MPSI
PCSI
PTSI

Serge Bays

PRÉPAS SCIENCES

COLLECTION DIRIGÉE PAR BERTRAND HAUCHECORNE

INFORMATIQUE
TRONC COMMUN

- Cours complet et détaillé
- Nombreux exemples
- Vrai/Faux
- Exercices avec indications
- Corrections et commentaires

2e édition

ellipses

PRÉPAS SCIENCES
collection dirigée par Bertrand Hauchecorne

Informatique
Tronc commun
MPSI - PCSI - PTSI

2e édition

Serge BAYS
Agrégé de mathématiques,
professeur d'informatique en classes préparatoires
au lycée Les Eucalyptus (Nice)

ellipses

COLLECTION
PRÉPAS SCIENCES

Retrouvez tous les titres de la collection et des extraits sur www.editions-ellipses.fr

Les notices culturelles « Un scientifique » et « Un peu d'histoire »
des pages de titre des chapitres ont été rédigées par Bertrand Hauchecorne.

Les macros de cet ouvrage ont été réalisées par Nicolas Nguyen en LaTeX.

ISBN 9782340-104211
Dépôt légal : juillet 2025
©Ellipses Édition Marketing S.A.
8/10 rue la Quintinie 75015 Paris

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle et artistique n'autorisant, aux termes des alinéas 2 et 3 de l'article L. 122-5, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

Informatique commune en classes préparatoires

Le nouveau programme d'informatique commune en classe préparatoires scientifiques a été adapté pour prendre en compte les contenus des programmes du lycée. Il s'applique à la rentrée 2021.

La première partie est composée de thèmes d'études qui se situent à des niveaux de difficulté progressifs. Il s'agit au travers d'activités de découverte par la pratique de prendre de bonnes habitudes. La deuxième partie définit et précise les savoirs à maîtriser. Elle est divisée en trois sections : écriture et analyse de programmes, représentations des nombres en machines, graphes. Ces deux parties constituent le programme de première année auquel est consacré ce livre.

L'ouvrage est divisé en quatre parties. La première correspond au premier semestre et les trois autres au second semestre. L'ensemble est au plus près du programme. Quelques développements et compléments sont proposés en plus de l'essentiel.

La première partie est composée de huit chapitres qui proposent principalement des exemples et exercices en rappelant les prérequis, notions rencontrées dans le secondaire, et en pointant les connaissances nouvelles. Le travail en classe consiste au cours de travaux pratiques, à partir des connaissances du lycée, à faire émerger de nouvelles notions, acquérir peu à peu de bonnes pratiques dans l'écriture de programmes et des capacités d'analyse des algorithmes.
Le premier chapitre est consacré aux boucles simples qui permettent de parcourir les éléments d'une structure séquentielle et de chercher ou compter. Le deuxième chapitre présente des situations où l'on est amené à introduire des boucles imbriquées, des boucles dans des boucles. Il s'agit de prérequis pour aborder plus tard des algorithmes de tri par exemple. Le chapitre trois présente quelques modules et bibliothèques d'usage courant, certains probablement déjà rencontrés dans le cycle secondaire. Le chapitre quatre est centré sur l'usage de la dichotomie. Il s'agit d'une stratégie classique de type diviser pour régner. Le point de départ est un algorithme de recherche rencontré en mathématiques au lycée.
Avec le chapitre cinq de nouvelles notions apparaissent. La récursivité est introduite au travers de situations où sa place est naturelle. Le chapitre six présente quelques exemples d'utilisation d'algorithmes gloutons dont le principe est simple et qui permettent de résoudre des problèmes pouvant se rencontrer dans la vie courante. Le chapitre sept concerne la manipulation de tableaux à deux dimensions étudiée dans le cadre attrayant de la modification d'images. Le chapitre huit traite différents algorithmes de tri utilisés dans de nombreux domaines.

La deuxième partie précise les pratiques d'écriture d'un programme au chapitre neuf et d'analyse d'un programme au chapitre dix. C'est l'occasion de faire un bilan de ce qui a été rencontré au cours de la première partie : préciser le contenu d'un programme, faire des choix de structure,

de stratégie, découper un programme en sous-programmes, utiliser des fonctions en les spécifiant. Ensuite il s'agit de prouver la correction, évaluer la complexité et tester pour tenter de repérer et corriger d'éventuels bugs.

La troisième partie de l'ouvrage précise la représentation des nombres dans une machine afin de comprendre la signification d'un résultat et ce qui est faisable avec les contraintes et les prudences requises. Le chapitre onze concerne les nombres entiers, le chapitre douze les nombres flottants.

La partie quatre présente les graphes qui permettent de modéliser de nombreuses situations. Le chapitre treize apporte le vocabulaire et les différents modes de représentation. Le chapitre quatorze étudie principalement des algorithmes de parcours d'un graphe et de recherche de chemins.

Le langage de programmation Python, version 3, est utilisé dans le cadre de cet enseignement. Ce langage est connu des élèves et pratiqué dès la classe de seconde. Une annexe présente le contenu exigible au programme qui est volontairement restreint. Un objectif est d'utiliser des moyens simples pour faire émerger des idées essentielles.
Un logiciel gratuit et librement téléchargeable comme GIMP ou XnView peut être utile pour afficher les images manipulées au chapitre sept.

Une relecture soigneuse a permis d'éliminer quelques erreurs dans le texte et les programmes mais il peut en subsister. Si c'est le cas, j'espère que la lecture sera quand même agréable.

Merci à Corinne Baud, Bertrand Hauchecorne et les éditions Ellipses pour leur confiance.
Un petit coucou pour Ambroise et Kélia et une pensée très forte pour Brigitte et Michel.

