
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	100

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 ».

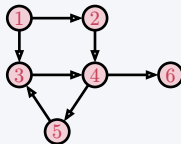
2.2.2. Problème de l'accessibilité

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

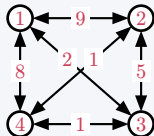
$$c(\pi(1), \pi(2)) + c(\pi(2), \pi(3)) + c(\pi(3), \pi(1)) = c(2, 3) + c(3, 1) + c(1, 2) = 1 + 2 + 7 = 10.$$

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. Tri par sélection 96
 - 3.5.2. Tri par insertion 98

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 x , notons

$$\tau(x)$$

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

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 $*$) ;
- $\tau(x = 0; y = \text{input}()) = 2 + \tau(y = \text{input}())$.

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.