
Algorithmique

INF5130

Samuele Giraudo

UQÀM

Automne 2026

Ce cours est basé sur du matériel de Louise Laforest (louise.laforest@uqam.ca).

Toute erreur dans ce cours est de la responsabilité de Samuele Giraudo.

1. Informations générales	4
2. Problèmes et algorithmes	15
3. Complexité temporelle	41
4. Rappels mathématiques	101
5. Équations de récurrence	140
6. Diviser pour régner	167

1. Informations générales

1.1. Informations pratiques 6

1.2. Aperçu 10

1.3. Références 13

1.1. Informations pratiques

- Titre du cours : *Algorithmique*
- Sigle : INF5130
- Département : Informatique
- Coordinateur : Samuele Giraudo
- Enseignant : Samuele Giraudo
- Courriel : giraudo.samuele@uqam.ca
- Bureau : PK-4430
- Page personnelle : <https://www.giraudo.uqam.ca/>
- Site du cours : <https://www.giraudo.uqam.ca/Teaching/INF5130/2026-09/INF5130.html>
- Plan de cours : https://info.uqam.ca/plan_cours/Automne%202026/INF5130.html

Prérequis :

- validation d'INF3105, *Structures de données et algorithmes* ;

ainsi que toutes ses dépendances :

- validation d'INF1120, *Programmation I* ;
- validation d'INF1132, *Mathématiques pour l'informatique* ou de MAT1060, *Mathématiques algorithmiques* ;
- validation d'INF2120, *Programmation II*.

Calendrier des évaluations avec leurs pondérations :

1. séance 9 : examen 1, 50% ;
2. séance 15 : examen 2, 50%.

Contenu général :

- l'examen 1 porte sur ce qui a été vu dans les 5 premières séances ;
- l'examen 2 porte sur ce qui a été vu dans les 10 premières séances.

Pour les examens, aucun document n'est autorisé. Tout appareil électronique est interdit.

Pour valider le cours, il faut avoir une moyenne supérieure à 50 %.

1.2. Aperçu

Objectifs principaux :

- maîtriser la notion de problème en informatique ;
- connaître les algorithmes de base de l'informatique ;
- analyser la complexité et l'efficacité de différents types d'algorithmes ;
- connaître et appliquer les grands principes de la conception des algorithmes.

Le cours s'articule en les axes suivants (présentés dans un ordre différent).

1. Bases mathématiques.

- Comportement asymptotique.
- Manipulation de calcul.

2. Problèmes et algorithmes.

- Problèmes.
- Réductions polynomiales.
- NP-complétude.

3. Complexité.

- Complexité temporelle.
- Opérations barométrique.
- Analyse amortie.

4. Types d'algorithmes.

- Diviser pour régner.
- Gloutons.
- À retour arrière.

5. Algorithmes sur les graphes.

- Parcours.
- Algorithmes de Prim et de Kruskal.
- Algorithme de Ford-Fulkerson.

1.3. Références

Les **principaux livres** sur lesquels s'appuie le cours sont

1. Cormen, T., Leiserson, C., Rivest, R., Stein, C., *Algorithmique*, 3e édition, Dunod, 2010 ;
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D., *Data Structures and Algorithms*, Addison-Wesley, 1983.

Pour des notions mathématiques utiles dans ce cours, des **références classiques** sont

1. Rosen, K.H., *Discrete Mathematics and its Applications*, 8th edition, Mc Graw Hill Education, 2019 ;
2. Graham, R.L., Knuth, D.E., Patashnik, O., *Concrete Mathematics: a Foundation for Computer Science*, Addison-Wesley, 1994.

2. Problèmes et algorithmes

- 2.1. Introduction 17
- 2.2. Problèmes 22
 - 2.2.1. Problème de la fouille 23
 - 2.2.2. Problème de l'accessibilité 25
 - 2.2.3. Problème de la satisfaisabilité 27
 - 2.2.4. Problème du voyageur de commerce 29
 - 2.2.5. Problème de la sélection 32
 - 2.2.6. Problème du tri 34
 - 2.2.7. Problème de l'arrêt 36
- 2.3. Programmes 38

2.1. Introduction

- **Problème** : description d'une tâche à accomplir, modélisée par
 - l'**entrée** : la donnée de départ ;
 - la **taille** : une mesure de la taille de l'entrée ;
 - la **sortie** : la donnée que l'on souhaite calculer.

Exemple

Soit le problème prenant en entrée un tableau t de nombres et produisant en sortie la version triée de t . La taille de t est son nombre d'éléments.

Une **instance** d'un problème P est une entrée de P .

Exemple

Le tableau $t := [4, 1, 1, 5, 3, 0, 9]$ est une instance du problème précédent.

Soit P un problème.

- Un processus pour P est une description finie d'un calcul pouvant être exécuté sur toute entrée de P .
- Un processus résout P si sur toute entrée de P dont l'exécution termine, il produit la sortie attendue.

Exemple

Problème des antécédents d'une fonction $f: X \rightarrow Y$: l'entrée est un $y \in Y$ dont la taille est 1 et la sortie est l'ensemble des $x \in X$ tels que $f(x) = y$.

Le processus qui consiste à parcourir tous les $x \in X$ et tester si $f(x) = y$ pour l'inclure au résultat est un processus qui résout P .

Soit P un problème.

- Un **algorithme pour P** est un processus qui résout P et dont l'exécution est en temps fini sur toute entrée.

Exemple

Problème du **gcd** de deux entiers : l'entrée est un couple (n_1, n_2) d'entiers dont la taille est le nombre de bits nécessaire pour représenter les deux entiers n_1 et n_2 , et la sortie est leur plus grand diviseur commun.

L'**algorithme d'Euclide** est bien un algorithme qui résout ce problème.

Soit P un problème.

Il peut exister **zéro, un ou plusieurs algorithmes** pour P .

- Lorsqu'il n'existe aucun algorithme pour P , P est **non calculable**.
- Soit R une ressource (comme le temps, la mémoire ou l'énergie).

Un algorithme A pour P est **optimal pour R** si pour tout algorithme A' pour P , pour toute entrée x de P , A utilise moins de ressource R que A' pour calculer la sortie de x .

2.2. Problèmes

2.2.1. Problème de la fouille

Le **problème de la fouille** est défini par

- entrée : un élément x et un tableau t ;
- taille : la taille n de t ;
- sortie : « oui » s'il existe un indice $i \in [0, n - 1]$ tel que $t[i] = x$ et « non » sinon.

Exemples
• Sur l'instance $(7, [3, -7, 10, 67, 25, 100])$ de taille 6, la réponse est « non ». • Sur l'instance $(67, [13, -7, 11, 67, 2])$ de taille 5, la réponse est « oui ».

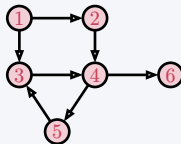
2.2.2. Problème de l'accessibilité

Le **problème de l'accessibilité** est défini par

- entrée : un graphe orienté $G = (S, A)$ et deux sommets $s, t \in S$;
- taille : le couple $(\#S, \#A)$;
- sortie : « oui » si t est accessible depuis s et « non » sinon.

Exemples

Soit G le graphe orienté



Sa taille est $(6, 7)$.

- Sur l'instance $(G, 1, 6)$, la réponse est « oui ».
- Sur l'instance $(G, 4, 1)$, la réponse est « non ».