Serge Bays

Sommaire

Thèmes d'étude

1. Parcours d'une structure séquentielle 3
2. Boucles imbriquées 23
3. Modules et bibliothèques 43
4. Algorithmes dichotomiques 65
5. Fonctions récursives 85
6. Algorithmes gloutons 109
7. Manipulation d'une image 133
8. Algorithmes de tri 157

Programmes et algorithmes

9. Écriture d'un programme 185
10. Analyse d'un programme 205

Représentation des nombres

11. Les entiers 231
12. Les flottants 251

Les graphes

13. Vocabulaire et représentations 271
14. Algorithmes 293

Annexe

15. Langage Python 325

Index 339

Thèmes d'étude

Chapitre 1

Parcours d'une structure séquentielle

UN SCIENTIFIQUE



Au III^e siècle avant notre ère, Alexandrie était le centre intellectuel du Monde. Une immense bibliothèque contenait toutes les connaissances de l'époque. **Ératosthène** (env. 276 av. J.-C. - env. 194 av. J.-C.), son conservateur, brillait dans toutes les disciplines, en particulier la géographie et les mathématiques. Son calcul du périmètre de la Terre s'est révélé presque exact et son nom reste attaché à la méthode du crible pour rechercher les nombres premiers.

Un peu d'histoire

L'algorithme conçu par Ératosthène pour la recherche des nombres premiers, connu sous le nom de crible, est sans doute le plus ancien de forme séquentielle. Pour obtenir, par exemple, tous les nombres premiers inférieurs à 1000, on les écrit tous à partir de 2. On laisse 2 et on raye tous ses multiples ; le plus petit nombre non rayé est alors 3 ; on le laisse et on raye tous ses multiples. On prolonge cette opération jusqu'à atteindre la partie entière de la racine carrée de 1000, soit 31. Les nombres non rayés sont les nombres premiers.

Depuis, des algorithmes de recherches de nombres premiers se sont multipliés comme celui conçu par le mathématicien et informaticien Derrick Lehmer. Ces recherches ont longtemps été considérées comme des curiosités sans application pratique. L'utilisation d'immenses nombres premiers en cryptographie leur a donné une importance cruciale.

Les boucles permettent l'exploration des éléments des tableaux unidimensionnels. Elles sont présentes dans de nombreux algorithmes étudiés au lycée. Avec elles, il est possible de déterminer la présence d'un élément particulier, d'effectuer un calcul sur les éléments.

OBJECTIFS

- Connaître quelques algorithmes classiques utilisant un parcours séquentiel :
  - être capable de lire et de modifier des éléments d'un tableau ;
  - savoir calculer une moyenne ;
  - maîtriser la recherche d'un extrémum et d'une valeur particulière ;
  - comprendre comment compter des éléments vérifiant une propriété.
- Comprendre la distinction entre un coût constant et un coût linéaire :
  - savoir évaluer le nombre d'opérations effectuées dans un programme ;
  - comprendre l'importance de la notion de coût ou de complexité.
- Découvrir l'intérêt des dictionnaires en Python :
  - comprendre le rôle d'un dictionnaire ;
  - connaître l'apport en terme de coût dans l'efficacité d'un programme.

Cours

Boucles

Un programme est composé d'une suite d'instructions, exécutées l'une après l'autre dans l'ordre où elles sont écrites, contenant des définitions de variables et de fonctions, des affectations, des boucles, des instructions conditionnelles, qui utilisent des expressions pouvant être des résultats d'appels de fonctions.

On distingue deux types de boucles :

- les boucles conditionnelles :

```
while condition:
    instructions
```

- les boucles non conditionnelles :

```
for elt in sequence:
    instructions
```

Avec une boucle non conditionnelle, le bloc d'instructions est répété n fois, n étant la longueur de la séquence (une liste, un tuple, une chaîne de caractères). La variable `elt` prend successivement la valeur de l'un des n éléments de la séquence. Cette variable `elt` peut être utilisée dans le bloc d'instructions ou pas.

Exemple :

```
for i in range(n):
    instructions
```

Avec le code qui suit, le bloc `instructions` est exécuté de la même manière :

```
i = 0
while i < n:
    instructions
    i = i + 1
```

Une variable `i` est créée. Cette variable n'est pas détruite après l'exécution de la boucle. Avec la boucle `for`, elle prend successivement les valeurs 0, 1, ..., `n`−1, avec la boucle `while` elle prend les valeurs 0, 1, ..., `n`.

■ Outils

■ Compteurs

Lorsqu'on utilise une boucle `while` on peut souhaiter compter le nombre de passages dans la boucle. De manière générale, on peut avoir besoin de compter le nombre d'apparitions d'un certain fait. On utilise alors une variable que l'on peut appeler *compteur*. Cette variable est initialisée à 0 et peut être incrémentée d'une unité à chaque passage dans la boucle.

Exemple avec une boucle conditionnelle sans test

Le paramètre `n` est un entier naturel. On compte le nombre de divisions euclidiennes successives de `n` par 2, jusqu'à arriver à un quotient nul. On obtient donc le nombre de chiffres dans l'écriture binaire de `n` si `n` est non nul.

```
def taille(n):
    cpt = 0
    while n > 0:
        cpt = cpt + 1
        n = n // 2
    return cpt
```

Avec une boucle inconditionnelle et un test

Le paramètre `n` est un entier naturel non nul. Le compteur est incrémenté quand `n` est divisible par `d`. On compte donc le nombre de diviseurs de `n` qui est le résultat renvoyé par la fonction.

```
def diviseurs(n):
    cpt = 0
    for d in range(1, n + 1):
        if n % d == 0:
            cpt = cpt + 1
    return cpt
```

■ Accumulateurs

Un accumulateur est semblable à un compteur mais il peut être incrémenté d'une valeur différente de 1 ou décrémenté.

```
def somme_pairs(liste):
    acc = 0
    for x in liste:
        if x % 2 == 0:
            acc = acc + x
    return acc
```

La boucle contient un test. Nous supposons que `liste` est une liste de nombres. La fonction renvoie la somme des nombres pairs contenus dans la liste.

L'exemple suivant qui concerne le calcul d'une moyenne est sans test.

```
def moyenne(liste):
    acc = 0
    for x in liste:
        acc = acc + x
    return acc / len(liste)
```

La liste est supposée non vide. Si n est sa longueur, nous disons que *le coût est linéaire en n*, car nous opérons n additions et n affectations effectuées dans la boucle. Nous supposons, et c'est le cas, que l'obtention de la longueur par `len(liste)` a un coût constant (une seule opération).

