Chapitre 9
Les premiers algorithmes

Ce chapitre présente les premiers **algorithmes** que l'on rencontre en informatique : chercher le minimum ou le maximum d'une liste, additionner des valeurs, calculer une moyenne, rechercher une valeur, compter des occurrences. Ces algorithmes reposent tous sur le même outil : le **parcours séquentiel** d'un tableau, c'est-à-dire la visite de ses éléments un par un, du premier au dernier. On verra d'abord comment décrire un algorithme avec un **pseudo-code**, puis comment l'**implémenter** en Python, enfin comment l'appliquer à des objets.

I. Les pseudo-codes et l'implémentation simple

Un algorithme est une suite finie d'instructions qui, à partir de **données d'entrée**, produit un **résultat de sortie**. Avant de l'écrire dans un langage de programmation, on peut le décrire en français ou en **pseudo-code** : une écriture à mi-chemin entre le langage courant et un langage de programmation. Le pseudo-code permet de se concentrer sur la **logique** de l'algorithme, sans se soucier de la syntaxe exacte de Python.

Exemple 9.1.
On veut afficher tous les éléments d'une liste. Le pseudo-code est simple : « Pour chaque élément de la liste, afficher l'élément. » La traduction en Python est tout aussi directe. ■

```
for element in [3, 7, 2]:
    print(element)
```

Exemple 9.2.
Un algorithme s'écrit souvent avec une entrée, une sortie et un corps. On précise ce que fait l'algorithme avant de détailler comment il le fait. ■

```
Entree : une liste L
Sortie : la somme des elements de L

somme = 0
Pour chaque element de L:
    somme = somme + element
```

```
Renvoyer somme
```

R On utilise l'anglicisme **pseudo-code** (sans tiret) pour désigner cette écriture intermédiaire. Le mot **pseudo** vient du grec et signifie « faux » : ce n'est pas un vrai code exécutable, mais il en a l'allure.

1. L'algorithme du minimum et du maximum

Dans ces algorithmes, on cherche le minimum (ou le maximum) en comparant tous les éléments au plus petit (ou au plus grand) élément déjà rencontré. Si un élément est plus petit que le plus petit connu, alors ce nouvel élément devient le minimum ; sinon, l'ancien minimum reste le minimum.

▪ Exemple 9.3.
Voici le pseudo-code de la recherche du minimum d'une liste. La variable `m` mémorise le minimum courant : elle est initialisée avec le premier élément, puis mise à jour à chaque fois que l'on rencontre une valeur plus petite. ▪

```
Entree : Liste
Sortie : m minimum

Algo
m = L[0]
Pour chaque element de Liste:
    Faire
        Si element < m Alors m = element
    Fin Faire
Renvoyer m
```

■ **Exemple 9.4.**
La traduction en Python de ce pseudo-code est directe : la boucle `for` parcourt les valeurs, le `if` compare et met à jour. ■

```
L = [8, 3, 10, 5]
m = L[0]
for element in L:
    if element < m:
        m = element
print(m)
```

■ **Exemple 9.5.**
On peut écrire la même recherche dans une **fonction**, pour la réutiliser sur n'importe quelle liste. La fonction renvoie le minimum avec `return`. ■

```
def minimum(L):
    m = L[0]
    for element in L:
        if element < m:
            m = element
    return m
```

■ **Exemple 9.6.**
Pour obtenir le **maximum**, on inverse simplement le sens de la comparaison : on met à jour `M` quand un élément est plus grand que le maximum courant. ■

```
def maximum(L):
    M = L[0]
    for element in L:
        if element > M:
            M = element
    return M
```

R Pourquoi initialiser `m` avec `L[0]` plutôt qu'avec `0` ? Parce que `0` pourrait être plus petit que tous les éléments de la liste : le résultat serait alors faux. En initialisant avec le premier élément, on part d'une valeur qui est **réellement dans la liste**.