2.2.3. Problème de la satisfaisabilité

Le **problème de la satisfaisabilité** est défini par

- entrée : une formule logique f sur les connecteurs $\neg$, $\wedge$ et $\vee$ appliqués sur un nombre arbitraire de variables logiques ;
- taille : le nombre total d'occurrences de variables et de connecteurs logiques de f ;
- sortie : « oui » si f est **satisfaisable** et « non » sinon.

Exemples
• Sur l'instance $v_1 \wedge (\neg v_2) \wedge (v_1 \vee (\neg v_3) \vee v_2)$ de taille 11, la réponse est « oui » car l'affectation $v_1 := 1$, $v_2 := 0$ et $v_3 := 0$ rend cette formule vraie. • Sur l'instance $(v_1 \wedge (\neg v_1)) \vee (v_2 \wedge (\neg v_2))$ de taille 9, la réponse est « non » car cette formule est contradictoire.

Il existe beaucoup de **variantes** au problème de la satisfaisabilité, notamment celui sur les formules en **forme normale conjonctive** (voir plus loin).

2.2.4. Problème du voyageur de commerce

Une **permutation** de taille n est une bijection de $[n]$ dans $[n]$.

Une **fonction de coût** de taille n est une fonction $c : [n] \times [n] \rightarrow \mathbb{N}$.

La **c -longueur** d'une permutation π de taille n est la quantité

$$\sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1)).$$

Exemple

Soit la permutation π de taille 3 définie par $\pi(1) := 2$, $\pi(2) := 3$ et $\pi(3) := 1$.

Soit la fonction de coût c de taille 3 définie par $c(i, i) := 0$ pour tout $i \in [3]$, $c(1, 2) := 7$, $c(1, 3) := 2$, $c(2, 1) := 4$, $c(2, 3) := 1$, $c(3, 1) := 2$ et $c(3, 2) := 6$.

La c -longueur de π est

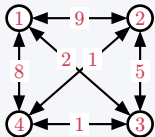
$$c(\pi(1), \pi(2)) + c(\pi(2), \pi(3)) + c(\pi(3), \pi(1)) = c(2, 3) + c(3, 1) + c(1, 2) = 1 + 2 + 7 = 10.$$

Le **problème du voyageur de commerce** est défini par

- entrée : une fonction de coût c ;
- taille : la taille n de c ;
- sortie : une permutation de taille n dont la c -longueur est minimale parmi toutes les permutations de taille n .

Remarque : pour ce problème, plusieurs sorties différentes pour une même entrée sont acceptées.

Exemple



Considérons la fonction de coût c exprimée via le graphe ci-contre.

Par exemple, $c(1,3) = c(3,1) = 2$.

Cette fonction de coût c est une instance au problème du voyageur de commerce.

La permutation π définie par $\pi(1) := 2$, $\pi(2) := 4$, $\pi(3) := 3$ et $\pi(4) := 1$ a pour c -longueur $1 + 1 + 2 + 9 = 13$ et est une sortie sur c .

2.2.5. Problème de la sélection

Le **problème de la sélection** est défini par

- entrée : un tableau t de n éléments distincts et un entier $i \in [n]$;
- taille : n ;
- sortie : l'élément x de t tel que
 - exactement $i - 1$ éléments de t sont strictement inférieurs à x ;
 - exactement $n - i$ éléments de t sont strictement supérieurs à x .

Exemple

Soit le tableau $t := [23, 12, 9, 5, 17, 45, 3, 29, 21, 37]$ et $i := 6$.

La réponse au problème de la sélection sur l'instance (t, i) est $x := 21$, car

- les 5 éléments 3, 5, 9, 12, 17 de t sont strictement inférieurs à 21 ;
- les 4 éléments 23, 29, 37, 45 de t sont strictement supérieurs à 21.

2.2.6. Problème du tri

Le **problème du tri** est défini par

- entrée : un tableau t de n éléments ;
- taille : la taille n de t ;
- sortie : un tableau t' obtenu par permutation des éléments de t et ordonné dans l'ordre croissant.

Exemple

Voici une instance de taille 9 du problème du tri et sa sortie :

i	0	1	2	3	4	5	6	7	8
t	18	3	7	12	23	11	17	15	3
t'	3	3	7	11	12	15	17	18	23

2.2.7. Problème de l'arrêt

Soit $\mathcal{L}$ un langage de programmation Turing-complet.

Le **problème de l'arrêt** est défini par

- entrée : un programme F écrit en $\mathcal{L}$ et une entrée x pour F ;
- taille : le nombre de symboles nécessaires pour représenter (F, x) ;
- sortie : « oui » si l'exécution de F sur x s'arrête et « non » sinon.

Résultat fondamental
Le problème de l'arrêt est non calculable : aucun algorithme ne le résout pour toutes ses instances.

Remarque : il existe des langages de programmation non Turing-complets (donc, strictement moins puissants que des langages qui le sont) mais qui l'avantage d'avoir le problème de l'arrêt calculable.

2.3. Programmes

Dans le contexte de ce cours, un **programme** est l'implantation d'un algorithme dans un langage de programmation.

Nous décrirons les algorithmes en **pseudo-code**, pouvant être compris comme du Python **simplifié**.

Ainsi, nous utiliserons

- les variables et affectations ;
- les listes pour coder les tableaux ;
- la création de tableaux ;
- l'accès aux cases des tableaux en lecture et en écriture ;
- les structures `if`, `if-else`, `while` et la récursion.

Nous éviterons en particulier la structure `for` et la fonction `range`.

Voici quelques conventions générales :

- toute variable apparaissant dans un pseudo-code est en police à chasse fixe (par exemple « `x` »). Son analogue mathématique est écrit en italique (par exemple « x ») ;
- la fonction `len` est paramétrée par un tableau `t` et renvoie la taille de `t`, c'est-à-dire son nombre de cases ;
- l'instruction `t = [x] * n`, où `x` est une valeur et n est un entier positif affecte à `t` un tableau de taille n dont toutes les cases sont initialisées à x ;
- si i est un entier compris entre 0 et `len(t) - 1`, alors `t[i]` désigne la case d'indice i de `t`. Les cases sont indicées à partir de 0 ;
- si i et j sont des entiers tels que $0 \leq i \leq j \leq \text{len}(t)$, alors `t[i : j]` est la **tranche** de `t` entre i et j . C'est le tableau `[t[i], t[i + 1], ..., t[j - 1]]` ;
- les **tranches** ne seront jamais utilisées dans les pseudo-codes. En revanche, elles seront utilisées uniquement pour les explications et manipulations mathématiques.

3. Complexité temporelle

- 3.1. Introduction 43
- 3.2. Modèle de coût 45
- 3.3. Manipulation des sommes 53
- 3.4. Exemples d'analyse 59
 - 3.4.1. Tri par sélection 60
 - 3.4.2. Tri par insertion 69
 - 3.4.3. Tri fusion 76
- 3.5. Opérations barométriques 87
 - 3.5.1. Principes généraux 88
 - 3.5.2. Tri par sélection 97
 - 3.5.3. Tri par insertion 99

3.1. Introduction

Pour une ressource donnée, la **complexité** d'un algorithme décrit la quantité de cette ressource consommée en fonction de la taille n de ses instances.

Il est en particulier intéressant de comprendre comment cela évolue lorsque n devient grand.