■ Recherche de valeurs

Définition : une *occurrence* est l'apparition d'un fait, par exemple la présence d'un mot dans un texte. Lorsqu'on cherche dans un conteneur une occurrence d'un élément, on cherche si une place est occupée par cet élément dans le conteneur. Une place est représentée par son indice.

■ Recherche d'une occurrence

Il s'agit de rechercher de manière séquentielle la présence d'une valeur dans un tableau. Cela signifie que la valeur cherchée est successivement comparées à toutes les valeurs du tableau. Cette méthode est aussi appelée *méthode par balayage* ou *recherche linéaire* (en anglais, *linear search*). Un tableau peut être ici une liste, un p-uplet ou une chaîne de caractères. On cherche donc une valeur précise dans une liste ou un p-uplet ou un caractère précis dans une chaîne.

L'algorithme s'arrête dès que l'élément est trouvé ou si la fin du tableau est atteinte.

Un algorithme de recherche d'un élément `x` dans un tableau `t` de longueur `n` utilise, de manière générale, une boucle conditionnelle :

```
i = 0
tant que i < n et x différent de t[i]
    i = i + 1
fin tant que
si i < n
    renvoyer i
```

Cet algorithme est l'occasion d'aborder la notion de *coût*, notion qui est précisée au chapitre 10. Disons que *le coût en temps d'un algorithme est déterminé par le nombre d'opérations élémentaires, (affectations, comparaisons, opérations arithmétiques simples), exécutées par l'algorithme.*

Nous commençons par compter le nombre maximum de comparaisons effectuées.

Si l'élément recherché n'est pas dans le tableau, il est nécessaire de parcourir tout le tableau, donc d'effectuer n itérations, pour examiner chaque élément du tableau avec $2n$ comparaisons (comparaisons entre `i` et `n`, et entre `x` et `t[i]`. Nous avons alors pour la boucle un total de n affectations et n additions, donc de $4n$ opérations. Nous disons que *le coût est linéaire en n.*

Lorsqu'on parcourt une liste ou une chaîne, on parle de parcours séquentiel. Dans la notion de séquence nous avons la notion d'ordre. Les éléments ont chacun une place numérotée par un indice.

Nous pouvons envisager plusieurs variantes qu'il faut maîtriser : soit on souhaite simplement chercher si une valeur est présente, soit on cherche un indice d'occurrence, soit on cherche tous les indices d'occurrence. La suite propose quelques exemples.

- **Recherche de la présence d'une valeur avec une boucle `while`**

```
def recherche1(tab, val):
    presence = False
    i = 0
    while i < len(tab) and not presence:
        if tab[i] == val:
            presence = True
        i = i + 1
    return presence
```

Une deuxième forme :

```
def recherche2(tab, val):
    i = 0
    while i < len(tab)-1 and tab[i] != val:
        i = i + 1
    return tab[i] == val
```

Avec une boucle `for` et une sortie de boucle anticipée :

```
def recherche3(tab, val):
    for i in range(len(tab)):
        if tab[i] == val:
            return True
    return False
```

Une fonction semblable à la précédente mais qui n'utilise pas les indices :

```
def recherche4(tab, val):
    for elt in tab:
        if elt == val:
            return True
    return False
```

Avec la fonction qui suit, c'est Python qui fait tout le travail.

```
def recherche5(tab, val):
    return val in tab
```

- **Recherche de la première occurrence**
  Avec une boucle `while` :

```
def recherche6(tab, val):
    indice = 0
    while indice < len(tab) and tab[indice] != val:
        indice = indice + 1
    return indice  # renvoie len(tab) si échec
```

Avec une boucle `for` et une sortie de boucle anticipée :

```
def recherche7(tab, val):
    for indice in range(len(tab)):
        if tab[indice] == val:
            return indice
    return len(tab)  # ou indice + 1
```

- **Recherche de la dernière occurrence**
  Pour la recherche de la dernière occurrence, on parcourt tout le tableau avec une boucle `for` :

```
def recherche8(tab, val):
    indice = len(tab)
    for i in range(len(tab)):
        if tab[i] == val:
            indice = i
    return indice
```

On pourrait aussi chercher la première occurrence à partir de la fin. Avec une petite modification de la fonction `recherche7`, on obtient la fonction `recherche9` ci-dessous :

```
def recherche9(tab, val):
    for i in range(len(tab)):
        indice = len(tab) - 1 - i
        if tab[indice] == val:
            return indice
    return len(tab)
```

■ Recherche d'un extrémum

Nous disposons d'un ensemble de nombres dans lequel nous cherchons un extrémum. On peut chercher un maximum, un minimum, ou les deux.

- **Algorithme de recherche du maximum**

Si la liste est non vide, on suppose que le maximum est le premier élément, puis on parcourt la liste et chaque fois qu'on rencontre un élément plus grand que le maximum provisoire, on dit que c'est le nouveau maximum provisoire.

Nous procédons à la recherche du maximum à l'aide d'un parcours séquentiel dans une liste de nombres non vide.

```
def maximum(liste):
    maxi = liste[0]
    for i in range(1, len(liste)):
        if liste[i] > maxi:
            maxi = liste[i]
    return maxi
```

Évaluons le nombre de comparaisons et le nombre d'affectations effectuées afin d'obtenir le coût en temps de l'algorithme. Si n est la longueur de la liste, nous avons dans tous les cas $n-1$ comparaisons et au maximum, (nous disons *dans le cas le moins favorable* ou *dans le pire des cas*), n affectations, (une avant d'entrer dans la boucle), donc un total de $2n-1$ opérations. Nous disons que le coût ou la complexité de l'algorithme est linéaire en fonction de la taille de la liste.

Il est évident que la recherche d'un minimum dans une liste de nombres s'effectue de manière similaire. Il suffit de remplacer le signe `>` par le signe `<` dans la comparaison.

Un problème semblable est de chercher les deux plus grands éléments. On pourrait commencer par chercher le maximum avec $n-1$ comparaisons puis recommencer pour chercher le deuxième maximum avec $n-2$ comparaisons.

