Chapitre 0

Rappels

Sommaire

Ce chapitre consiste en de **brefs** rappels concernant des notions supposées connues pour le restant de ce cours, à savoir :

1. la complexité algorithmique,
2. la programmation orientée objet en Python,
3. les techniques classiques de démonstration, et
4. la description d'algorithmes en pseudocode.

Il est très fortement conseillé de se (re)mettre à niveau rapidement en cas de besoin, en consultant les notes des cours que vous avez suivis sur ces notions ou les ouvrages de référence mentionnés en fin de ce chapitre (Cormen et al. [1], Skiena [2] pour les aspects algorithmiques, et Swinnen [3] pour la programmation en Python).

0.1 Complexité algorithmique

Un algorithme doit être correct, et donc se terminer, et également être le plus efficace possible : cela signifie qu'il doit être rapide (en termes de temps d'exécution) et économe en ressources (espace de stockage, mémoire utilisée).

On ne veut pas passer son temps à implémenter, débugger et tester tous les algorithmes que l'on rencontre juste pour décider lequel est le meilleur ; d'une part parce que cette approche serait inefficace, et d'autre part parce que les résultats obtenus varieraient selon le langage choisi et la machine utilisée. On aura donc plutôt recours à des outils théoriques permettant de mesurer l'efficacité de ces algorithmes. On voudra également que ces mesures nous

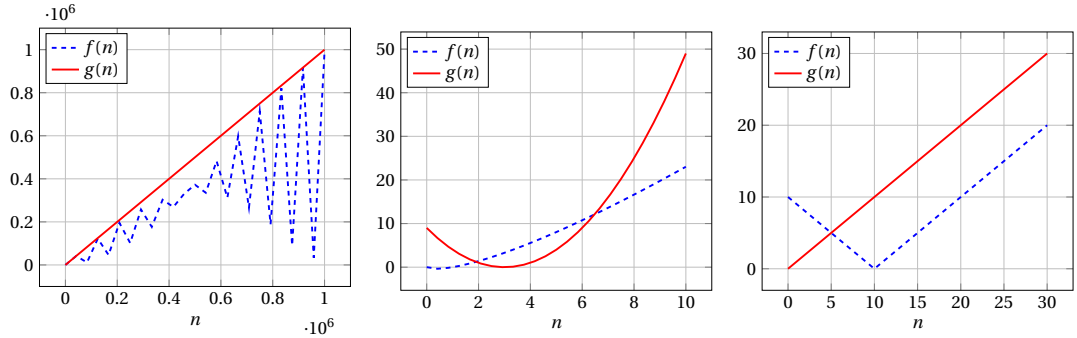
permettent de prédire le temps d'exécution au pire cas de ces algorithmes et de savoir lesquels pourront s'exécuter rapidement même sur des données de grande taille. À ces fins, on aura donc recours à une **approximation** de ce temps de calcul en fonction de la taille des données, représentée par la notation $O(\cdot)$.

Définition 1. Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\, n_0 \in \mathbb{N}, \exists\, c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près. Les valeurs absolues importent dans la définition, mais on verra qu'en pratique, on ne devra jamais s'en soucier : les fonctions qui nous intéresseront mesureront toujours le temps d'exécution d'un algorithme, qui sera forcément positif.

Exemple 1. Voici quelques cas où $f(n) = O(g(n))$:



La notation $O(\cdot)$ va nous permettre de quantifier l'efficacité d'un algorithme sous certaines hypothèses :

Définition 2. La **complexité** d'un algorithme est la mesure **asymptotique** de son temps d'exécution **dans le pire cas**. Elle s'exprime à l'aide de la notation $O(\cdot)$ en fonction de la taille des données reçues en entrée.