Deux types principaux de complexité :

- complexité **spatiale** : quantité de mémoire utilisée.
- complexité **temporelle** : temps d'exécution.

Le temps d'exécution réel d'un algorithme dépend notamment

- du langage de programmation choisi pour implanter l'algorithme ;
- de l'implantation de l'algorithme ;
- du compilateur pour obtenir un exécutable ;
- de la plateforme matérielle qui exécute l'exécutable.

3.2. Modèle de coût

L'analyse de la complexité temporelle **abstrait** les facteurs propres à une machine ou à une implantation particulière.

On fixe pour cela un **modèle de coût** attribuant un coût aux opérations élémentaires, puis on exprime le coût total en fonction de la taille des instances.

La complexité temporelle peut être étudiée

- dans le **pire cas** ;
- dans le **cas moyen** ;
- dans le **meilleur cas**.

La plupart des langages de programmation impératifs proposent des opérations ou instructions que l'on peut considérer comme **élémentaires** (également qualifiées d'**atomiques**) dans un modèle abstrait :

1. opérations élémentaires :

- arithmétiques : $+$, $-$, $\times$, $/$, *etc.* ;
- logiques : $\vee$, $\wedge$, $\neg$, *etc.* ;
- comparaisons : $<$, $\leq$, $>$, $\geq$, $=$, *etc.* ;

2. instructions élémentaires :

- affectation ;
- appel d'une fonction ;
- retour d'une valeur par une fonction ;

3. instructions structurées :

- `if` et `if-else` ;
- `while` .

Sauf mention contraire, une **opération** ou une **instruction élémentaire** coûte **une unité de temps**, à laquelle s'ajoute le coût d'évaluation de ses opérandes.

Exemple

L'instruction `f(a, b) + g(c)` a pour coût la somme

- du coût de l'opération élémentaire `+`, qui par convention est `1` ;
- du coût de l'évaluation de l'opérande `f(a, b)` ;
- du coût de l'évaluation de l'opérande `g(c)` .

Une exception à cela est l'instruction d'initialisation de tableau `t = [x] * n` qui coûte $1 + n \times c_x$ unités de temps, où c_x est le coût en unités de temps nécessaire à l'évaluation de `x` (en particulier, lorsque `x` est une constante, $c_x = 1$).

Note : ceci est une convention propre à ce cours qui ne dépend pas de la manière dont Python gère cela.

Exemple

L'instruction `t = [2 + 3 + 5] * 12` a pour coût $1 + 5 \times 12 = 61$.

Pour tout fragment de programme I , notons

$$\tau(I)$$

le **coût temporel** nécessaire à l'évaluation ou l'exécution de I .

Exemples

Dans le modèle de coût choisi :

- $\tau(x + 4) = 3$ (évaluation de x et de 4 , et calcul du $+$) ;
- $\tau(y = 2 * x) = 4$ (évaluation de 2 et de x , calcul du $*$ et affectation) ;
- $\tau((x + 4) * y) = 5$ (évaluation de x , 4 et y , et calcul du $+$ et du $*$) ;
- $\tau(x = 0; y = \text{input}()) = 2 + \tau(y = \text{input}()) = 3 + \tau(\text{input}())$.

Une instruction conditionnelle est de la forme

```
if C:  
    I_1  
else:  
    I_2
```

Si les coûts des deux branches sont fixes (elles ne dépendent par exemple pas du résultat d'une entrée au cours de leur exécution), alors le coût $\tau(\text{if } C: I_1 \text{ else: } I_2)$ s'exprime de la façon suivante.

- Pire cas : $\tau(C) + \max(\tau(I_1), \tau(I_2))$;
- meilleur cas : $\tau(C) + \min(\tau(I_1), \tau(I_2))$;
- cas moyen : si la première branche est prise avec probabilité p , $\tau(C) + p\tau(I_1) + (1 - p)\tau(I_2)$.

Attention : dans le cas moyen, on ne peut prendre $p = \frac{1}{2}$ que si les deux branches sont équiprobables pour la distribution d'entrées considérée.

Une boucle conditionnelle est de la forme

```
while C:  
  I
```

Supposons que le coût de la condition `C` et celui du corps `I` soient constants d'une itération à l'autre. Si `I` est exécuté n fois, alors `C` est évaluée $n + 1$ fois et

$$\tau = (n + 1)\tau(C) + n\tau(I).$$

En notant respectivement $n_{\max}$, $n_{\min}$ et n_{moy} le nombre maximal, minimal et moyen d'itérations, nous obtenons les coûts correspondants à ces cas en remplaçant n par ces valeurs.

Remarque : si le coût du corps `I` n'est pas constant, il faut utiliser une somme sur le coût de chaque itération de `I`. Un exemple de cette situation suivra.

Puisque nous n'utiliserons pas de structure `for`, une boucle à compteur s'écrit via un `while` :

```
i = v_0
while i <= v_1:
    I
    i = i + 1
```

Le nombre d'itérations est

$$n = v_1 - v_0 + 1.$$

Le coût total τ est

$$\begin{aligned}\tau &= \tau(i = v_0) + (n + 1)\tau(i \leq v_1) + n\tau(I) + n\tau(i = i + 1) \\ &= 1 + \tau(v_0) + (n + 1)(1 + \tau(v_1)) + n\tau(I) + 2n \\ &= 3n + 2 + \tau(v_0) + (n + 1)\tau(v_1) + n\tau(I).\end{aligned}$$

3.3. Manipulation des sommes

Pour dénombrer combien de fois est exécutée une instruction I au sein d'une boucle, la sommation (dont le symbole est « $\sum$ ») est l'outil principal.

Exemple

Considérons les instructions

```
i = 0
while i < n:
    j = 0
    while j < i:
        I
        j = j + 1
    i = i + 1
```

pour lesquelles nous souhaitons compter le nombre de fois que I est exécutée.

Cette quantité est

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1.$$

En effet, la première sommation $\sum_{i=0}^{n-1}$ résulte de première boucle et la deuxième somme $\sum_{j=0}^{i-1}$ résulte de la seconde.

Notons bien que la borne $i-1$ de la seconde sommation dépend de la variable d'itération i de la première.

Il reste maintenant à décrire des méthodes pour **simplifier** ce type d'expressions.

Il est important de **simplifier** les expressions faisant intervenir des **sommations** : lorsque cela est possible, il faut les **éliminer**

Voici plusieurs **identités sur les sommations** pour cet objectif.

- **Addition multiple.**

$$\sum_{i=\alpha}^{\beta} \lambda = (\beta - \alpha + 1)\lambda,$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$, et λ ne dépend pas de i .

- **Éclatement.**

$$\sum_{i=\alpha}^{\beta} (f_1(i) + f_2(i)) = \sum_{i=\alpha}^{\beta} f_1(i) + \sum_{i=\alpha}^{\beta} f_2(i),$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$ et $f_1, f_2 : \mathbb{Z} \rightarrow \mathbb{Z}$.

- Distributivité.

$$\sum_{i=\alpha}^{\beta} \lambda f(i) = \lambda \sum_{i=\alpha}^{\beta} f(i),$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$, λ ne dépend pas de i et $f: \mathbb{Z} \rightarrow \mathbb{Z}$.

- Décalage des indices.

$$\sum_{i=\alpha}^{\beta} f(i) = \sum_{i=\alpha+d}^{\beta+d} f(i-d),$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$, $f: \mathbb{Z} \rightarrow \mathbb{Z}$ et $d \in \mathbb{Z}$.