La méthode appliquée ci-dessous est différente, c'est celle qui est appliquée à la recherche d'un maximum : les deux premiers éléments constituent le couple cherché, puis on parcourt la liste pour remplacer éventuellement le plus petit de ces deux éléments. On suppose que la liste contient au moins deux éléments et que les éléments sont tous distincts.

```
def maxima2(liste):
    maxi1, maxi2 = liste[0], liste[1]
    if maxi1 < maxi2:
        maxi1, maxi2 = maxi2, maxi1
    for i in range(2, len(liste)):
        x = liste[i]
        if x < maxi1:
            if x > maxi2:
                maxi2 = x
        else:
            maxi1, maxi2 = x, maxi1
    return maxi1, maxi2
```

Le nombre `maxi1` est le maximum, `maxi2` est le second maximum. Suivant les cas, le nombre total de comparaisons, en comptant celle effectuée avant la boucle, est compris entre $n-1$ et $2n-3$. Si la première comparaison échoue à chaque fois, il n'y a que $n-1$ comparaisons, si elle réussie à chaque fois, il y en a $2n-3$.

Il est possible de diminuer le nombre de comparaisons pour cette recherche. En effet lorsqu'on a effectué $n-1$ comparaisons pour trouver le maximum, on est sûr que le second maximum fait partie des éléments qui ont été comparés au maximum. L'algorithme est nettement moins simple à écrire.

Pour la recherche conjointe du maximum et du minimum, le nombre de comparaisons peut aussi être réduit. Une méthode est proposée au chapitre 2.

■ Comptage

Un nouveau type d'objets nommés *dictionnaires* est utilisé. C'est le type **dict**.

Dans un objet de type list, les éléments sont ordonnés. Ils sont indexés par une suite d'entiers 0, 1, 2, etc. Avec le type dict, les éléments sont indexés par des clés qui peuvent être de n'importe quel type *non mutable*, par exemple int, float, tuple. L'utilisation d'un dictionnaire est décrite à la fin du livre en annexe.

Si les éléments à compter sont des entiers naturels, on peut utiliser une liste. On considère alors que les indices de la liste représentent ces entiers et les valeurs de la liste leur nombre. Soit le maximum est donné, soit on le calcule avec la fonction `maximum`. Le langage Python propose aussi une fonction `max`.

```
def compte(entiers):
    m = max(entiers)  # ou maximum(entiers)
    cpts = [0 for i in range(m+1)]
    for n in entiers:
        cpts[n] = cpts[n] + 1
    return cpts
```

Une condition pour que la méthode soit efficace est que la dispersion des éléments ne soit pas trop importante afin de ne pas avoir une liste trop longue dont les éléments sont en majorité nuls et ne servent à rien. Si c'est le cas, une solution est d'utiliser un dictionnaire. Les valeurs des éléments du tableau sont les clés du dictionnaire. Cette méthode permet en plus de traiter les cas où les éléments à compter ne sont pas des entiers naturels.

```
def compte(liste):
    d = {}
    for elt in liste:
        if elt in d:
            d[elt] = d[elt] + 1
        else:
            d[elt] = 1
    return d
```

Vrai/Faux

	Vrai	Faux
1. Avec `for i in range(1, 5)`, la variable `i` prend successivement cinq valeurs.	☐	☐
2. En Python, une boucle `for` unique se termine toujours après un nombre fini d'étapes.	☐	☐
3. Avec les instructions `i=1` puis `while i<5: i=i+1`, la variable `i` prend successivement cinq valeurs.	☐	☐
4. Après les instructions `lst = []` puis `lst.append([1, 2])`, la liste `lst` a pour valeur `[1, 2]`.	☐	☐
5. Après les instructions `lst = []` puis `for i in range(1, 5): lst.append(i)`, la liste `lst` a pour valeur `[1, 2, 3, 4]`.	☐	☐
6. Avec une liste de nombres, le calcul de la moyenne des éléments a un coût linéaire.	☐	☐
7. La recherche d'un minimum dans une liste a un coût linéaire.	☐	☐
8. Soit `d` un dictionnaire. Si la clé `k` n'est pas dans ce dictionnaire, l'instruction `d[k]=5` provoque une erreur.	☐	☐
9. Si `d` est un dictionnaire, l'instruction `d[5]=[10,15]` provoque une erreur.	☐	☐
10. Si `d` est un dictionnaire, l'instruction `d[[10,15]]=5` provoque une erreur.	☐	☐

Énoncé des exercices

Exercice 1.1 : Quelles sont les valeurs finales de `i` et `n` après le code qui suit ?

```
n = 1
for i in range(2, 5):
    n = n * i
```

Exercice 1.2 : Écrire une fonction `longueur` qui prend en paramètre une chaîne de caractères ou une liste et renvoie la longueur de la chaîne ou de la liste. Il est interdit d'utiliser la fonction `len`.

Exercice 1.3 : Écrire une fonction qui prend en argument un n-uplet composé d'entiers et renvoie un couple de deux listes : la première contient les nombres pairs et la seconde les nombres impairs.

Exercice 1.4 : Écrire une fonction `produit` qui prend en paramètres un entier naturel `n` et une liste de nombres `nombres` et renvoie une nouvelle liste obtenue en multipliant chaque élément de la liste `nombres` par `n`.

Exercice 1.5 : Calcul d'écarts

1. Écrire une fonction `distance` qui prend en argument une liste de nombres, calcule les écarts en valeur absolue entre chaque nombre de la liste et la moyenne de ces nombres, et renvoie la somme des écarts. (En Python la fonction `abs` renvoie la valeur absolue du nombre donné en paramètre).

2. Quel est le coût de ce programme ?

Exercice 1.6 : On considère la liste `[5, 8, 12, 13, 17, 2]` et le code qui suit :

```
k = liste[5]
j = 4
while j > 0 and liste[j] > k:
    j = j - 1
liste[j], liste[5] = liste[5], liste[j]
```

Obtient-on une erreur et sinon quel est l'état final de la liste ?