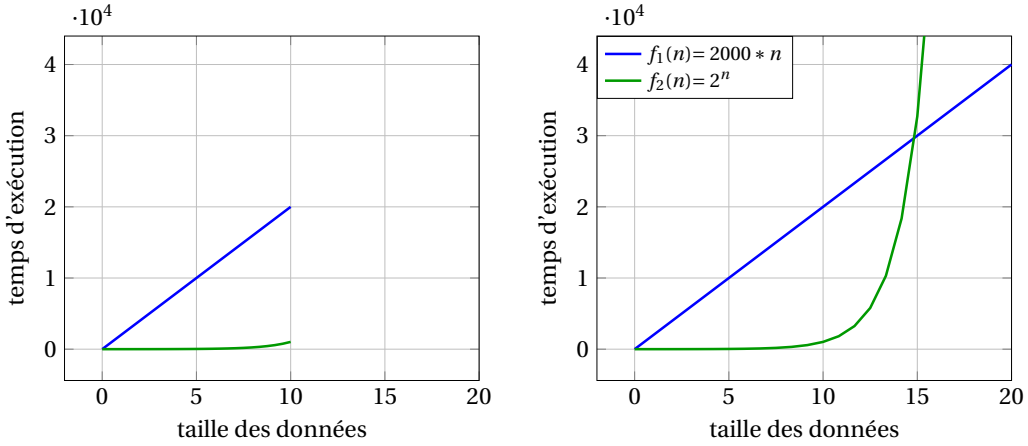
Les deux précisions sur le caractère de cette mesure importent :

1. **asymptotique** signifie que l'on s'intéresse à des données très grandes ; en effet, comme on le verra plus bas, les petites valeurs ne sont pas assez informatives ;
2. **"dans le pire cas"** signifie que l'on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ; on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.

L'exemple suivant illustre la raison pour laquelle on a intérêt à s'intéresser à l'aspect asymptotique.

Exemple 2. Les deux graphiques suivants comparent les temps d'exécution de deux algorithmes ; la version (a) ne contient pas assez d'informations pour prendre une bonne décision, puisqu'on ne va pas assez loin dans les tailles de données pour se rendre compte que l'algorithme qui semble être le plus performant est en fait beaucoup plus

lent, ce que montre bien la version (*b*) :



(fin exemple 2)

0.1.1 Hypothèses, calcul et règles de simplification

Le calcul de la complexité d'un algorithme se base sur les hypothèses suivantes :

- chaque **instruction basique** (affectation d'une variable de type basique ou comparaison de deux types basiques, $+$, $-$, $*$, $/$, ...) consomme un temps constant, représenté par la notation $O(1)$;
- chaque **itération** d'une boucle rajoute la complexité de ce qui est effectué dans le corps de cette boucle ;
- chaque **appel de fonction** rajoute la complexité de cette fonction ;
- pour obtenir la complexité de l'algorithme, on additionne le tout.

On aura aussi recours aux simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 3 (simplifications)**.** Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

et on a donc $g(n) = O(n^3)$.

0.1.2 Combinaison des complexités

On additionne les complexités d'opérations en séquence :

$$O(f_1(\cdot)) + O(f_2(\cdot)) = O(f_1(\cdot) + f_2(\cdot))$$

Et on applique le même traitement aux branchements conditionnels :

si <*condition*> **alors** ; $O(g(\cdot))$
    `# instructions (1)` $O(f_1(\cdot))$
sinon ;
    `# instructions (2)` $O(f_2(\cdot))$

$\Big\}$ $= O(g(\cdot) + f_1(\cdot) + f_2(\cdot))$

Les règles de simplification vues plus haut s'appliqueront ensuite. En particulier, une somme d'opérations constantes sera constante : $O(1) + O(1) = O(1)$.

Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations. La complexité d'une boucle se calcule comme suit :

`# en supposant qu'on a m itérations`
tant que <*condition*> **faire** ; $O(g(\cdot))$
    `# instructions` $O(f(\cdot))$

$\Big\}$ $= O(m * (g(\cdot) + f(\cdot)))$

Si deux boucles se suivent, il s'agit de blocs d'opérations en séquence : on additionnera donc leurs complexités. En revanche, si la boucle 1 contient la boucle 2 et qu'elles sont indépendantes, alors chaque itération de la boucle 1 nous fera exécuter toutes les itérations de la boucle 2, et il faudra donc *multiplier* leurs complexités.

