1ère 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

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, il 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) ?*

*Un algorithme est une **suite finie d'instructions** permettant de **résoudre un problème**. Il est structuré en trois parties :*

a. *L'entrée des données ;*
b. *Le traitement des données ;*
c. *La sortie des données*

→*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^{ème} République ;*
c. *Les règles du jeu d'échecs ;*
d. *Les règles à appliquer pour résoudre une équation du 2nd degré ;*
e. *Les instructions de montage d'un meuble Ikea.*

a. *Oui c'en est un – ingrédients en entrée, étapes de confection, gâteau en sortie. C'est bien la résolution d'un problème (en l'occurrence "comment fabriquer un gâteau au chocolat ?").*
b. *Non ce n'en est pas un – c'est une liste d'informations, pas des étapes à suivre.*
c. *Ce n'en est pas un non plus – ce n'est qu'une liste de principes. En revanche on pourrait les utiliser pour mettre en œuvre un algorithme qui joue aux échecs (et résoud le problème "comment jouer – et gagner – une partie d'échecs ?).*
d. *Oui c'en est un – c'est même écrit dans la description ("résoudre").*
e. *Oui c'en est un également.*

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 1ère et la 4ème 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* `CuissonGateauChocolat` ;

— 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 connait 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 :

```
def Fonction(n1, n2, n3):
    if n1 < n2 < n3 or n3 < n2 < n1:
        return n2
    elif n1 < n3 < n2 or n2 < n3 < n1:
        return n3
    elif n2 < n1 < n3 or n3 < n1 < n2:
        return n1
    elif n1 == n2 and n2 == n3:
        return n1
    else:
        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.

On aura noté deux problèmes ici – l'absence de docstring, mais également l'absence d'un nom explicite à la fonction ("Fonction", ce n'est franchement pas génial...). Remédions à tout cela :

```
def Mediane(n1, n2, n3):
    '''
    Fonction qui prend en entrée trois valeurs numériques (int ou float).
    Elle renvoie la médiane de ces trois valeurs quand celle-ci existe.
    Si elle n'existe pas (deux valeurs égales, la troisième différente),
    elle renvoie None.
    Par ex: Mediane(0, 2, 1) renverra 1;
    Mediane(1, 1, 1) renverra 1;
    Mediane(0, 2, 2) renverra None.
    '''
    if n1 < n2 < n3 or n3 < n2 < n1:
        return n2
    elif n1 < n3 < n2 or n2 < n3 < n1:
        return n3
    elif n2 < n1 < n3 or n3 < n1 < n2:
        return n1
    elif n1 == n2 and n2 == n3:
        return n1
    else:
        return None
```

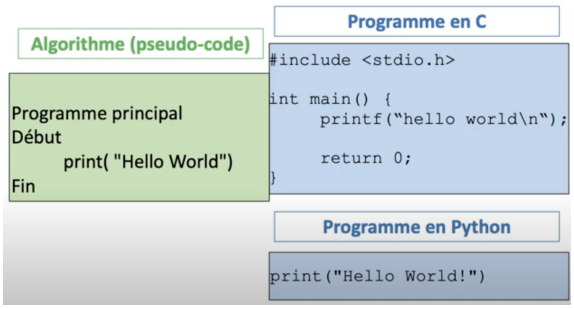
Vous remarquerez l'inclusion d'exemples dans la docstring – il ne faut surtout pas hésiter à y recourir, c'est ce qu'il y a de plus parlant pour quelqu'un qui découvre votre code !

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'expliciter 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 :

a. Le pseudo-code est une façon de décrire un algorithme pour qu'il soit compréhensible "entre humains".
b. 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 [1]. A titre d'exemple, voici un "Hello World" en deux langages de programmation distincts, mais partant du même pseudo-code :



c. Conséquence : les règles de syntaxe de pseudo-code sont inspirées des éléments communs à la plupart des langages de programmation.
d. 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 1ère NSI:

a. On spécifie explicitement en début d'algorithme les entrées attendues et les sorties prévues ;
b. On utilise une flèche vers la gauche ("←" ou "◁") pour affecter des variables ;
c. On utilise l'indentation pour délimiter les fonctions, les conditions, les boucles... [a]
d. On explicite la fin de toute structure (fonction, condition, boucle...) débutée ;
e. 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 ("→" ou "▷")
f. ... et c'est tout !

a. 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 ← 0
3:     pour tout Element de liste faire
4:         si Element > Max alors                ▷ On a trouvé un nouveau max
5:             Max ← 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 :

a. Un algorithme qui prend deux nombres en entrée et affiche leur somme.

b. Un algorithme qui calcule la somme des N premiers entiers naturels.

c. 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...).

d. Un algorithme qui vérifie si une chaîne de caractères est un palindrome (se lit de la même manière dans les deux sens).

e. [**BONUS**] — Supposez que vous avez une liste Lst de nombres réels ordonnée (c'est-à-dire où $Lst[0] \leq Lst[1] \leq (...) \leq Lst[n]$) 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 ?

Il n'y a jamais (ou rarement) une solution algorithmique unique à un problème – ce qui suit n'est donc que des propositions de solution (qui sont, évidemment, valides – mais pas uniques).

a. ***Entrée:*** *deux nombres a et b* $\in \mathbb{R}$
Sortie: $a + b$