Exercice 1.7 : Écrire une fonction `indice_maximum` qui prend en paramètre une liste de nombres et renvoie le maximum de ces nombres avec son indice dans la liste. Si plusieurs éléments sont égaux au maximum, la fonction renvoie l'indice le plus petit parmi les indices de ces éléments.

Exercice 1.8 : Écrire une fonction `indices_maximum` qui prend en paramètre une liste de nombres non vide et renvoie le maximum de ces nombres avec la liste des indices de tous les éléments égaux au maximum.

Exercice 1.9 : Voici une fonction `mystere` :

```
def mystere(liste1, liste2):
    liste = []
    i, j = 0, 0
```

```
    while i < len(liste1) and j < len(liste2):
        if liste1[i] < liste2[j]:
            liste.append(liste1[i])
            i = i + 1
        else:
            liste.append(liste2[j])
            j = j + 1
    return liste
```

On appelle cette fonction avec l'instruction `mystere([2, 5, 6, 8], [1, 4, 7, 8, 9])`. Quel est le résultat renvoyé ? Trouver le résultat avant de tester sur machine !

Exercice 1.10 : Écrire une fonction qui prend en argument une chaîne de caractères, qui peut représenter un mot ou un texte, et qui renvoie le nombre de 'e' contenus dans la chaîne.

Exercice 1.11 : Écrire une fonction `compte` qui prend en paramètre un mot et renvoie un dictionnaire qui pour chaque caractère du mot associe le nombre d'occurrences de ce caractère.

Un mot est une variable de type str qui ne contient que des caractères alphanumériques.

Exercice 1.12 : Écrire une fonction `stat` qui prend en paramètre un texte (type str) et renvoie un dictionnaire dont les clés sont les différentes lettres du texte et les valeurs le nombre d'occurrences de chaque lettre. On suppose le texte écrit en lettres capitales non accentuées. On ne comptabilise pas les autres caractères (ponctuation, espaces).

Exercice 1.13 : Écrire une fonction qui prend en paramètre un mot (type str) et qui renvoie un dictionnaire ayant pour clés les caractères du mot et pour valeurs les distances (les plus courtes si des caractères apparaissent plusieurs fois) entre chaque caractère du mot et le dernier caractère.

Exercice 1.14 : La question est de mesurer l'intérêt d'une représentation par un dictionnaire plutôt que par une liste de listes. Nous allons donc comparer les temps de recherche d'un élément.

1. Construire une liste dont les éléments sont de la forme `[i, i]` pour `i` allant de 0 à $10^6 - 1$. Mélanger cette liste avec la fonction `shuffle` du module `random`. Il suffit d'importer la fonction et d'écrire `shuffle(liste)`. Créer alors le dictionnaire correspondant qui contient les couples `clé:valeur` de la forme `i:i` à l'aide de la fonction `dict`.

2. Écrire une fonction `recherche1` qui prend en paramètres une liste de listes et une variable `k` et renvoie le deuxième élément de la sous-liste dont le premier élément a la valeur de `k`.

3. Écrire une fonction `recherche2` qui prend en paramètres un dictionnaire et une variable `k` et renvoie la valeur correspondant à la clé `k`.

4. Tester les fonctions de recherche sur la liste et le dictionnaire en utilisant pour le paramètre `k` les valeurs de 0 à 49.

Pour les mesures, utiliser la fonction `time` du module `time`. Le mode d'utilisation est :

```
from time import time
top = time()
# ici le programme à exécuter
print(time() - top)
```

Indications

Ex. 1.2

Il est nécessaire de parcourir le conteneur, chaîne ou liste.

Ex. 1.8

Il faut commencer par trouver le maximum.

Ex. 1.11

S'inspirer du comptage des éléments d'une liste présenté dans le cours.

Ex. 1.12

Parcourir le texte avec une boucle. Vérifier que le caractère lu n'est pas une espace ou un signe de ponctuation.

Corrigé du vrai/faux

1	2	3	4	5	6	7	8	9	10
F	F	V	F	V	V	V	F	F	V

Quelques explications

1. La variable `i` prend successivement les valeurs de 1 à 4.

2. Exemple : `liste=[0]`, puis `for k in liste: liste.append(k)`. Cette boucle ne termine pas.

3. La variable `i` prend successivement les valeurs de 1 à 5.

4. La liste `lst` a pour valeur finale `[[1, 2]]`.

8. Si elle n'existe pas, la clé est créée.

10. Une clé ne peut pas être une liste (qui est un objet mutable). Elle peut être un n-uplet. L'instruction `d[(10,15)]=5` serait correcte.

❑ **Erreurs classiques et conseils.—**

- Il faut être très vigilant concernant les indices des éléments d'une liste ou d'une chaîne de caractères. Le premier élément a pour indice 0, le dernier a pour indice `n-1` si `n` est la longueur de la liste ou de la chaîne. Avec l'instruction `for i in range(n)`, on parcourt exactement les indices des éléments d'une liste ou d'une chaîne de longueur `n`.
- Si `d` est un dictionnaire qui ne contient pas la clé `c`, l'instruction `d[c]=val` crée la clé `c` et lui associe la valeur `val`.
- Attention, les clés d'un dictionnaire ne peuvent pas être du type list.

Corrigé des exercices

Exercice 1.1

La variable `i` prend successivement les valeurs 2, 3 et 4. Au premier passage dans la boucle `i` vaut 2 donc `n` prend la valeur 2. Au deuxième passage `i` vaut 3 et `n` prend la valeur 6. Au troisième passage `i` vaut 4 et `n` prend la valeur 24. Les valeurs finales de `i` et `n` sont respectivement 4 et 24.