■ **Exemple 9.7.**
Le piège de l'initialisation à `0` apparaît avec des valeurs toutes négatives : aucune ne dépasse `0`, le maximum affiché est donc `0`, ce qui est faux. ■

```
L = [-3, -7, -2]
M = 0
for element in L:
    if element > M:
        M = element
print(M)
```

2. L'algorithme de recherche naïf

On veut savoir si une **valeur cible** est présente dans une liste, et parfois à quelle **position** elle se trouve. La recherche **naïve** (ou recherche séquentielle) consiste à parcourir la liste du début à la fin en comparant chaque élément à la cible, et à s'arrêter dès qu'on l'a trouvée — ou à la fin de la liste si elle n'y est pas.

■ **Exemple 9.8.**
Voici le pseudo-code de la recherche naïve, qui répond par `True` si la valeur est trouvée et `False` sinon. Le booléen `trouve` sert de **drapeau** (flag) : il mémorise le fait que l'on a trouvé la valeur. ■

```
Entree : Liste, valeur
Sortie : True si valeur est dans Liste, False sinon

Algo
trouve = False
Pour chaque element de Liste:
    Faire
        Si element == valeur Alors trouve = True
    Fin Faire
Renvoyer trouve
```

■ **Exemple 9.9.**
La traduction en Python : la variable `trouve` démarre à `False` et passe à `True` dès que la cible est rencontrée. ■

```
L = [5, 3, 8, 1]
valeur = 8
trouve = False
for element in L:
    if element == valeur:
        trouve = True
print(trouve)
```

■ **Exemple 9.10.**
Dans une fonction, on peut utiliser `return True` dès que l'on trouve la valeur : l'algorithme s'arrête immédiatement, sans parcourir la fin de la liste. Si la boucle se termine sans avoir trouvé, on renvoie `False`. ■

```
def contient(L, valeur):
    for element in L:
        if element == valeur:
            return True
    return False
```

■ **Exemple 9.11.**
Si l'on veut la **position** de la valeur plutôt que sa présence, on mémorise l'indice courant. La valeur `-1` sert de code « non trouvé », car `-1` n'est pas un indice valide pour une recherche par indice positif. ■

```
def position(L, valeur):
    for i in range(len(L)):
        if L[i] == valeur:
            return i
    return -1
```

R Si la valeur apparaît plusieurs fois, la recherche naïve avec `return` renvoie la **première** occurrence. Pour obtenir la dernière, il ne faut pas s'arrêter pendant la boucle : on écrase l'indice à chaque occurrence.

■ **Exemple 9.12.**
Cet algorithme est appelé **naïf** car il examine les éléments un par un, sans exploiter d'information sur la liste (par exemple un ordre). Dans le pire des cas, il faut parcourir toute la liste : son coût est **linéaire**, proportionnel au nombre d'éléments.
■

II. Des algorithmes sur des objets

Les algorithmes de parcours fonctionnent sur des listes de **n'importe quel type** : nombres, chaînes, mais aussi objets. On reprend ici le principe du chapitre sur les classes : on définit un objet, on en crée une liste, puis on fait tourner nos algorithmes sur cette liste en comparant un **attribut** des objets.

■ **Exemple 9.13.**
On modélise des cadeaux avec une classe `Cadeau`. Chaque cadeau a un numéro et une valeur aléatoire comprise entre 1 et 1000. La méthode `__str__` permet d'afficher proprement un cadeau. ■

```
import random as rd

class Cadeau:
    def __init__(self, numero):
        self.numero = numero
        self.valeur = rd.randint(1, 1000)

    def __str__(self):
        return f"cadeau numero {self.numero} : valeur {self.valeur}"
```

■ **Exemple 9.14.**
On crée une liste de 500 cadeaux par compréhension. ■

```
liste_cadeau = [Cadeau(i) for i in range(500)]
```

■ **Exemple 9.15.**
On peut afficher tous les cadeaux avec une simple boucle. ■

```
for cadeau in liste_cadeau:
    print(cadeau)
```