0.1.3 Application des règles

Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque "partie" de l'algorithme ;
2. on combine ces complexités conformément aux règles qu'on vient de voir ;
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues :
   — élimination des constantes, et
   — conservation du (des) terme(s) dominant(s).

Exemple 4 (factorielle de n)**.** Calculons la complexité de l'algorithme suivant implémenté sous la forme d'une fonction Python et qui calcule la factorielle de n :

```
1  def factorielle(n):
2      """Renvoie la factorielle du naturel n."""
3      fact = 1              # initialisation: O(1)
4      i = 2                 # initialisation: O(1)
5
6      while i <= n:         # n-1 itérations: O(n)
7          fact = fact * i   # multiplication: O(1)
8          i = i + 1         # incrémentation: O(1)
9
10     return fact           #         renvoi: O(1)
```

Conformément aux règles vues plus haut, la complexité de la procédure est $O(1) + O(n) * O(1) + O(1) = O(n)$.

0.1.4 Calcul de la complexité grâce aux limites

Lorsqu'on doit classer deux fonctions $f(n)$ et $g(n)$ en termes de notation $O(\cdot)$, la première étape consiste à appliquer les règles de simplifications et à voir si les fonctions simplifiées

sont plus faciles à comparer. Il existe cependant des cas où l'on ne voit pas directement, même sur les fonctions simplifiées, laquelle domine l'autre en termes de croissance asymptotique. Dans ce cas, il est utile de procéder à un calcul de limite : intuitivement, si la croissance de la fonction $g(n)$ domine celle de $f(n)$ pour des grandes valeurs de n, alors la valeur de $\lim_{n\to+\infty} f(n)/g(n)$ doit être une constante (éventuellement nulle).

Exemple 5. Quelle fonction croît le plus rapidement : $\ln n$ ou $\sqrt{n}$? Pour le déterminer, calculons $\lim_{n\to+\infty} \ln n/\sqrt{n}$; en appliquant la règle de l'Hospital, on obtient

$$\lim_{n\to+\infty} \frac{\ln n}{\sqrt{n}} \overset{RH}{=} \lim_{n\to+\infty} \frac{1/n}{1/2\sqrt{n}} = \lim_{n\to+\infty} \frac{2\sqrt{n}}{n}.$$

Cette limite tend vers 0 (ce que l'on peut prouver par le même procédé), et l'on en conclut donc que $O(\ln n)$ est inférieure à $O(\sqrt{n})$.

0.1.5 Classification d'algorithmes

La relation "être en grand O de ..." est réflexive, mais pas symétrique. En effet :

— $f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$: par exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;

— $f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$: par exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$.

Lorsque la relation est symétrique, les deux fonctions comparées sont jugées "équivalentes".

Définition 3. Deux fonctions $f(n)$ et $g(n)$ sont **équivalentes** du point de vue de la notation $O(\cdot)$ si

$$f(n) = O(g(n)) \text{ et } g(n) = O(f(n)).$$

On peut "ranger" les fonctions équivalentes dans une même **classe**. La Figure 1 montre quelques classes de complexité fréquentes (par ordre croissant en termes de $O(\cdot)$).