Exercice 1.2

Une solution est de parcourir le conteneur en comptant les éléments un par un.

```
def longueur(conteneur):
    """conteneur est du type str ou list
    renvoie le nombre d'éléments du conteneur"""
    cpt = 0
    for elt in conteneur:
        cpt = cpt + 1
    return cpt
```

Exercice 1.3

```
def pair_impair(uplet):
    """uplet est un tuple contenant des entiers,
    renvoie la liste des entiers pairs et celle des entiers impairs"""
    liste_pair = []
    liste_impair = []
    for n in uplet:
        if n % 2 == 0:
            liste_pair.append(n)
        else:
            liste_impair.append(n)
    return liste_pair, liste_impair
```

Exercice 1.4

```
def produit(n, nombres):
    """n est de type int et nombres de type list
    multiplie chaque élément de nombres par n et renvoie la nouvelle liste"""
    liste = []
    for x in nombres:
        liste.append(x * n)
    return liste
```

```
# ou plus simplement en compréhension
def produit(n, nombres):
    return [n * x for x in nombres]
```

Exercice 1.5

1. Il faut commencer par calculer la moyenne avec une boucle sur les éléments de la liste. Ensuite à l'aide d'une seconde boucle, on calcule les écarts dont on fait la somme.

```
def distance(liste):
    somme = 0
    for x in liste:
        somme = somme + x
    m = somme / len(liste)
    ecarts = 0
    for x in liste:
        ecarts = ecarts + abs(x - m)
    return ecarts
```

2. Le coût de la deuxième boucle est linéaire en la taille de la liste.
Donc le coût du programme est similaire à celui du calcul de la moyenne.

Exercice 1.6

La variable `k` prend la valeur 2. La valeur finale de `j` est 0 et `liste[0]` est le premier élément de la liste, soit 5, qui est alors échangé avec 2. On obtient la liste `[2, 8, 12, 13, 17, 5]`.

Exercice 1.7

```
def indice_maximum(liste):
    if liste == []:
        return None
    else:
        indice = 0
        maxi = liste[0]
        for i in range(len(liste)):
            if liste[i] > maxi:
                indice = i
                maxi = liste[i]
    return maxi, indice
```

Remarque : si on remplace la ligne `if liste[i] > maxi` par la ligne `if liste[i] >= maxi`, on obtient l'indice le plus grand.

Exercice 1.8

On commence par chercher le maximum puis on construit la liste des indices des éléments égaux au maximum.

```
def indices_maximum(liste):
    maxi = liste[0]
    for x in liste:
        if x > maxi:
            maxi = x
    indices = []
    for i in range(len(liste)):
        if liste[i] == maxi:
            indices.append(i)
    return maxi, indices
```

Exercice 1.9

La fonction `mystere` fusionne les deux listes passées en paramètres en respectant l'ordre, tant que tous les éléments d'une des deux listes n'ont pas été choisis.

L'appel `mystere([2, 5, 6, 8], [1, 4, 7, 8, 9])` renvoie donc la liste `[1,2,4,5,6,7,8,8]`.

Exercice 1.10

```
def nombre_de_e(chaine):
    cpt = 0
    for car in chaine:
        if car == 'e':
            cpt = cpt + 1
    return cpt
```

Nous testons cette fonction dans l'interpréteur :

```
>>> nombre_de_e("Le soleil brille")
3
>>> nombre_de_e("Combien de e dans cette chaîne de caractères ?")
8
```

Attention, cette fonction ne compte que les 'e' pas les 'é', les 'è', les 'ê'.

Exercice 1.11

```
def compte(mot):
    lettres = {}
    for c in mot:
        if c in lettres:
            lettres[c] = lettres[c] + 1
        else:
```

```
            lettres[c] = 1
    return lettres
```

Une instruction comme `lettres[c] = lettres[c] + 1` provoquerait une erreur si la clé `c` n'existe pas, ce qui serait le cas dès la première lettre. On teste donc pour chaque lettre l'appartenance aux clés du dictionnaire.

Exercice 1.12

```
def stat(texte):
    lettres = {}
    for c in texte:
        if c not in " ,;:!?.":
            if c in lettres:
                lettres[c] = lettres[c] + 1
            else:
                lettres[c] = 1
    return lettres
```

Testons sur un exemple :

```
>>> stat("MON PROGRAMME FONCTIONNE, JE SUIS CONTENT !")
{'M': 3, 'O': 5, 'N': 6, 'P': 1, 'R': 2, 'G': 1, 'A': 1, 'E': 4, 'F': 1,
'C': 2, 'T': 3, 'I': 2, 'J': 1, 'S': 2, 'U': 1}
```

On peut trier le résultat avec différentes instructions.
Testons sur un exemple :

```
>>> d = stat("MON PROGRAMME FONCTIONNE, JE SUIS CONTENT !")
>>> sorted(d)
['A', 'C', 'E', 'F', 'G', 'I', 'J', 'M', 'N', 'O', 'P', 'R', 'S', 'T', 'U']
>>> sorted(d.items())
[('A', 1), ('C', 2), ('E', 4), ('F', 1), ('G', 1), ('I', 2), ('J', 1),
('M', 3), ('N', 6), ('O', 5), ('P', 1), ('R', 2), ('S', 2), ('T', 3), ('U', 1)]
```

Exercice 1.13

```
def distances(mot):
    n = len(mot)
    d = {}
    for i in range(n):
        d[mot[i]] = n - 1 - i
    return d
```

Exercice 1.14

1. Création des objets :

```
liste = [[i, i] for i in range(10**6)]
from random import shuffle
shuffle(liste)
dico = dict(liste)
```

2. Fonction `recherche1` :

```
def recherche1(liste, k):
    for elt in liste:
        if elt[0] == k:
            return elt[1]
```

3. Pour la fonction `recherche2`, si on écrit une fonction similaire à `recherche1` comme ci-dessous, on ne profite pas des propriétés de performance permises par l'implémentation des dictionnaires.

```
def recherche2(dico, k):
    for cle in dico:
        if cle == k:
            return dico[k]
```

Il vaut mieux écrire une fonction comme celle-ci :

```
def recherche3(dico, k):
    if k in dico:
        return dico[k]
```

4. Tests :

```
from time import time

st = time()
for i in range(50):
    recherche1(liste, i)
print(time() - st)  # affiche 7.461 (secondes)

st = time()
for i in range(50):
    recherche2(dico, i)
```

Corrigé

```
print(time() - st)  # affiche 3.297 (secondes)

st = time()
for i in range(50):
    recherche3(dico, i)
print(time()-st)  # affiche 0.0 (secondes)

st = time()
for i in range(50000):
    recherche3(dico, i)
print(time() - st)  # affiche 0.031 (secondes)
```

Les résultats sont parlants ! La fonction `recherche2` semble environ deux fois plus rapide que la fonction `recherche1` alors que la fonction `recherche3` est environ deux cent mille fois plus rapide. Nous avons dans le dernier test mille fois plus de recherches et le temps mesuré est environ deux cent fois plus petit que celui obtenu pour la fonction `recherche1`.