■ **Exemple 9.16.**
L'algorithme du minimum s'applique aux cadeaux : on compare les **attributs** `valeur` des objets. Le minimum est le cadeau dont la valeur est la plus petite. ■

```
meilleur = liste_cadeau[0]
for cadeau in liste_cadeau:
    if cadeau.valeur < meilleur.valeur:
        meilleur = cadeau
print(meilleur)
```

■ **Exemple 9.17.**
La recherche naïve s'applique de la même manière : on cherche, par exemple, un cadeau dont la valeur vaut exactement 42. ■

```
trouve = False
for cadeau in liste_cadeau:
    if cadeau.valeur == 42:
        trouve = True
print(trouve)
```

R Les algorithmes ne changent pas selon le type des éléments : seule la **comparaison** change (ici `cadeau.valeur < meilleur.valeur` au lieu de

`element < m`). C'est cela, la puissance du parcours séquentiel : un même algorithme sert pour des listes de nombres, de chaînes ou d'objets.

III. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : calcul mental d'expressions Python, lecture de code court, prédiction d'affichage, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 9.1.

Rappels : types, calculs et booléens. Répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
2 + 2.0
```

2. Que vaut l'expression suivante ?

```
5 ** 3
```

3. Que vaut l'expression suivante ?

```
(3 > 2) or (5 < 4)
```

4. Que vaut l'expression suivante ?

```
not (7 <= 3)
```

■

■ Exercice 9.2.

Algorithmes : prédire le résultat d'un petit algorithme sur un tableau.

1. Qu'affiche le programme suivant ?

```
t = [8, 3, 10, 5]
m = t[0]
for v in t:
    if v < m:
        m = v
print(m)
```

2. Qu'affiche le programme suivant ?

```
t = [12, 5, 18, 9]
M = t[0]
for v in t:
    if v > M:
```

```
        M = v
print(M)
```

3. Qu'affiche le programme suivant ?

```
notes = [10, 14, 12]
somme = 0
for n in notes:
    somme = somme + n
print(somme / len(notes))
```

■

■ **Exercice 9.3.**

Rappels : boucles et fonctions.

1. Combien de fois s'exécute la boucle `while` suivante, partant de `i = 1` ?

```
i = 1
while i < 20:
    i = i * 3
```

2. Qu'affiche le programme suivant ?

```
for i in range(2, 6, 2):
    print(i)
```

3. Que renvoie l'appel `f(3)` ?

```
def f(x):
    return x * x - x
```

4. Que renvoie l'appel `g(-3)` ?

```
def g(x):
    if x > 0:
        return "positif"
    return "negatif"
```

■

■ **Exercice 9.4.**

Algorithmes et comptage : compléter et prédire.

1. Quelle ligne complète correctement l'algorithme du maximum ?

```
t = [4, 9, 2, 7]
M = t[0]
for v in t:
```

```
    if v > M:
        M = ____
print(M)
```

2. Qu'affiche le programme suivant, qui compte les valeurs strictement supérieures à 10 ?

```
t = [15, 8, 12, 20]
compteur = 0
for v in t:
    if v > 10:
        compteur = compteur + 1
print(compteur)
```

3. Qu'affiche le programme suivant, où l'on a écrit `somme + 1` au lieu de `somme + v` ?

```
t = [7, 2, 9]
somme = 0
for v in t:
    somme = somme + 1
print(somme)
```

■

Exercices d'application

Les exercices d'application écrivent des algorithmes de parcours sur des tableaux, du plus simple au plus difficile. Les trois premiers reprennent les exercices du chapitre (jeu de cartes, recherche, cadeaux).

■ **Exercice 9.5.**
Algorithme du minimum à la main. Avec votre jeu de cartes (en prenant des cartes de 1 à 10) mélangé, réaliser l'algorithme du minimum avec vos mains : on pose la première carte face visible, puis on compare chaque carte suivante ; si elle est plus petite que la carte posée, on la remplace.

1. Décrire, avec vos mots, ce que fait la main à chaque étape.
2. Combien de comparaisons effectue-t-on pour un jeu de 10 cartes ?

■

■ **Exercice 9.6.**
Liste aléatoire et minimum. Faire une liste `L` par compréhension de 10 nombres aléatoires entre 1 et 300, puis écrire un script qui affiche le minimum de `L`.