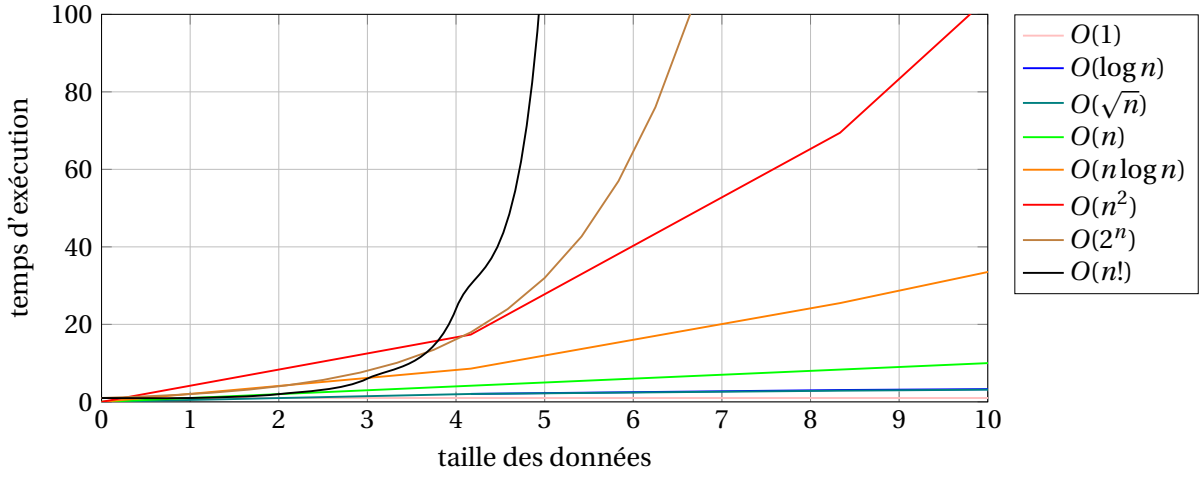


FIGURE 1 – Quelques classes de complexité fréquentes.

Le Tableau 1 montre la croissance du temps d'exécution en fonction de la taille des données sur base d'unités concrètes de temps et permet de mieux se rendre compte des impacts pratiques.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

TABLEAU 1 – Comparaison en pratique du temps de calcul qu'impliquent les différentes complexités [2].

Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité. On fait une première distinction entre les deux classes suivantes :

1. les algorithmes dits **polynomiaux**, dont la complexité est en $O(n^k)$ pour une certaine **constante** k ; si k fait partie des données que l'algorithme reçoit, il s'agit par définition d'une **variable** et l'on ne peut donc pas parler de complexité polynomiale ;
2. les algorithmes dits **exponentiels**, dont la complexité ne peut pas être majorée par une fonction polynomiale.

Si seul le temps de calcul nous intéresse, et qu'on a le choix entre un algorithme A en $O(2^n)$ et un algorithme B en $O(\log n)$, il n'y a en général pas à hésiter : on choisira B pour résoudre notre problème. Il arrivera cependant qu'on doive choisir entre plusieurs algorithmes de même complexité (par exemple en $O(n)$), et il faudra alors pousser l'analyse plus loin pour déterminer quel algorithme choisir. On en verra quelques exemples dans les exercices.

0.2 Programmation orientée objet en Python

La programmation orientée objet a été couverte en long et en large dans le cours de Java, et l'on ne reviendra donc pas sur le vocabulaire associé. On se contentera ici de présenter la manière dont Python l'implémente, en soulignant au besoin les différences éventuelles entre les deux langages.

0.2.1 Une première classe basique

Passons directement à la manière dont on définit les classes en Python. Le format le plus simple est le suivant :

```
1 class MaClasse(object):
2     """Classe vide."""
3     def __init__(self):
4         """Initialise les membres de la classe."""
5         pass
```

```
6
7       def ma_methode(self, parametre):
8           """Une méthode ne faisant rien."""
9           pass
```

Pour des raisons qu'on ne développera pas ici, il est recommandé que les classes écrites héritent de la classe `object`, ce qu'on réalise avec la syntaxe `class Descendant(Ancetre)`.

Pour définir une méthode d'une classe, il faut respecter deux règles :

1. le premier paramètre de cette méthode **doit** être `self`. Il s'agit d'un mot-clé dont le rôle est exactement celui du `this` en Java et en C++, dont on expliquera l'utilisation un peu plus bas.
2. la définition de la méthode **doit** se trouver dans celle de la classe, ce que l'on réalise avec l'indentation appropriée.

La méthode `__init__` est particulière : elle joue le rôle du constructeur de la classe et sera automatiquement appelée lors de la création d'une instance de la classe. On ne le fait pas soi-même : l'instruction `x = MaClasse()` créera ainsi l'objet `x` qui sera une instance de `MaClasse`, et `MaClasse.__init__` sera automatiquement exécutée. C'est dans la méthode `__init__` qu'on initialisera le cas échéant des membres de la classe.

