

*1<sup>ère</sup> NSI — Thème 6: Algorithmique*

# Algorithmique & Mise au Point de Programmes

Lycée Fustel de Coulanges, Massy

Marc Biver, février 2024, *v0.1*

*Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.*

# Table des matières

<b>1</b>	<b>Point d'étape – où est-on / où va-t-on ?</b>	<b>3</b>
1.1	Ce qu'on a couvert jusqu'à présent . . . . .	3
1.2	Ce dont on va parler dans ce nouveau chapitre . . . . .	3
1.3	Comment on va procéder . . . . .	4
<b>2</b>	<b>Introduction à la conception d'algorithmes</b>	<b>5</b>
2.1	Écrire un algorithme : spécification . . . . .	5
2.2	Écrire un algorithme : pseudo-code . . . . .	6
2.3	Tester un algorithme . . . . .	9
<b>3</b>	<b>Preuve et complexité d'algorithmes</b>	<b>10</b>

# 1 Point d'étape – où est-on / où va-t-on ?

## 1.1 Ce qu'on a couvert jusqu'à présent

- Rudiments de l'architecture physique d'un ordinateur – le modèle de Von Neumann.
- Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
- Introduction à Python, et plus spécifiquement :
  - Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
  - Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
  - Un type dit "construit" – les listes.
- Un peu de théorie : la représentation des types et valeurs de base en machine (les entiers naturels et relatifs, les réels, les alphanumériques).
- Un peu plus de théorie : introduction à la logique booléenne (que l'on n'a couverte que très rapidement – on y reviendra en fin d'année).
- Un retour à la pratique : le traitement de données en table, la manipulation de fichiers dans Python, et des types de données plus complexes — les dictionnaires, les listes de dictionnaires...

## 1.2 Ce dont on va parler dans ce nouveau chapitre

On a donc fait des "sauts" de la théorie vers la pratique, puis vers la théorie de nouveau — pour finalement atterrir ici, dans ce chapitre sur l'algorithmique qui se trouve presque exactement au "milieu" de cet axe théorie–pratique.

Comme je vous l'ai dit à plusieurs reprises dans les parties précédentes – et spécialement dans celle portant sur le traitement de données en table, l'algorithmique, vous en faites déjà : je vous pose des problèmes par le biais de notebooks Jupyter dans Capytale v2 et vous réfléchissez à comment les résoudre. Dit autrement, *vous concevez des algorithmes*.

Ce que l'on va faire dans ce nouveau chapitre c'est formaliser cette démarche, la structurer, puis étudier quelques exemples d'algorithmes beaucoup plus avancés que ce que l'on a vu jusqu'à présent.

Spécifiquement, on va parcourir le chemin suivant :

- Introduction à l'algorithmique – étapes de conception et de rédaction d'un algorithme ;
- Pratique : tests d'algorithmes / de programmes ;
- Preuve d'algorithmes ;
- Complexité d'algorithmes ;
- Etude de certains algorithmes spécifiques :
  - Algorithme de recherche dichotomique dans un tableau ;
  - Algorithmes de tri - par sélection, par insertion ;
  - Algorithmes des k plus proches voisins – algorithme d'apprentissage, "machine learning" ;

- Algorithmes gloutons.
- Pratique : mise au point de programmes ; programmation défensive.

### 1.3 Comment on va procéder

Comme dit plus haut, on est ici à la frontière entre la théorie et la pratique – on va donc avoir un fonctionnement hybride en classe :

- **Prise de notes essentielle** — vous commencez à connaître les cours que je vous fournis, ils sont *très* longs. Ils doivent vous servir de référence, vous permettre surtout de bien revoir les corrections d'exercices, mais votre savoir, lui, doit venir de votre prise de notes ;
- Plusieurs exercices sur papier qu'on fera en classe et dont il sera très important que vous gardiez une trace ;
- En parallèle et en complément, quelques applications / exercices sur machine.

## 2 Introduction à la conception d'algorithmes

Quelques questions pour commencer...

→ Qu'est-ce qu'un algorithme (indice : c'est constitué de trois parties) ?

→ Parmi les éléments suivants, qu'est-ce qui est un algorithme, qu'est-ce qui n'en est pas ?

- a. Une recette de gâteau au chocolat ;
- b. La liste des présidents de la V<sup>ème</sup> République ;
- c. Les règles du jeu d'échecs ;
- d. Les règles à appliquer pour résoudre une équation du 2<sup>nd</sup> degré ;
- e. Les instructions de montage d'un meuble Ikea.



### Définition:

L'algorithmique est :

- La conception (et la production) d'algorithmes ;
- Leur étude – leur analyse, la mesure de leur fiabilité (est-ce qu'il répond vraiment au problème posé ?) et de leur efficacité (est-ce qu'il le fait en un temps acceptable ?).