```
1: fonction Somme(a, b)
```

```
    Resultat ← a + b
    retourner Resultat
fin fonction
```

b. **Entrée:** $N \in \mathbb{N}$
Sortie: *la somme des N premiers entiers naturels*

```
fonction SommeEntiers(N)
    Resultat ← 0                          ▷ On initialise le résultat à 0
    pour i allant de 1 à N faire
        Resultat ← Resultat + i
    fin pour
    retourner Resultat
fin fonction
```

c. **Entrée:** $N \in \mathbb{N}$ *avec* $N \geq 2$ ▷ *On précise bien les valeurs acceptables en entrée*
Sortie: *aucune* ▷ *On va afficher les résultats, pas les retourner*

```
fonction Fibonacci(N)
    ValeurPrec ← 0
    ValeurCour ← 1
    Afficher ValeurPrec ▷ On affiche le 0 puis on rentre dans la boucle
    pour i allant de 2 à N faire
        Afficher ValeurCour
        Suivant ← ValeurPrec + ValeurCour
        ValeurPrec ← ValeurCour
        ValeurCour ← Suivant
    fin pour
fin fonction
```

d. **Entrée:** *chn, une chaine de caractères*
Sortie: $Vrai$ *ou* $Faux$ *selon si la chaine est un palindrome ou non*

```
1:  fonction EstPalindrome(chn)
2:      Long ← longueur(chn)
3:      si Long est pair alors
4:          Milieu ← Long/2
5:      sinon
6:          Milieu ← (Long − 1)/2
7:      fin si
8:      pour i allant de 0 à Milieu faire
9:          si chn[i] ≠ chn[Long − i] alors
10:             retourner Faux
11:         fin si
12:     fin pour
13:     retourner Vrai
14: fin fonction
```

Quelques remarques sur cet algorithme :

— *Lignes 8 et 9 : le but ici est de bien comprendre l'algorithme – avec cette formule on comprend bien ce qu'on fait : on part du début pour aller jusqu'au milieu. Strictement parlant les formules ne marchent pas (en Python chn[Long] donnerait une erreur) – mais on s'en fiche :*

la démarche est claire ici, et c'est le but.

- *Ligne 3 : on vérifie la parité de la longueur puisque dans le cas des longueurs impaires on ignorera le caractère du milieu. Vous noterez qu'on dit ici "si Long est pair" sans expliquer comment on fait – la difficulté de l'algorithme n'étant pas là, c'est tout à fait acceptable (on imagine qu'il y aura un algorithme pour une fonction "EstPair" explicité ailleurs).*
- *Lignes 10 et 13 : c'est une démarche qu'on a déjà utilisée ailleurs pendant notre cours, qui est très classique, et qu'il faut impérativement maîtriser : on part du principe que quelque chose est vrai — ici "chaîne est un palindrome" — et dès qu'on trouve une preuve du contraire (ici : un caractère est différent de son homologue de l'autre côté) on retourne "Faux", c'est-à-dire qu'on arrête immédiatement la fonction puisqu'on connaît son résultat. Si on arrive au bout de la boucle, c'est qu'on a pas trouvé de preuve que c'est faux – donc c'est vrai et c'est ce qu'on retourne.*

Exercice 3: Traduction de pseudo-code en Python

a. Choisissez l'un des 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.

Pour la suite de Fibonacci, on pourra utiliser le code suivant :

```
1  def Fibonacci(N):
2      '''
3      Fonction qui prend en entrée un entier N > 1 et ne renvoie rien.
4      Elle affiche les N premiers termes de la séquence de Fibonacci
5      Par ex: Fibonacci(5) affichera successivement 0, 1, 1, 2, et 3.
6      '''
7      ValPrec = 0
8      ValCur = 1
9      print(ValPrec)
10     for i in range(1, N):
11         print(ValCur)
12         Suivant = ValPrec + ValCur
13         ValPrec = ValCur
14         ValCur = Suivant
```

Pour la tester, il suffira de la lancer avec différentes valeurs de N et de vérifier que le résultat est correct. Assez rapidement on constatera que si l'on passe une valeur de N qui n'est pas dans le champ des possibles (1, par exemple, ou -2) alors on rencontre une erreur – nous reviendrons plus tard dans ce cours sur comment gérer cela.

Pour le palindrome, on pourra utiliser le code suivant :

```python
def EstPalindrome(chn):
    '''
    Fonction qui prend en entrée une chaine de caractères.
    Elle renvoie Vrai si la chaine est un palindrome, Faux sinon.
    Par ex: EstPalindrome("abba") renverra Vrai;
    EstPalindrome("abbac") renverra Faux.
    '''
    Long = len(chn)
    if Long % 2 == 0:
        Milieu = Long // 2
        # Utilisation de la division entière ("//" et non "/" pour la
        ↪  division simple) de manière à avoir des entiers et non des float
    else:
        Milieu = (Long - 1) // 2

    for i in range(Milieu):
        if chn[i] != chn[Long - 1 - i]: # Sans le -1 on aura une erreur
            return False

    return True
```

Pour la tester, il faudra réfléchir un peu plus à tous les cas de figures possibles :

- *Chaine de longueur paire qui est un palindrome ;*
- *Chaine de longueur paire qui n'en est pas un ;*
- *Chaine de longueur impaire qui est un palindrome ;*
- *Chaine de longueur impaire qui n'en est pas un.*

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