Plusieurs limitations gravitent autour de la méthode `__init__`. On les signale ici pour être complet, mais elles ne nous gêneront pas dans l'élaboration des classes vues dans ce cours :

— une classe doit posséder exactement une méthode `__init__`. Si l'on ressent le besoin d'en créer d'autres pour des usages différents, il faudra contourner cette limitation en utilisant des paramètres avec des valeurs par défaut, ou avoir recours à aux techniques permettant d'avoir un nombre arbitraire de paramètres (`__init__(self, *args)` ou `__init__(self, *args, **kwargs)`).

— la méthode `__init__` ne peut rien renvoyer.

La syntaxe à utiliser pour se servir des méthodes de la classe varie selon l'endroit dans le code où l'on se trouve :

— si l'on a besoin d'utiliser `ma_methode` dans la définition de la classe, on utilisera la syntaxe `self.ma_methode(mon_parametre)` (il n'y a donc plus de `self` entre les parenthèses).

— si l'on en a besoin en dehors de la définition de la classe, par exemple après avoir créé l'instance `x`, on écrira `x.ma_methode(mon_parametre)` : le `x` prend alors la place du `self` dans cet appel.

0.2.2 Champs privés, protégés et publics

Les notions de champs `private`, `protected` et `public` n'existent pas en Python. Il est possible de simuler une variable privée en faisant précéder son nom de deux *underscores* — par exemple : `self.__pas_touche = 0`. Comme on le voit dans le terminal, ce champ n'est pas accessible simplement :

```
1  >>> class MaClasse(object):
2  ...     def __init__(self):
3  ...         self.__pas_touche = 0
4  ...
5  >>> x = MaClasse()
```

```
>>> print(x.__pas_touche)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MaClasse' object has no attribute '__pas_touche'
```

Il est possible de contourner cette protection, mais les développeurs Python supposent que leurs collègues sont des utilisateurs responsables et ne vont pas tenter de le faire. Ainsi, le préfixe `__` est plus vu comme une convention indiquant qu'il ne faut pas essayer de modifier la variable concernée.

Cette approche nous permet d'éviter une longue liste de *getters* et *setters* : s'il faut accéder directement à un champ de la classe, on peut le faire en lecture et en écriture. Dans l'optique où l'on implémentera des graphes dans ce cours, on fera toutefois l'effort de fournir certaines de ces méthodes, non pas pour protéger les données, mais dans le but de masquer les détails d'implémentation : il sera commode de supposer qu'on dispose d'une classe `Graphe` avec une API correspondante, et selon ce qui nous intéresse, on détaillera ou non l'implémentation des méthodes concernées.

0.2.3 Programmation générique

L'intérêt de la programmation objet dans le cadre de ce cours est de pouvoir utiliser plusieurs classes différentes pour implémenter nos graphes, et d'avoir des algorithmes qui fonctionnent quelle que soit la représentation sous-jacente.

On sera parfois amené dans nos fonctions à devoir renvoyer des graphes qui devront être du même type que celui donné en entrée. Pour ce faire, on aura recours à la fonction `type` de Python : comme `type(x)` nous renvoie le type de l'objet `x`, l'expression `type(x)()` appellera le constructeur de la classe de cet objet. On pourra donc écrire des fonctions génériques de la manière suivante :

```
def ma_fonction(graphe):
    resultat = type(graphe)()
    # ... faire des trucs sur graphe et resultat ...
    return resultat
```

0.3 Techniques de preuves

On aura à plusieurs reprises besoin de *prouver* ou de *démontrer* des propriétés relatives aux structures manipulées, qui nous permettront ensuite de prouver que les algorithmes examinés sont corrects, ou d'en concevoir de plus efficaces. Il n'est pas inutile de rappeler le principe de fonctionnement de quelques-unes des techniques de preuves qui nous serviront. Il est fortement recommandé de s'entraîner à les mettre en œuvre.

— les preuves par *induction* consistent à prouver qu'une propriété est vraie en partant de l'hypothèse qu'elle est vraie pour une certaine valeur (ce qu'on appelle l'*hypothèse d'induction*). Il nous faut prouver la propriété pour un ou plusieurs *cas de base* (par exemple $n = 0$), ce qui est généralement assez simple ; ensuite, on suppose la propriété vraie pour une certaine valeur (par exemple un certain naturel $n = k$), et on doit montrer que cela implique que la propriété est vraie pour la valeur suivante (par

exemple $n = k + 1$). Si c'est bien le cas, la propriété est alors vraie pour toute valeur de k à partir du ou des cas de base identifié(s).

