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

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 grand 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

- **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 $f : X \to Y$: 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 ».

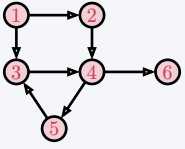
2.2.2. Problème de l'accessibilité

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 $c : [n] \times [n] \to \mathbb{N}$.

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

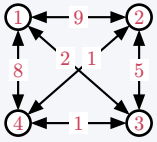
$$c(\pi(1), \pi(2)) + c(\pi(2), \pi(3)) + c(\pi(3), \pi(1)) = c(2,3) + c(3,1) + c(1,2) = 1 + 2 + 7 = 10.$$

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 donction 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 $0 \leq i \leq j \leq$ `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

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 = input()` $) = 2 + \tau($ `y = input()` $) = 3 + \tau($ `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 $\tau($`if C: I_1 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

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

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

$$n = \texttt{v\_1} - \texttt{v\_0} + 1.$$

Le coût total τ est

$$\begin{aligned}\tau &= \tau(\texttt{i = v\_0}) + (n+1)\tau(\texttt{i <= v\_1}) + n\tau(\texttt{I}) + n\tau(\texttt{i = i + 1}) \\ &= 1 + \tau(\texttt{v\_0}) + (n+1)(1 + \tau(\texttt{v\_1})) + n\tau(\texttt{I}) + 2n \\ &= 3n + 2 + \tau(\texttt{v\_0}) + (n+1)\tau(\texttt{v\_1}) + n\tau(\texttt{I}).\end{aligned}$$

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 $f_1, f_2 : \mathbb{Z} \to \mathbb{Z}$.

- **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 $f : \mathbb{Z} \to \mathbb{Z}$.

- **Décalage des indices.**

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

où $\alpha \leq \beta$, $\alpha, \beta \in \mathbb{Z}$, $f : \mathbb{Z} \to \mathbb{Z}$ 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 :

$$\begin{aligned}
\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 &= \sum_{i=0}^{n-1}(i-1-0+1)\times 1 && \text{(Addition multiple.)}\\
&= \sum_{i=0}^{n-1} i && \text{(Simplification directe.)}\\
&= \frac{(n-1)n}{2} && \text{(Somme des premiers entiers.).}
\end{aligned}$$

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,

$$\tau(\texttt{indice\_du\_plus\_grand(i, t)}) = c_1 + c_2 + ic_3 + (i-1)c_4 + \ell c_5 + (i-1)c_6 + c_7.$$

- Dans le **meilleur cas**, `j_max` n'est jamais mis à jour, donc $\ell = 0$ et
$$\tau(\texttt{indice\_du\_plus\_grand(i, t)}) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_6) + c_7.$$
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
$$\tau(\texttt{indice\_du\_plus\_grand(i, t)}) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_5 + c_6) + c_7.$$
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

$$\tau(\texttt{tri\_selection(t)}) = c_8 + c_9 + nc_{10} + (n-1)(c_{11} + c_{12} + c_{13}) + \sum_{i=1}^{n-1} \tau(\texttt{indice\_du\_plus\_grand(i + 1, t)}).$$

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**, $\tau($ `indice_du_plus_grand(i, t)` $) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_6) + c_7$ et ainsi,

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

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

Dans le **pire cas**, $\tau(\texttt{indice\_du\_plus\_grand(i, t)}) = c_1 + c_2 + ic_3 + (i-1)(c_4 + c_5 + c_6) + c_7$ et ainsi,

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

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

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,

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

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

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

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

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

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 considéres :

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 $\frac{n}{2}$.

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.