Chapitre 2

Boucles imbriquées

UN SCIENTIFIQUE



Euclide (env. 330 av. J.-C. - env. 275 av. J.-C.) enseignait à Alexandrie au début du III^{e} siècle avant notre ère. Son ouvrage fondamental intitulé *Les Éléments*, regroupe presque toutes les connaissances mathématiques de l'époque. Son apport principal concerne la géométrie ; cependant, dans son VII^{e} livre qui traite de l'arithmétique, il introduit la notion de nombre premier, de diviseurs et de nombres premiers entre eux et sans doute le premier algorithme de l'histoire.

Un peu d'histoire

L'algorithme d'Euclide est certainement le premier à comporter une boucle. Il consiste à déterminer si deux nombres entiers donnés sont premiers entre eux, c'est-à-dire que 1 est leur seul diviseur commun. Si on note $a > b$ ces deux nombres on effectue la division euclidienne $a = bq + r$ avec $r < b$. On réitère le procédé avec b et r jusqu'à l'obtention d'un reste nul. Si le dernier reste non nul est 1, les deux nombres sont premiers entre eux ; sinon, le dernier reste non nul est le plus grand diviseur commun.
Tant que les calculs se faisaient à la main, le nombre de boucles, éventuellement imbriquées était restreint. Les potentialités de l'outil informatique ont permis de mettre en valeur des méthodes jusque-là inutilisables. Par exemple, la méthode du pivot de Gauss est de nos jours privilégiée pour la résolution de système linéaire alors que des méthodes très théoriques étaient préférées autrefois.

Certains algorithmes nécessitent d'effectuer une deuxième boucle à chaque passage dans une boucle. On parle de boucles imbriquées avec une première boucle *externe* et une seconde *interne*.

OBJECTIFS

- Comprendre le principe des boucles imbriquées :
  - reconnaître leur utilité dans certaines recherches sur des structures séquentielles ;
  - être capable de concevoir un algorithme utilisant des boucles imbriquées ;
  - découvrir leur rôle dans un algorithme de tri.
- Savoir analyser un algorithme présentant des boucles imbriquées :
  - comprendre le calcul de la complexité dans différents cas ;
  - savoir différencier des coûts quadratiques des coûts linéaires ;
  - appréhender la notion de correction d'un algorithme.
- Savoir effectuer une recherche dans un texte :
  - maîtriser la programmation d'une recherche naïve ;
  - découvrir les principes d'algorithmes plus performants.

Cours

Les programmes rencontrés dans le chapitre 1 contiennent des boucles simples. Mais il est possible de placer plusieurs boucles `while` ou `for` à l'intérieur d'une boucle `while` ou `for`. On parle alors de boucles imbriquées.

Par exemple :

```
for i in range(4):
    for j in range(3):
        print(i + j)
```

Les valeurs successives des variables `i` et `j` sont :

`i = 0` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 1` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 2` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 3` et `j = 0`, puis `j = 1`, puis `j = 2`.

Pour chacune des quatre valeurs de `i`, (0, 1, 2, 3), `j` prend trois valeurs, (0, 1, 2), et nous aurons donc douze affichages avec la fonction `print`.

Voici deux exemples pour créer des listes emboîtées avec une définition en compréhension :

```
>>> liste = [[x, y] for x in range(3) for y in range(3)]
>>> liste
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
>>> liste = [[x, y] for y in range(3) for x in range(3) if x != y]
>>> liste
[[1, 0], [2, 0], [0, 1], [2, 1], [0, 2], [1, 2]]
```


Recherche de valeurs

Recherche simultanée d'un maximum et d'un minimum

Nous pouvons envisager de chercher conjointement un minimum et un maximum dans une liste de nombres. Cette recherche est basée sur le parcours séquentiel de la liste, comme au chapitre 1.

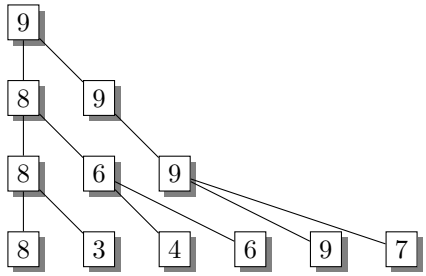
Exemple de programme calqué sur le programme `maximum` écrit dans le chapitre 1 :

```
def minimaxi(liste):  # la liste est supposée non vide
    mini = liste[0]
    maxi = liste[0]
    for i in range(1, len(liste)):
        if liste[i] > maxi:
            maxi = liste[i]
```

```
        elif liste[i] < mini:
            mini = liste[i]
    return mini, maxi
```

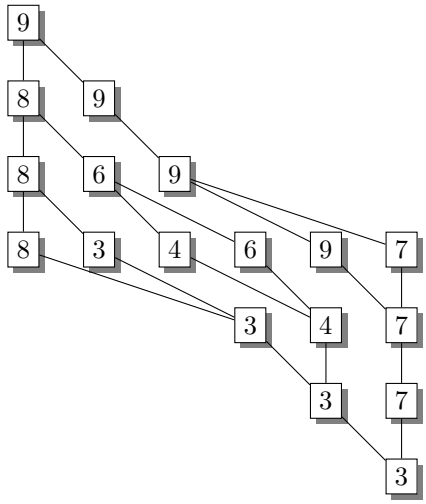
Dans le pire des cas, nous avons 2 comparaisons à chaque passage dans la boucle, donc $2(n-1)$ comparaisons à effectuer. Un même élément de la liste est comparé à `maxi` et à `mini`.

Utilisons une autre manière de procéder pour trouver le maximum. On compare deux éléments consécutifs et on garde le plus grand des deux à chaque fois. On garde aussi le dernier élément si le nombre d'éléments est impair. Puis on recommence avec la liste des plus grands, et ainsi de suite.