```
import random as rd
```

```
L = [rd.randint(1, 300) for i in range(10)]
# Ecrire ici l'algorithme du minimum
```

■

■ **Exercice 9.7.**
Fonction maximum. Écrire une fonction qui prend en entrée une liste `L` et qui renvoie le maximum de cette liste. Tester avec la liste `[3, 7, 2, 9, 4]`.

```
def maximum(L):
    # Ecrire ici l'algorithme du maximum
```

■

■ **Exercice 9.8.**
Recherche à la main. Avec votre jeu de cartes (en prenant des cartes de 1 à 10) mélangé, chercher la position de la carte numéro 7. Écrire un dessin ou un schéma de cette recherche de valeur : on montre les cartes examinées une à une, et la position trouvée. ■

■ **Exercice 9.9.**
Présence d'une valeur. Faire une liste `L` par compréhension de 10 nombres aléatoires entre 1 et 300, puis écrire un script qui affiche `True` si 7 est dans la liste et `False` sinon.

```
import random as rd

L = [rd.randint(1, 300) for i in range(10)]
# Ecrire ici la recherche naive
```

■

■ **Exercice 9.10.**
Fonction de recherche. Écrire une fonction qui prend en entrée une liste `L` et une valeur `valeur` et qui renvoie `True` si `valeur` est dans la liste et `False` sinon. Tester avec la liste `[5, 3, 8, 1]` et la valeur `8`, puis la valeur `10`.

```
def contient(L, valeur):
```

```
    # Ecrire ici la recherche naive
```

■

■ **Exercice 9.11.**
Fonction somme. Écrire une fonction `somme_liste` qui prend en entrée une liste de nombres et qui renvoie leur somme. Tester avec la liste `[12, 8, 15, 10]`.

```
def somme_liste(L):
    # Ecrire ici le parcours
```

■

■ **Exercice 9.12.**
Fonction moyenne. Écrire une fonction `moyenne` qui prend en entrée une liste de notes et qui renvoie leur moyenne. On rappelle que la moyenne est la somme divisée par le nombre de valeurs.

```
def moyenne(notes):
    # Ecrire ici le parcours
```

■

■ **Exercice 9.13.**
Fonction position. Écrire une fonction `position` qui prend en entrée une liste `L` et une valeur `valeur`, et qui renvoie l'indice de la **première** occurrence de `valeur` dans `L`, ou `-1` si `valeur` n'y est pas. Tester avec la liste `[6, 3, 8, 5]` et la valeur `8`.

```
def position(L, valeur):
    # Ecrire ici la recherche
```

■

■ **Exercice 9.14.**
Somme des valeurs paires. Écrire un script qui affiche la somme des valeurs paires de la liste `L = [1, 4, 7, 10, 3]`. On rappelle qu'un nombre est pair si son reste dans la division par 2 vaut 0.

```
L = [1, 4, 7, 10, 3]
# Ecrire ici le parcours avec test de parite
```

■

■ **Exercice 9.15.**
Fonction comptage. Écrire une fonction `compte_occurrences` qui prend en entrée une liste `L` et une valeur `cible`, et qui renvoie le nombre de fois où `cible` apparaît dans `L`. Tester avec la liste `[3, 6, 3, 8, 3]` et la cible `3`.

```
def compte_occurrences(L, cible):
    # Ecrire ici le comptage
```

■

■ **Exercice 9.16.**
Fonction compte_positifs. Écrire une fonction `compte_positifs` qui prend en entrée une liste de nombres et qui renvoie le nombre de valeurs **strictement positives** de cette liste. Tester avec la liste `[18, -5, 11, 0, 14]`.