- Somme des premiers entiers.

$$\sum_{i=0}^{\beta} i = \frac{\beta(\beta+1)}{2}, \quad \beta \in \mathbb{N},$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$.

- Somme des carrés des premiers entiers.

$$\sum_{i=0}^{\beta} i^2 = \frac{\beta(\beta+1)(2\beta+1)}{6}, \quad \beta \in \mathbb{N},$$

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$.

Exemple

L'expression de l'exemple précédent est

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1.$$

Elle se simplifie, en éliminant ses sommations, ainsi :

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} (i - 1 - 0 + 1) \times 1 \quad (\text{Addition multiple.})$$

$$= \sum_{i=0}^{n-1} i \quad (\text{Simplification directe.})$$

$$= \frac{(n-1)n}{2} \quad (\text{Somme des premiers entiers.})$$

3.4. Exemples d'analyse

3.4.1. Tri par sélection

Le **tri par sélection** d'un tableau t construit progressivement un suffixe définitivement trié. À l'étape d'indice i , on cherche un plus grand élément dans $t[0 : i]$, puis on l'échange avec $t[i]$.

Invariant : juste après l'itération d'indice i , le suffixe $t[i : n]$ contient les plus grands éléments du tableau, à leur position définitive (donc, dans l'ordre croissant).

Exemple									
À chaque itération, le plus grand élément de la partie non triée est placé à droite. La zone colorée est définitivement triée : elle n'est plus examinée aux itérations suivantes.		0	1	2	3	4	5	6	7
	t	5	18	3	2	15	7	11	10
	Itération $i = 7$	5	10	3	2	15	7	11	18
	Itération $i = 6$	5	10	3	2	11	7	15	18
	Itération $i = 5$	5	10	3	2	7	11	15	18
	Itération $i = 4$	5	7	3	2	10	11	15	18
	Itération $i = 3$	5	2	3	7	10	11	15	18
	Itération $i = 2$	3	2	5	7	10	11	15	18
	Itération $i = 1$	2	3	5	7	10	11	15	18

Nous utilisons pour commencer l'algorithme suivant. Le paramètre `i` désigne la taille du préfixe du tableau `t` considéré.

Algorithme `indice_du_plus_grand`

```
def indice_du_plus_grand(i, t):  
    j_max = 0           # c_1  
    j = 1               # c_2  
    while j < i:        # c_3  
        if t[j] > t[j_max]: # c_4  
            j_max = j    # c_5  
        j = j + 1       # c_6  
    return j_max        # c_7
```

Toute exécution de cet algorithme renvoie l'indice d'un plus grand élément du tableau `t` entre les indices `0` et `i - 1`.

Convention : chaque `# c_k` désigne le coût c_k de l'instruction sur laquelle ce commentaire est placé.

Idée : pour calculer le coût d'une exécution de l'algorithme, au lieu de tenir compte du nombre exact d'unités de temps de chaque instruction, nous allons utiliser les c_k .

Soit ℓ le nombre d'exécutions de l'instruction `j_max = j`. Bien entendu, $0 \leq \ell \leq i - 1$.

Les différentes lignes sont exécutées :

- 1 fois pour les lignes de coûts c_1 , c_2 et c_7 ;
- i fois pour le test de coût c_3 ;
- $i - 1$ fois pour les lignes de coûts c_4 et c_6 ;
- ℓ fois pour la ligne de coût c_5 .

Ainsi,

$$\tau(\text{indice_du_plus_grand}(i, t)) = c_1 + c_2 + ic_3 + (i - 1)c_4 + \ell c_5 + (i - 1)c_6 + c_7.$$

- Dans le **meilleur cas**, `j_max` n'est jamais mis à jour, donc $\ell = 0$ et

$$\tau(\text{indice_du_plus_grand}(i, t)) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_6) + c_7.$$

Cela arrive par exemple lorsque la première case de t contient sa plus grande valeur.

- Dans le **pire cas**, `j_max` est mis à jour à chaque itération, donc $\ell = i-1$ et

$$\tau(\text{indice_du_plus_grand}(i, t)) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_5 + c_6) + c_7.$$

Cela arrive par exemple lorsque les valeurs de t sont strictement croissantes de gauche à droite.

Dans les deux cas, le temps d'exécution est un polynôme de degré 1 (aussi appelé « fonction affine ») en i .

Considérons l'algorithme suivant du tri par sélection.

Algorithme `tri_selection`

```
def tri_selection(t):  
    n = len(t)                # c_8  
    i = n - 1                 # c_9  
    while i >= 1:             # c_10  
        i_max = indice_du_plus_grand(i + 1, t) # c_11  
        t[i], t[i_max] = t[i_max], t[i]        # c_12  
        i = i - 1             # c_13
```

Toute exécution de cet algorithme trie le tableau `t`.

Ici, le coût c_{11} désigne le coût de l'instruction correspondant à `# c_11`, sans y inclure le coût de l'appel `indice_du_plus_grand(i + 1, t)`. Il sera traité isolément.

La boucle principale est exécutée $n - 1$ fois.

Soit `t` le tableau en entrée et n sa taille.

Le temps d'exécution du tri est

$$\tau(\text{tri_selection}(t)) = c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) + \sum_{i=1}^{n-1} \tau(\text{indice_du_plus_grand}(i+1, t)).$$

Évaluons et simplifions cette expression autant que possible, à la fois pour le pire cas et le meilleur cas.

Observons que le meilleur et le pire cas dépendent uniquement de l'exécution du sous-algorithme

`indice_du_plus_grand` :

- le meilleur cas est atteint lorsque `indice_du_plus_grand` est dans le meilleur cas ;
- le pire cas est atteint lorsque `indice_du_plus_grand` est dans le pire cas.

Dans le meilleur cas, $\tau(\text{indice_du_plus_grand}(i, t)) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_6) + c_7$ et ainsi,

$$\begin{aligned}\tau(\text{tri_selection}(t)) &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) \\ &\quad + \sum_{i=1}^{n-1} (c_1 + c_2 + (i+1)c_3 + ic_4 + ic_6 + c_7) \\ &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) \\ &\quad + (n-1)(c_1 + c_2 + c_7) + c_3 \sum_{i=1}^{n-1} (i+1) + (c_4 + c_6) \sum_{i=1}^{n-1} i \\ &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13} + c_1 + c_2 + c_7) \\ &\quad + \frac{(n^2 + n - 2)c_3}{2} + \frac{(n^2 - n)(c_4 + c_6)}{2}.\end{aligned}$$

Ce meilleur cas apparaît pour des tableaux de la forme $[n, n-1, \dots, 1]$.

Dans le pire cas, $\tau(\text{indice_du_plus_grand}(i, t)) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_5 + c_6) + c_7$ et ainsi,

$$\begin{aligned}\tau(\text{tri_selection}(t)) &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) \\ &\quad + \sum_{i=1}^{n-1} (c_1 + c_2 + (i+1)c_3 + i(c_4 + c_5 + c_6) + c_7) \\ &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) \\ &\quad + (n-1)(c_1 + c_2 + c_7) + c_3 \sum_{i=1}^{n-1} (i+1) + (c_4 + c_5 + c_6) \sum_{i=1}^{n-1} i \\ &= c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13} + c_1 + c_2 + c_7) \\ &\quad + \frac{(n^2 + n - 2)c_3}{2} + \frac{(n^2 - n)(c_4 + c_5 + c_6)}{2}.\end{aligned}$$

Ce pire cas apparaît pour des tableaux de la forme $[1, 2, \dots, n]$.