Nous avons cinq comparaisons pour une liste de six éléments, donc aucune amélioration. Mais nous pouvons utiliser les trois comparaisons du premier niveau pour obtenir le minimum.

Nous économisons ainsi les comparaisons du premier niveau.



Le nombre total de comparaisons est $3 + 2(1 + 1) = 7$ au lieu de 10.

Un exemple de programme, en supposant la liste non vide :

```
def maximini(liste):  # la liste est supposée non vide
    n = len(liste)
    k = n // 2
```

```
    liste_pg, liste_pp = k * [0], k * [0]
    for i in range(k):
        if liste[2*i] > liste[2*i+1]:
            liste_pg[i], liste_pp[i] = liste[2*i], liste[2*i+1]
        else:
            liste_pg[i], liste_pp[i] = liste[2*i+1], liste[2*i]
    if n % 2 == 1:  # gestion d'un nombre impair d'éléments
        liste_pg.append(liste[n-1])
        liste_pp.append(liste[n-1])
    n = len(liste_pg)
    while n >= 2:
        k = n // 2
        # les nouveaux plus grands (npg) et les nouveaux plus petits (npp)
        npg, npp = k * [0], k * [0]
        for i in range(k):
            npg[i] = max(liste_pg[2*i], liste_pg[2*i+1])
            npp[i] = min(liste_pp[2*i], liste_pp[2*i+1])
        if n % 2 == 1:
            npg.append(liste_pg[n-1])
            npp.append(liste_pp[n-1])
        liste_pg, liste_pp = list(npg), list(npp)
        n = len(liste_pg)
    return liste_pg[0], liste_pp[0]
```

Une recherche des deux plus proches voisins d'un point peut s'effectuer de manière linéaire. On stocke les deux premiers points. On parcourt la liste et on effectue deux comparaisons avec chaque élément. La recherche des deux points les plus proches et un problème moins simple.

■ Recherche des deux points les plus proches

On cherche une paire de points dont la distance est minimale dans un ensemble fini de points.

Algorithme naïf : on teste toutes les paires. Si l'ensemble contient n éléments, il y a $n(n-1)/2$ paires à tester. On peut utiliser des nombres avec pour distance entre x et y, `abs(x-y)`, ou se placer dans le plan avec la distance euclidienne.

```
def plus_proches(liste, dist):
    n = len(liste)
    paire = [0, 1]
    d = dist(liste[0], liste[1]) # distance à définir
    for i in range(n-1):
        for j in range(i+1, n):
            dij = dist(liste[i], liste[j])
            if dij < d:
                d = dij
                paire = [i, j]
    return paire
```

Pour chaque valeur de `i`, `j` prend les valeur de `i+1` à `n-1`. L'algorithme consiste à calculer toutes les distances entre deux points sans répétitions. Nous avons donc $n(n-1)/2$ distances à calculer. Nous disons que la complexité de l'algorithme en fonction de n est de l'ordre de n^2.

■ Un algorithme de tri

■ Introduction

Trier des données, c'est les ranger suivant un ordre défini au préalable. Par exemple avec des données numériques, nous pouvons trier ces données en utilisant l'ordre défini en mathématiques : a est avant b si $b-a$ est positif ($a \leq b$ si $b-a \geq 0$). Si nous trions l'ensemble de nombres $\{3, 8, 5, 2\}$, nous obtenons l'ensemble $\{2, 3, 5, 8\}$ et nous disons que les nombres sont rangés suivant l'ordre croissant. Si nous les rangeons dans l'ordre inverse, nous parlons d'ordre décroissant.

■ Le tri à bulles

L'algorithme du tri à bulles consiste à parcourir une liste plusieurs fois jusqu'à ce qu'elle soit triée, en comparant à chaque parcours les éléments consécutifs et en procédant à leur échange s'ils sont mal triés. Les étapes sont :

- on parcourt la liste et si deux éléments consécutifs sont rangés dans le désordre, on les échange ;
- si à la fin du parcours au moins un échange a eu lieu, on recommence l'opération ;
- sinon, la liste est triée, on arrête.

Les éléments les plus grands remontent ainsi dans la liste comme des bulles d'air qui remontent à la surface d'un liquide. Commençons par un exemple avec la liste : `[12, 17, 14, 11, 8]`. Les éléments permutés sont en gras.

Parcours 1 : [12, **14**, **17**, 11, 8], puis [12, 14, **11**, **17**, 8], puis [12, 14, 11, **8**, **17**].

Parcours 2 : [12, **11**, **14**, 8, 17], puis [12, 11, **8**, **14**, 17].

Parcours 3 : [**11**, **12**, 8, 14, 17], puis [11, **8**, **12**, 14, 17].

Parcours 4 : [**8**, **11**, 12, 14, 17].

Parcours 5 : [8, 11, 12, 14, 17].

Lors du parcours 5, aucune permutation n'est effectuée. La liste est donc triée.

Si une liste de n éléments est déjà triée, il y a un seul parcours de la liste. Nous en déduisons qu'il y a $n-1$ comparaisons. C'est le cas le plus favorable.

On considère la liste : $[n, n-1, \ldots, 3, 2, 1]$.

La valeur n se déplace après chaque comparaison jusqu'à la fin de la liste. Donc le nombre de permutations nécessaires pour amener n à la position correcte est $n-1$.

De manière similaire, pour amener $n-1$ à la bonne place il faut $n-2$ permutations et pour amener $n-2$ à la bonne place il en faut $n-3$.

Le nombre total de permutations nécessaires pour trier une liste de n éléments rangés initialement en ordre décroissant est donc $(n-1)+(n-2)+\ldots+1 = \dfrac{n(n-1)}{2}$. Nous disons que le coût de l'algorithme en fonction de n, ou sa complexité, est quadratique.

Programme : il n'est pas nécessaire de parcourir la liste jusqu'à la fin à chaque parcours. En effet, après le premier parcours, l'élément le plus grand est placé à la dernière place. Après le second parcours, les deux plus grands éléments sont rangés à leur place définitive. Et ainsi de suite.