```
def compte_positifs(L):
    # Ecrire ici le comptage
```

■

■ **Exercice 9.17.**
Fonction liste_paires. Écrire une fonction `liste_paires` qui prend en entrée une liste de nombres et qui renvoie une **nouvelle liste** contenant uniquement les valeurs paires, dans l'ordre. Tester avec la liste `[1, 4, 7, 10, 3, 8]`.

```
def liste_paires(L):
    # Ecrire ici la collecte
```

■

■ **Exercice 9.18.**
Cadeaux : le minimum. On reprend la classe `Cadeau` du cours (numéro et valeur aléatoire entre 1 et 1000).

1. Créer une liste de 100 cadeaux par compréhension.

2. Écrire un script qui affiche le cadeau de plus petite valeur.
3. Écrire une fonction `cadeau_le_plus_cher` qui prend une liste de cadeaux et renvoie le cadeau de plus grande valeur.

```
import random as rd

class Cadeau:
    def __init__(self, numero):
        self.numero = numero
        self.valeur = rd.randint(1, 1000)
    def __str__(self):
        return f"cadeau numero {self.numero} : valeur {self.valeur}"
```

■

■ **Exercice 9.19.**

Cadeaux : la recherche naïve. On reprend la même liste de 100 cadeaux.

1. Écrire un script qui affiche `True` si un cadeau de valeur 42 existe dans la liste, et `False` sinon.
2. Écrire une fonction `position_cadeau` qui prend une liste de cadeaux et une valeur, et qui renvoie le numéro du premier cadeau dont la valeur est égale à la valeur cherchée (ou `-1` si aucun).

```
# On suppose la liste liste_cadeau deja construite
```

■

Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : déroulement de programmes, repérage d'erreurs, questions ouvertes sur les algorithmes de parcours. Ils sont classés du plus abordable au plus exigeant.

■ **Exercice 9.20.**

Compléter le pseudo-code de la moyenne. Recopier et compléter le pseudo-code suivant, qui calcule la moyenne des éléments d'une liste.

```
Entree : Liste
Sortie : la moyenne des elements

Algo
somme = 0
Pour chaque element de Liste:
    somme = somme + element
moyenne = ____
Renvoyer moyenne
```

1. Quelle expression remplace le premier ____ ?
2. Pourquoi la division doit-elle utiliser le nombre d'éléments de la liste, et pas une valeur fixe ?

■

■ **Exercice 9.21.**

Le programme suivant provoque une erreur de logique : il n'affiche pas le minimum attendu.

```
t = [4, 9, 2, 7]
m = 0
for v in t:
    if v < m:
        m = v
print(m)
```

1. Qu'affiche ce programme ? Pourquoi ce n'est pas le minimum de la liste ?
2. Corriger le programme.

■

■ **Exercice 9.22.**

Le programme suivant contient une erreur de comparaison.

```
def minimum(L):
    m = L[0]
    for v in L:
        if v > m:
            m = v
    return m
```

1. Que renvoie cette fonction pour la liste `[8, 3, 10, 5]` ? Que devrait-elle renvoyer ?
2. Expliquer pourquoi le résultat est faux.

3. Corriger la fonction.

■

■ **Exercice 9.23.**

Coût d'un parcours. On considère l'algorithme du minimum sur une liste de n éléments.

1. Combien de comparaisons effectue-t-on en tout, dans le pire des cas ?
2. Même question pour la recherche naïve : si la valeur cherchée n'est pas dans la liste, combien de comparaisons faut-il ?
3. Expliquer pourquoi on dit que le coût de ces algorithmes est **linéaire** en n.

■

■ **Exercice 9.24.**

Compléter un algorithme de comptage. On veut compter les occurrences d'une valeur `cible` dans une liste. Recopier et compléter le programme suivant, puis le tester avec la liste `[4, 7, 4, 9, 4]` et la cible `4`.

```
t = [4, 7, 4, 9, 4]
cible = 4
compteur = ____
for v in t:
    if v == cible:
        compteur = ____
print(compteur)
```

1. Quelle est la valeur initiale de `compteur` ? Pourquoi ?
2. Que renverrait le programme avec la cible `10` ? Est-ce un résultat correct ?

■