3.4.2. Tri par insertion

Le **tri par insertion** d'un tableau t maintient un préfixe trié. À l'itération i , l'élément $t[i]$ est inséré à sa position dans le préfixe $t[0 : i]$.

Invariant : juste après l'itération d'indice i , le préfixe $t[0 : i + 1]$ est une version triée du préfixe de longueur $i + 1$ du tableau t initial.

Exemple

À chaque itération, l'élément en position i est inséré à la bonne position dans la partie colorée.

La zone colorée est **relativement triée** : des éléments hors de la zone pourront s'y insérer plus tard dans l'exécution.

	0	1	2	3	4	5	6	7
t	5	18	3	2	15	7	11	10
Itération $i = 1$	5	18	3	2	15	7	11	10
Itération $i = 2$	3	5	18	2	15	7	11	10
Itération $i = 3$	2	3	5	18	15	7	11	10
Itération $i = 4$	2	3	5	15	18	7	11	10
Itération $i = 5$	2	3	5	7	15	18	11	10
Itération $i = 6$	2	3	5	7	11	15	18	10
Itération $i = 7$	2	3	5	7	10	11	15	18

Considérons l'algorithme suivant.

Algorithme tri_insertion

```
def tri_insertion(t):  
    n = len(t)                # c_1  
    i = 1                     # c_2  
    while i < n:               # c_3  
        x = t[i]               # c_4  
        j = i                  # c_5  
        while j >= 1 and t[j - 1] > x: # c_6  
            t[j] = t[j - 1]    # c_7  
            j = j - 1          # c_8  
        t[j] = x               # c_9  
        i = i + 1              # c_10
```

Toute exécution de cet algorithme trie le tableau `t`.

La boucle principale est exécutée $n - 1$ fois et son test est effectué n fois.

Soit `t` le tableau en entrée et n sa taille.

Pour chaque $i \in [n-1]$, notons ℓ_i le nombre d'itérations de la boucle intérieure lors de l'itération i de la boucle principale. Cette valeur vérifie $0 \leq \ell_i \leq i$.

Le test de la boucle intérieure est effectué $\ell_i + 1$ fois, tandis que les instructions de coûts c_7 et c_8 sont effectuées ℓ_i fois.

Ainsi,

$$\tau(\text{tri_insertion}(t)) = c_1 + c_2 + nc_3 + (n-1)(c_4 + c_5 + c_9 + c_{10}) + \sum_{i=1}^{n-1} ((\ell_i + 1)c_6 + \ell_i(c_7 + c_8)).$$

Dans le meilleur cas, la condition de la boucle intérieure est fausse dès sa première évaluation. Ainsi, $\ell_i = 0$ pour tout $i \in [n-1]$.

Nous obtenons alors

$$\begin{aligned}\tau(\text{tri_insertion}(t)) &= c_1 + c_2 + nc_3 + (n-1)(c_4 + c_5 + c_9 + c_{10}) + \sum_{i=1}^{n-1} c_6 \\ &= c_1 + c_2 + nc_3 + (n-1)(c_4 + c_5 + c_6 + c_9 + c_{10}).\end{aligned}$$

Le temps d'exécution dans le meilleur cas est donc une fonction affine de n .

Ce meilleur cas apparaît notamment pour les tableaux déjà triés.

Dans le **pire cas**, chaque nouvel élément doit traverser tout le préfixe déjà trié. Ainsi, $\ell_i = i$ pour tout $i \in [n-1]$.

Nous obtenons ainsi

$$\begin{aligned}\tau(\text{tri_insertion}(t)) &= c_1 + c_2 + nc_3 + (n-1)(c_4 + c_5 + c_9 + c_{10}) + \sum_{i=1}^{n-1}((i+1)c_6 + i(c_7 + c_8)) \\ &= c_1 + c_2 + nc_3 + (n-1)(c_4 + c_5 + c_9 + c_{10}) \\ &\quad + \frac{(n^2 + n - 2)c_6}{2} + \frac{(n^2 - n)(c_7 + c_8)}{2}.\end{aligned}$$

Le temps d'exécution dans le pire cas est donc une fonction quadratique de n .

Ce pire cas apparaît notamment pour les tableaux triés dans l'ordre décroissant.

Voici une comparaison des analyses des deux tris considérés :

Cas	Tri par sélection	Tri par insertion
Pire cas	quadratique	quadratique
Meilleur cas	quadratique	affine

Observations :

- le tri par insertion bénéficie fortement d'une entrée déjà triée ;
- à l'inverse, le tri par sélection est peu dépendant des caractéristiques du tableau en entrée.

3.4.3. Tri fusion

Le **tri fusion** repose sur le paradigme « **diviser pour régner** » :

1. diviser le tableau en deux sous-tableaux de tailles similaires ;
2. **trier récursivement** chacun des deux sous-tableaux ;
3. fusionner les deux sous-tableaux triés.

Exemple
Soit le tableau à trier $t := [5, 18, 3, 2, 15, 7, 11, 10]$. 1. Division en deux sous-tableaux : $[5, 18, 3, 2]$ et $[15, 7, 11, 10]$. 2. Tri récursif des deux sous-tableaux : $[2, 3, 5, 18]$ et $[7, 10, 11, 15]$. 3. Fusion : $[2, 3, 5, 7, 10, 11, 15, 18]$.

Considérons l'algorithme suivant.

Algorithme tri_fusion

```
def tri_fusion(t, debut, fin):  
    if debut < fin:                                # c_1  
        milieu = (debut + fin) // 2                # c_2  
        tri_fusion(t, debut, milieu)               # c_3  
        tri_fusion(t, milieu + 1, fin)              # c_4  
        u = fusionner(t, debut, milieu, fin)        # c_5  
        k = 0                                       # c_6  
        while k < len(u):                          # c_7  
            t[debut + k] = u[k]                    # c_8  
            k = k + 1                               # c_9
```

L'appel `tri_fusion(t, debut, fin)` trie le sous-tableau `t[debut : fin + 1]`.

Si ce sous-tableau contient n éléments, les deux appels récursifs portent sur des sous-tableaux dont les tailles sont voisines de $\frac{n}{2}$.

Ici, les coûts c_3 , c_4 et c_5 n'incluent pas les coûts des appels de fonctions.

Algorithme fusionner

```

def fusionner(t, debut, milieu, fin):
    u = [0] * (fin - debut + 1)    # c_10
    i = debut                       # c_11
    j = milieu + 1                  # c_12
    k = 0                           # c_13
    while i <= milieu and j <= fin: # c_14
        if t[i] <= t[j]:            # c_15
            u[k] = t[i]             # c_16
            i = i + 1               # c_17
        else:
            u[k] = t[j]             # c_18
            j = j + 1               # c_19
            k = k + 1               # c_20

```

```

    while i <= milieu:              # c_21
        u[k] = t[i]                 # c_22
        i = i + 1                   # c_23
        k = k + 1                   # c_24
    while j <= fin:                  # c_25
        u[k] = t[j]                 # c_26
        j = j + 1                   # c_27
        k = k + 1                   # c_28
    return u                         # c_29

```

La précondition à la fusion est que `t[debut : milieu + 1]` et `t[milieu + 1 : fin + 1]` soient triés.

