3 , etc.

C'est beaucoup plus **raisonnable** que l'exponentiel.

👉 Si n double, le temps ne fait pas une explosion totale, il augmente modérément.

👉 C'est souvent ce qu'on **accepte** en pratique : les algorithmes polynomiaux sont considérés comme « efficaces ».

Les types de fonctions en algorithmique

c) Fonction logarithmique

Forme : $O(\log n)$

Exemple : recherche dichotomique (binary search): exemple du dictionnaire.

C'est **très rapide** : même si n devient grand, le temps augmente très lentement.

👉 La recherche dichotomique est rapide **parce qu'elle réduit le problème de moitié à chaque étape**, et le temps de recherche croît **très lentement** même si la liste devient énorme.

Ordre de domination des fonctions

👉 Ces notations expriment la complexité d'un algorithme en fonction de la taille de l'entrée n .

- Toute fonction **exponentielle** domine toute fonction **polynomiale**.
- Toute fonction **polynomiale** domine toute fonction **logarithmique**.

Autrement dit :

logarithmique < **polynomiale** < **exponentielle**

- Les algorithmes exponentiels sont très coûteux en temps.
- On privilégie donc les algorithmes polynomiaux ou mieux.
- En pratique : on évite tout ce qui est exponentiel.

COMPLEXITE

Petit lexique

Un algorithme dont la complexité est	est dit
$O(1)$	Constant
$O(\log n)$	Logarithmique
$O(n)$	Linéaire
$O(n^2)$	Quadratique
$O(n^3)$	Cubique

Ordre de domination des fonctions