Nous allons dans ce cours nous intéresser à ces deux composantes, en commençant, dans ce chapitre, par la première dont les étapes peuvent se résumer ainsi :

1. Énoncé d'un problème à résoudre ;
2. Spécification de l'algorithme – nom, entrées, sorties ;
3. Explicitation de la démarche en pseudo-code – description du traitement des données qui va permettre de résoudre le problème ;
4. Traduction en langage de programmation.

La 1<sup>ère</sup> et la 4<sup>ème</sup> étape sont à la marge de notre propos ici :

- L'énoncé d'un problème à résoudre par le biais d'un programme informatique est une activité à part entière (souvent appelée "expression de besoin" dans le monde professionnel). Vous y avez un petit peu touché dans le cadre de vos projets, mais dans l'ensemble, dans le contexte de la NSI, les problèmes vous sont posés – et votre rôle consiste à savoir les résoudre ;
- La traduction en langage de programmation, que dans le contexte de la NSI nous réalisons en Python, a fait l'objet d'un pan du cours distinct ; nous allons évidemment y revenir en partie ici, mais la syntaxe Python n'est pas l'objet de notre étude ici.

### 2.1 Écrire un algorithme : spécification

Avant d'écrire un algorithme il faut bien définir ce que l'on veut faire et à partir de quoi ; il s'agit de donner **une spécification au problème**. Pour cela on doit :

- Donner un nom explicite à l'algorithme — *par exemple CuissonGâteauChocolat* ;
- Décrire les conditions d'utilisation de l'algorithme, les données qu'il attend en entrée et les conditions dans lesquelles il va pouvoir être exécuté, sa **précondition** — *par exemple "Beurre et Lait non périmés"* :

- De même, décrire le résultat attendu, sa **postcondition**, la nature des données renvoyées et à quoi elles correspondent — *par exemple "gâteau rond, moelleux, et succulent"*.

Cette étape de spécification est fondamentale – c’est en quelque sorte la "carte d’identité" de notre algorithme, ce qui va permettre à quelqu’un qui ne le connaît pas de le comprendre sans avoir besoin de lire son code. On va donc la transcrire dans notre code Python en tête de la fonction lui correspondant – et c’est exactement ce que je vous demande de faire dans vos projets.



### Méthode:

La transcription de la spécification d’un algorithme en tête de la fonction Python lui correspondant s’appelle "**le docstring**" ou "**la documentation**" de la fonction. Il est inscrit entre deux séries de trois apostrophes – par exemple :

```
1 def MaxNombre(n1, n2):
2     '''
3     Fonction dont les paramètres sont entiers ou réels.
4     Elle renvoie la plus grande de ces deux valeurs ou, en cas
5     d'égalité, la première valeur.
6     '''
7     if n1 < n2:
8         return n2
9     else:
10        return n1
```

## Exercice 1: Rédaction d’une spécification de fonction

Considérez la fonction suivante :

```
1 def Fonction(n1, n2, n3):
2     if n1 < n2 < n3 or n3 < n2 < n1:
3         return n2
4     elif n1 < n3 < n2 or n2 < n3 < n1:
5         return n3
6     elif n2 < n1 < n3 or n3 < n1 < n2:
7         return n1
8     elif n1 == n2 and n2 == n3:
9         return n1
10    else:
11        return None
```

Est-ce que ce qu’elle fait est clair d’entrée de jeu ?

Rédigez la docstring de cette fonction pour remédier à cela.

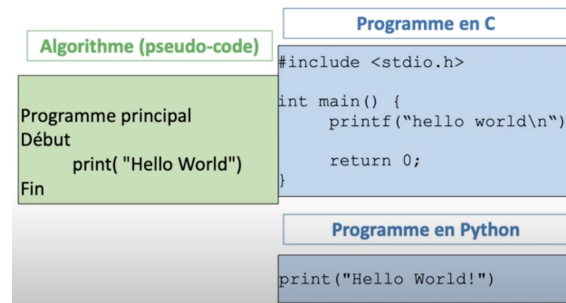
## 2.2 Ecrire un algorithme : pseudo-code

Si l’on se réfère aux étapes listées plus haut, on en est maintenant au moment où l’on sait *ce que va réaliser* notre programme, *ce qu’il va prendre en entrée*, et *ce qu’il va retourner en sortie*. Il s’agit à présent d’explicitier le **comment** – quelles sont les étapes qui vont être effectuées pour résoudre le problème ? Quel traitement va-t-on appliquer aux données en entrée pour produire les données en sortie ?

On a déjà utilisé à de multiples reprises le pseudo-code dans ce cours – donc (*en*

*théorie*) vous devriez déjà être convaincus de son intérêt et savoir l'utiliser. Nous allons donc passer directement à quelques exercices d'application – en rappelant tout de même au préalable les principes et règles suivants :

- Le pseudo-code est une façon de décrire un algorithme pour qu'il soit compréhensible "entre humains".
- Le pseudo-code est indépendant du langage de programmation – un algorithme convenablement écrit devrait en théorie pouvoir être implémenté aussi aisément en Python qu'en C ou qu'en JavaScript<sup>1</sup>. A titre d'exemple, voici un "Hello World" en deux langages de programmation distincts, mais partant du même pseudo-code :



- Conséquence : les règles de syntaxe de pseudo-code sont inspirées des éléments communs à la plupart des langages de programmation.
- Il n'y a pas de pseudo-code universel – le seul principe à respecter, c'est que les règles de syntaxe appliquées soient bien définies, comprises et partagées par tous-tes celles et ceux qui seront amené-e-s à lire les algorithmes.