Analysons l'algorithme `fusionner`.

Soient $n_1 := \text{milieu} - \text{debut} + 1$ et $n_2 := \text{fin} - \text{milieu}$.

Les deux sous-tableaux `t[debut : milieu + 1]` et `t[milieu + 1 : fin + 1]` contiennent respectivement n_1 et n_2 éléments, avec $n = n_1 + n_2$.

Notons p_1 (resp. p_2) le nombre d'itérations de la première boucle lors desquelles un élément du premier (resp. second) sous-tableau est placé dans le nouveau tableau `u`.

La première boucle est donc exécutée $p_1 + p_2$ fois.

Lorsqu'elle se termine, l'un des deux sous-tableaux est entièrement parcouru. Ainsi, $p_1 = n_1$ ou $p_2 = n_2$.

Les différentes lignes sont exécutées comme suit :

- la ligne de coût c_{14} est exécutée $p_1 + p_2 + 1$ fois ;
- les lignes de coûts c_{15} et c_{20} sont exécutées $p_1 + p_2$ fois ;
- les lignes de coûts c_{16} et c_{17} sont exécutées p_1 fois ;
- les lignes de coûts c_{18} et c_{19} sont exécutées p_2 fois ;
- la ligne de coût c_{21} est exécutée $n_1 - p_1 + 1$ fois ;
- les lignes de coûts c_{22} , c_{23} et c_{24} sont exécutées $n_1 - p_1$ fois ;
- la ligne de coût c_{25} est exécutée $n_2 - p_2 + 1$ fois ;
- les lignes de coûts c_{26} , c_{27} et c_{28} sont exécutées $n_2 - p_2$ fois.

Les lignes de coûts c_{10} , c_{11} , c_{12} , c_{13} et c_{29} sont exécutées une seule fois.

On obtient donc

$$\begin{aligned}\tau(\text{fusionner}(t, \text{debut}, \text{milieu}, \text{fin})) &= c_{10} + c_{11} + c_{12} + c_{13} + (p_1 + p_2 + 1)c_{14} + (p_1 + p_2)c_{15} \\ &\quad + p_1(c_{16} + c_{17}) + p_2(c_{18} + c_{19}) + (p_1 + p_2)c_{20} \\ &\quad + (n_1 - p_1 + 1)c_{21} + (n_1 - p_1)(c_{22} + c_{23} + c_{24}) \\ &\quad + (n_2 - p_2 + 1)c_{25} + (n_2 - p_2)(c_{26} + c_{27} + c_{28}) + c_{29}.\end{aligned}$$

Comme la première boucle se termine dès que l'un des deux sous-tableaux est épuisé,

$$\min(n_1, n_2) \leq p_1 + p_2 \leq n - 1.$$

Par ailleurs,

$$p_1 + p_2 + (n_1 - p_1) + (n_2 - p_2) = n.$$

Ainsi, exactement n éléments sont placés dans `u` au total, en comptant les trois boucles.

Tous les nombres d'exécutions apparaissant dans l'expression précédente sont donc bornés par une fonction affine de n .

Le temps d'exécution de `fusionner` est par conséquent linéaire en n .

Analysons à présent `tri_fusion`.

Pour simplifier, supposons que n est une puissance de 2.

Pour $n \geq 2$, les deux appels récursifs portent alors chacun sur $\frac{n}{2}$ éléments.

Par ailleurs :

- les lignes de coûts c_1 à c_6 sont exécutées une fois ;
- le test de la boucle de coût c_7 est exécuté $n + 1$ fois ;
- les lignes de coûts c_8 et c_9 sont exécutées n fois ;
- l'appel à `fusionner` porte sur un tableau de taille n .

Notons

$$\tau(n) := \tau(\text{tri_fusion}(t, \text{debut}, \text{fin})).$$

Lorsque $\text{fin} - \text{debut} + 1 = n$, nous avons

$$\begin{aligned}\tau(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \\ &\quad + 2\tau\left(\frac{n}{2}\right) + \tau(\text{fusionner}(t, \text{debut}, \text{milieu}, \text{fin})) \\ &\quad + (n+1)c_7 + n(c_8 + c_9).\end{aligned}$$

Comme le temps d'exécution de `fusionner` est linéaire en n , il existe des constantes a et b telles que

$$\tau(n) = 2\tau\left(\frac{n}{2}\right) + an + b.$$

Lorsque $n \leq 1$, seul le test de coût c_1 est exécuté. Ainsi,

$$\tau(0) = \tau(1) = c_1.$$

Nous obtenons donc l'équation de récurrence

$$\tau(n) = \begin{cases} c_1 & \text{si } n \leq 1 \\ 2\tau\left(\frac{n}{2}\right) + an + b & \text{si } n \geq 2. \end{cases}$$

Le temps d'exécution d'un algorithme récursif s'exprime naturellement par une **équation de récurrence**.

Ici, les deux appels récursifs produisent le terme $2\tau\left(\frac{n}{2}\right)$, tandis que la fusion et la copie produisent un coût linéaire en n .

3.5. Opérations barométriques

3.5.1. Principes généraux

Lorsqu'on analyse un algorithme, il n'est pas toujours nécessaire de compter toutes ses opérations.

Une **opération barométrique** est une **opération élémentaire** choisie de sorte que le nombre de ses exécutions soit **représentatif** du travail effectué par l'algorithme.

On estime alors le temps d'exécution en comptant seulement les exécutions de cette opération.

Choix : une bonne opération barométrique est généralement une opération exécutée dans la partie la plus répétée de l'algorithme et dont le nombre d'exécutions varie avec la taille de l'instance.

Il peut par exemple s'agir d'une **comparaison**, d'une **opération arithmétique** ou d'une **affectation**.

Exemple

Considérons la recherche séquentielle suivante.

```
def contient(x, t):  
    i = 0                # c_1  
    while i < len(t):    # c_2  
        if t[i] == x:    # c_3  
            return True  # c_4  
        i = i + 1        # c_5  
    return False         # c_6
```

La comparaison `t[i] == x` est un bon choix d'opération barométrique.

Dans le pire cas, elle est effectuée exactement n fois, où n est la taille du tableau t .

Les autres instructions de la boucle sont exécutées un **nombre similaire de fois**, tandis que les instructions situées hors de la boucle ne sont exécutées qu'un nombre inférieur (ici, constant) de fois.

Contre-exemple

Pour le même algorithme, l'affectation `i = 0` n'est pas un bon choix d'opération barométrique.

Elle est exécutée exactement une fois, quelle que soit la taille du tableau.

Son comptage ne permet donc pas de distinguer une recherche sur un tableau de taille n d'une recherche sur un tableau de taille n' lorsque $n \neq n'$.

Principe : une instruction exécutée seulement un nombre fixe de fois est rarement un bon choix lorsque le temps d'exécution augmente avec la taille de l'instance.

Exemple

Considérons les instructions suivantes.

```
i = 0 # c_1
while i < n: # c_2
    j = 0 # c_3
    while j < i: # c_4
        x = x + 1 # c_5
        j = j + 1 # c_6
    i = i + 1 # c_7
```

L'instruction `x = x + 1` est un bon choix d'opération barométrique.

Elle est exécutée

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

fois et se trouve dans la partie la plus répétée de l'algorithme.