— les preuves par *contraposition* consistent à prendre le problème "dans l'autre sens" : on les rencontre souvent dans les cas où il semble difficile de montrer qu'une propriété A implique une autre propriété B. Dans ce cas, au lieu de prouver $A \Rightarrow B$, on tente de montrer que $\neg B \Rightarrow \neg A$, ce qui est équivalent (cf. cours de logique).

— les preuves par *l'absurde* ou par *contradiction* ressemblent aux preuves par contraposition. On les rencontre dans les cas où il semble difficile de prouver qu'une propriété est vraie. Dans ces cas-là, il est parfois plus facile de montrer qu'il n'est pas possible que la propriété soit fausse : le procédé consiste alors à supposer que la propriété est fausse, et à montrer que cela mène à une contradiction.

Idéalement, on préfère obtenir des preuves "directes", c'est-à-dire un raisonnement qui utilise les hypothèses connues pour en déduire directement ce qu'on cherche à prouver. Toutefois, les idées peuvent manquer, et dans ce genre de situation, il sera bon d'avoir le réflexe de recourir à une des stratégies rappelées ci-dessus.

0.4 Pseudocode

Pour finir, quelques instructions de rappel sur le pseudocode s'imposent. Le *pseudocode* est un langage fictif de description d'algorithmes, à mi-chemin entre le langage naturel et un langage de programmation. Notre but, en utilisant ce langage plutôt qu'un vrai langage de programmation, est de nous affranchir des détails d'implémentation au profit du raisonnement mis en œuvre pour élaborer l'algorithme décrit. Un bon pseudocode évite donc au maximum tous les détails d'implémentation liés au langage (allocation mémoire, pointeurs, ...), mais doit aussi être suffisamment précis pour que la traduction de l'algorithme décrit en pseudocode vers un code source dans le langage choisi puisse se faire facilement et sans ambigüité.

Le format que nous adopterons sera le suivant :

1. un nom d'algorithme présenté sous la forme d'une fonction écrite à l'aide de CETTE POLICE, suivi d'une liste de paramètres entre parenthèses ;
2. la description des entrées de l'algorithme : c'est-à-dire ce que représentent chacun des paramètres fournis, avec leur type, et les éventuelles hypothèses à leur sujet ;
3. la description de la sortie de l'algorithme (c'est-à-dire ce qu'il renvoie) ou le cas échéant de son résultat (donc les éventuelles modifications que subissent ses paramètres).

Un format similaire s'applique à la description des *problèmes* à résoudre. Par exemple, le *problème de décision* — un problème dont la réponse est "oui" ou "non" — consistant à déterminer si un tableau donné contient un élément donné se formaliserait comme suit :

RECHERCHE D'ÉLÉMENT
Données: un tableau T de n éléments, un élément x.
Question: T contient-il x ?

Le problème d'optimisation consistant à trouver le minimum d'un tableau s'exprimerait comme suit :

RECHERCHE DU MINIMUM
Données: un tableau T de $n > 0$ éléments.
Objectif: renvoyer le plus petit élément de T.

Et un algorithme trivial résolvant ce problème s'exprimerait en pseudocode comme suit :

Algorithme 1 : TROUVERMINIMUM(T)

Entrées : un tableau T non vide.
Sortie : le plus petit élément de T.

```
1 minimum ← T[0];
2 pour i allant de 1 à longueur(T)−1 faire
3 |   si T[i] < minimum alors minimum ← T[i];
4 renvoyer minimum;
```

De nombreux autres exemples apparaîtront plus loin dans le cours.

Bibliographie

[1] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, ET C. STEIN, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[2] S. S. SKIENA, *The Algorithm Design Manual*, Springer, 2ème édition, 2008.

[3] G. SWINNEN, *Apprendre à programmer avec Python 3*, Eyrolles, 2012. https://inforef.be/swi/download/apprendre_python3_5.pdf.