Ce dernier point implique qu'il y ait quand même une ossature de règles minimales dans un contexte donné – comme pour ce cours par exemple :



### Règles pseudo-code 1<sup>ère</sup> NSI:

- On spécifie explicitement en début d'algorithme les entrées attendues et les sorties prévues ;
- On utilise une flèche vers la gauche (" $\leftarrow$ " ou " $\triangleleft$ ") pour affecter des variables ;
- On utilise l'indentation pour délimiter les fonctions, les conditions, les boucles...<sup>a</sup>
- On explicite la fin de toute structure (fonction, condition, boucle...) débütée ;
- On n'hésite pas à inclure des commentaires pour expliquer les étapes – et dans ce cas, on les préfixe d'une flèche vers la droite (" $\rightarrow$ " ou " $\triangleright$ ")
- ... et c'est tout !

<sup>a</sup> Ici on triche un peu puisque l'indentation est spécifique à Python – mais pas tant que ça puisque cela restera compréhensible même pour une implémentation dans un autre langage.

1. C'est évidemment un peu simpliste d'écrire ceci ainsi, mais en théorie le principe est vrai : lorsque vous rédigez un algorithme en pseudo-code, vous devriez ne pas avoir de langage de programmation spécifique en tête.

Pour illustrer ces principes, voici le pseudo-code d'une fonction (qu'on a déjà vue dans un chapitre précédent, d'ailleurs) prenant une liste de réels positifs en entrée et qui en renvoie le maximum :

**Entrée:** *liste* dont tous les éléments  $\in \mathbb{R}_+$

**Sortie:** *Max*  $\in \mathbb{R}_+$

```
1: fonction TROUVEMAX(liste)
2:   Max  $\leftarrow$  0
3:   pour tout Element de liste faire
4:     si Element > Max alors                                 $\triangleright$  On a trouvé un nouveau max
5:       Max  $\leftarrow$  Element
6:     fin si
7:   fin pour
8:   retourner Max
9: fin fonction
```

### Exercice 2: Rédaction d'algorithmes en pseudo-code

En appliquant les principes énoncés ci-dessus, rédigez les algorithmes suivants :

- Un algorithme qui prend deux nombres en entrée et affiche leur somme.
- Un algorithme qui calcule la somme des  $N$  premiers entiers naturels.
- Un algorithme qui génère les  $N$  premiers termes de la séquence de Fibonacci. (rappel : c'est une suite de nombres dont les deux premiers sont 0 et 1 et dont chaque élément est la somme des deux précédents – donc : 0, 1, 1, 2, 3, 5, 8, 13, etc...).
- Un algorithme qui vérifie si une chaîne de caractères est un palindrome (se lit de la même manière dans les deux sens).
- [**BONUS**] — Supposez que vous avez une liste *Lst* de nombres réels ordonnée (c'est-à-dire où  $Lst[0] \leq Lst[1] \leq (\dots) \leq Lst[n]$ ) et que vous cherchez à savoir si elle contient un nombre  $N$  en particulier (la conclusion sera donc booléenne - Vrai ou Faux). Rédigez puis comparez deux algorithmes :
  - Un qui fait une recherche "normale", en commençant à un bout et en parcourant toute la liste ;
  - Un autre qui effectue une recherche dite "dichotomique" :
    - Il regarde le milieu de la liste :
      - S'il est supérieur à  $N$  il élimine la partie supérieure de la liste ;
      - S'il est inférieur à  $N$  il élimine la partie inférieure de la liste ;
    - Il recommence l'opération jusqu'à soit trouver  $N$  soit aboutir à une impossibilité.

Que peut-on dire des efficacités relatives de ces deux algorithmes ?



### Exercice 3: Traduction de pseudo-code en Python

- a. Choisissez l'un des deux derniers algorithmes de la question précédente (éléments de la suite de Fibonacci ou vérification de palindrome) et traduisez-le en une fonction Python.
- b. **Expliquez comment vous allez tester votre fonction.**
- c. **[BONUS]** — Traduisez en Python l'algorithme de recherche dichotomique.

## 2.3 Tester un algorithme

Cette étape est cruciale dans le développement d'un programme informatique car les erreurs dans les phases de rédaction de l'algorithme et de traduction en langage de programmation sont plus que fréquentes – elles sont systématiques dès lors qu'un programme atteint un certain niveau de complexité.

### 3 Preuve et complexité d'algorithmes