Exemple

Dans la suite d'instructions précédente, le test `j < i` est réalisé à chaque tour de la boucle extérieure une fois de plus que l'instruction `x = x + 1` précédemment choisie.

Ce test est ainsi exécuté

$$\sum_{i=0}^{n-1} \sum_{j=0}^i 1 = \sum_{i=0}^{n-1} (i+1) = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

fois.

Cette quantité est strictement supérieure à celle précédente, mais elle est **du même ordre de grandeur**.

Ainsi, le test `j < i` est à plus forte raison une opération barométrique de la suite d'instruction précédente, et l'affectation `x = x + 1` en est bien toujours une.

Principe : une opération barométrique est une opération qui est exécutée plus souvent que toutes les autres opérations, à l'ordre de grandeur près.

Contre-exemple

Dans l'algorithme précédent, l'affectation `i = i + 1` est exécutée seulement n fois.

Elle n'est pas du même ordre de grandeur que les deux opérations précédemment considérées. Il ne s'agit donc pas d'une opération barométrique.

Principe : dans des boucles imbriquées, une opération située hors de la boucle la plus profonde n'est généralement pas une opération barométrique.

Exemple

Considérons l'algorithme suivant.

```
i = 0           # c_1
while i < n:     # c_2
    x = x + 1    # c_3
    i = i + 1    # c_4

j = 0           # c_5
while j < n * n: # c_6
    y = y + 1    # c_7
    j = j + 1    # c_8
```

L'affectation `x = x + 1` est exécutée n fois.

Cette opération n'est pas une opération barométrique car la seconde boucle effectue n^2 itérations.

En revanche, l'affectation `y = y + 1` est exécutée n^2 fois et reflète la partie qui détermine principalement le temps d'exécution.

Principe : une opération n'est pas barométrique simplement parce qu'elle appartient à une boucle.

Lorsque plusieurs parties de l'algorithme sont exécutées successivement, il faut choisir une opération qui reflète la partie effectuant le plus de travail.

Exemple

Considérons l'algorithme suivant.

```
if condition(t):      # c_1
    i = 0              # c_2
    while i < n:        # c_3
        x = x + 1      # c_4
        i = i + 1      # c_5
else:
    i = 0              # c_6
    while i < n * n:    # c_7
        x = x + 1      # c_8
        i = i + 1      # c_9
```

L'instruction de coût `c_4` ne peut pas servir de baromètre pour toutes les entrées : elle n'est jamais exécutée lorsque la branche `else` est choisie.

De même, l'instruction de coût `c_8` n'est jamais exécutée dans la première branche.

Ici, nous avons besoin de ces deux opérations barométriques prises ensemble afin de couvrir tous les cas d'exécution.

Principe : une opération barométrique doit être choisie en fonction du cas que l'on souhaite analyser.

Il faut vérifier que son nombre d'exécutions reflète bien la partie du calcul qui domine le travail effectué dans ce cas.

3.5.2. Tri par sélection

Pour `tri_selection`, choisissons comme instruction barométrique la comparaison `t[j] > t[j_max]` effectuée par `indice_du_plus_grand`.

Lors d'un appel sur un préfixe de taille i , cette comparaison est effectuée exactement $i - 1$ fois, indépendamment des valeurs contenues dans le tableau.

Les appels successifs de `tri_selection` sur un tableau de taille n portent sur des préfixes de tailles $n, n - 1, \dots, 2$. Le nombre total d'exécutions de l'instruction barométrique est donc

$$\begin{aligned}(n - 1) + (n - 2) + \dots + 1 &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n - 1)}{2}.\end{aligned}$$

Il est donc quadratique en n dans le meilleur et pire cas, propriété conforme à celle de l'analyse précédente.

3.5.3. Tri par insertion

Pour `tri_insertion`, choisissons comme instruction barométrique la comparaison `t[j - 1] > x`.

Son nombre d'exécutions dépend de l'ordre des éléments du tableau en entrée de taille n .

Dans le **meilleur cas**, la comparaison échoue dès sa première évaluation à chaque itération. Ainsi, le nombre d'exécutions de l'opération barométrique est

$$\sum_{i=1}^{n-1} 1 = n - 1.$$

Ce nombre est donc affine en n , propriété conforme à celle de l'analyse précédente.

Dans le **pire cas**, l'élément `x` doit traverser tout le préfixe déjà trié. À l'itération i , la comparaison est alors effectuée exactement i fois. Ainsi, le nombre d'exécutions de l'opération barométrique est

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Ce nombre est donc quadratique en n , propriété conforme à celle de l'analyse précédente.

4. Rappels mathématiques

- 4.1. Notations de Landau 103
 - 4.1.1. Notation O 105
 - 4.1.2. Notation Ω 108
 - 4.1.3. Notation Θ 111
 - 4.1.4. Notations o et ω 113
 - 4.1.5. Critères par limites 115
 - 4.1.6. Propriétés 117
- 4.2. Polynômes 119
- 4.3. Logarithmes et polynômes logarithmiques 125
- 4.4. Polynômes exponentiels 129
- 4.5. Comparaison des familles usuelles 132
- 4.6. Plancher et plafond 135
- 4.7. Suites 137

4.1. Notations de Landau

Le temps d'exécution d'un algorithme est une fonction de la taille de ses instances. Dans le cas d'une complexité temporelle, nous considérons typiquement une fonction

$$f: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}.$$

Elle associe à toute taille n le coût temporel $f(n)$.

Pour comparer de telles fonctions f lorsque n devient grand, nous utilisons les notations de Landau.

Notation	Idée intuitive	Nom
O	$\leq$	grand O
o	$<$	petit o
Ω	$\geq$	oméga
ω	$>$	petit oméga
Θ	$=$	thêta

Ces notations décrivent le comportement asymptotique des fonctions.

4.1.1. Notation O

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

La fonction f est **dominée asymptotiquement** par g , noté

$$f \in O(g),$$

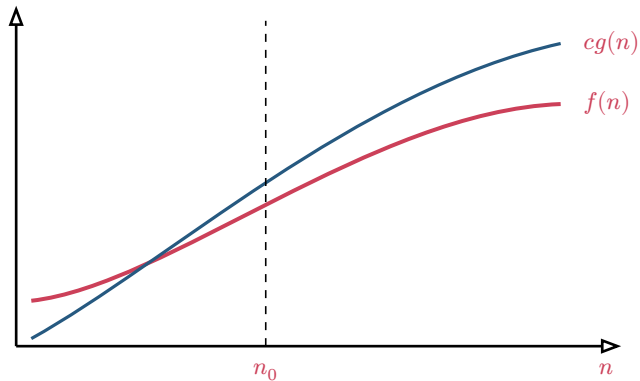
s'il existe des constantes $c > 0$ et $n_0 \in \mathbb{N}$ telles que

$$f(n) \leq cg(n)$$

pour tout $n \geq n_0$.

Interprétation : à partir d'une certaine taille, f reste sous un multiple constant de g .

La notation O fournit donc une **borne supérieure asymptotique**.



Lorsque $f \in O(g)$, pour tout $n \geq n_0$, la courbe de $f(n)$ reste sous celle de $cg(n)$.

4.1.2. Notation Ω

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

La fonction f domine asymptotiquement g , noté

$$f \in \Omega(g),$$

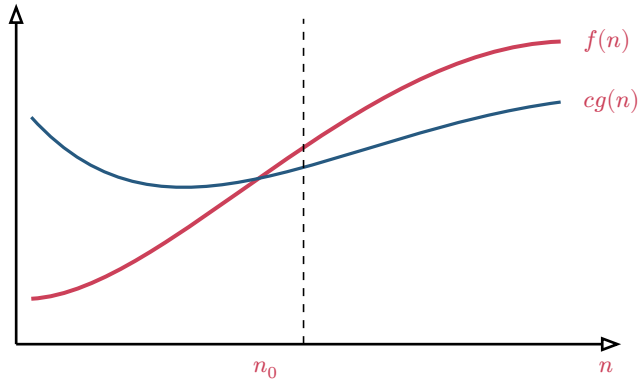
s'il existe des constantes $c > 0$ et $n_0 \in \mathbb{N}$ telles que

$$f(n) \geq cg(n)$$

pour tout $n \geq n_0$.

Interprétation : à partir d'une certaine taille, f reste au-dessus d'un multiple constant de g .

La notation Ω fournit donc une borne inférieure asymptotique.



Lorsque $f \in \Omega(g)$, pour tout $n \geq n_0$, la courbe de $f(n)$ reste au-dessus de celle de $cg(n)$.

4.1.3. Notation Θ

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

Les fonctions f et g ont le même ordre de croissance, noté

$$f \in \Theta(g),$$

si $f \in O(g)$ et $f \in \Omega(g)$.

Autrement dit, il existe des constantes $c_1, c_2 > 0$ et $n_0 \in \mathbb{N}$ telles que

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

pour tout $n \geq n_0$.

Interprétation : à partir d'une certaine taille, f est comprise entre deux multiples constants de g .

Les fonctions f et g ont donc le même comportement asymptotique.

4.1.4. Notations o et ω

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

- $f \in o(g)$ si, pour toute constante $c > 0$, il existe $n_0 \in \mathbb{N}$ tel que $f(n) \leq cg(n)$ pour tout $n \geq n_0$.
Dans ce cas, f croît **strictement moins vite** que g .
- $f \in \omega(g)$ si, pour toute constante $c > 0$, il existe $n_0 \in \mathbb{N}$ tel que $f(n) \geq cg(n)$ pour tout $n \geq n_0$.
Dans ce cas, f croît **strictement plus vite** que g .

Ainsi,

- O et Ω autorisent une croissance au même rythme ;
- o et ω expriment au contraire une domination **stricte**.

4.1.5. Critères par limites

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

Supposons qu'il existe un $m \in \mathbb{N}$ tel que pour tout $n \geq m$, $g(n) > 0$.

Les limites fournissent des critères particulièrement utiles. En effet, en posant

$$\ell := \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)},$$

- si $\ell \in \mathbb{R}$, alors $f \in O(g)$;
- si $\ell = 0$, alors $f \in o(g)$;
- si $\ell \in \mathbb{R} \setminus \{0\} \cup \{\infty\}$, alors $f \in \Omega(g)$;
- si $\ell = \infty$, alors $f \in \omega(g)$;
- si $\ell \in \mathbb{R} \setminus \{0\}$, alors $f \in \Theta(g)$.

Attention : les définitions de O , o , Ω , ω et Θ ne nécessitent pas que le quotient $\frac{f(n)}{g(n)}$ possède une limite. Une limite peut permettre de **conclure rapidement** mais elle **ne remplace pas** les démonstrations basées sur les définitions générales des notations de Landau.

4.1.6. Propriétés

Transitivité.

- $f \in O(g)$ et $g \in O(h)$ implique $f \in O(h)$;
- $f \in o(g)$ et $g \in o(h)$ implique $f \in o(h)$;
- $f \in \Omega(g)$ et $g \in \Omega(h)$ implique $f \in \Omega(h)$;
- $f \in \omega(g)$ et $g \in \omega(h)$ implique $f \in \omega(h)$;
- $f \in \Theta(g)$ et $g \in \Theta(h)$ implique $f \in \Theta(h)$.

Symétrie.

- $f \in \Theta(g)$ si et seulement si $g \in \Theta(f)$.

Quelques conséquences :

- pour démontrer $f \in \Omega(g)$, on peut de manière équivalente démontrer $g \in O(f)$;
- pour démontrer $f \in o(g)$, on peut de manière équivalente démontrer $g \in \omega(f)$;
- pour démontrer $f \in \Theta(g)$, on peut de manière équivalente démontrer $g \in \Theta(f)$.

Réflexivité.

- $f \in O(f)$;
- $f \in \Omega(f)$;
- $f \in \Theta(f)$.

Symétrie transposée.

- $f \in O(g)$ si et seulement si $g \in \Omega(f)$;
- $f \in o(g)$ si et seulement si $g \in \omega(f)$.

4.2. Polynômes

Soit $d \in \mathbb{N}$.

Un **polynôme** en n de **degré** d est une fonction de la forme

$$p(n) = \sum_{i=0}^d a_i n^i = a_0 + a_1 n + a_2 n^2 + \dots + a_d n^d,$$

où les $a_0, a_1, \dots, a_d$ sont des constantes et $a_d \neq 0$. Le nombre a_d est le **coefficient dominant** de p .

Exemples

- $3n^4 + 28n - 4$ est de degré 4 et son coefficient dominant est 3 ;
- $n^3 + n^2 + n + 1$ est de degré 3 et son coefficient dominant est 1 ;
- $100n^{18}$ est de degré 18 et son coefficient est 100 ;
- -1 est de degré 0 et son coefficient dominant est -1 .

Propriété

Soit p un polynôme de degré d dont le coefficient dominant est positif.

Alors

- $p \in \Theta(n^d)$;
- $p \in O(n^m)$ pour tout $m \geq d$;
- $p \in \Omega(n^m)$ pour tout $0 \leq m \leq d$.

Exemple

Soit $p(n) := 15n^5 - 4n^2 + 10n - 5$.

Alors

- $p \in \Theta(n^5)$;
- $p \in O(n^5)$, $p \in O(n^6)$ et $p \in O(n^{1024})$;
- $p \in \Omega(n^5)$, $p \in \Omega(n^4)$ et $p \in \Omega(1)$.

Exemple 1

Montrons que $2n^2 \in O(n^2)$.

Il faut trouver $c > 0$ et $n_0 \in \mathbb{N}$ tels que $2n^2 \leq cn^2$ pour tout $n \geq n_0$.

En choisissant $c := 2$ et $n_0 := 0$, l'inégalité est immédiate.

On retrouve aussi le résultat par

$$\lim_{n \rightarrow \infty} \frac{2n^2}{n^2} = 2.$$

Exemple 2

Montrons que $2n^2$ n'appartient pas à $O(n)$.

Si $2n^2 \leq cn$ pour tout n assez grand, alors $2n \leq c$ pour tout n assez grand, ce qui est impossible.

Exemple 3

Montrons que $2n^2 - 3n + 5 \in O(n^3)$.

Nous avons

$$\begin{aligned} 2n^2 - 3n + 5 &\leq 2n^2 - 3n + 5 && \text{pour tout } n \geq 0 \\ &\leq 2n^2 + 3n + 5 && \text{pour tout } n \geq 0 \\ &\leq 2n^3 + 3n^3 + 5n^3 && \text{pour tout } n \geq 1 \\ &\leq 10n^3 && \text{pour tout } n \geq 1. \end{aligned}$$

Ainsi, $c := 10$ et $n_0 := 1$ conviennent.