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

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.

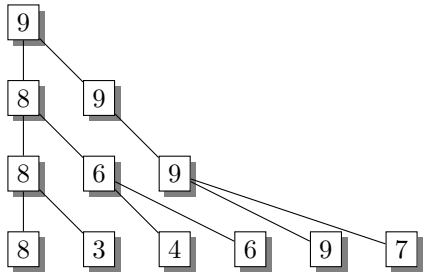
Exemple de programme calqué sur le programme `maximum` écrit dans le chapitre 1 :

```
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
```

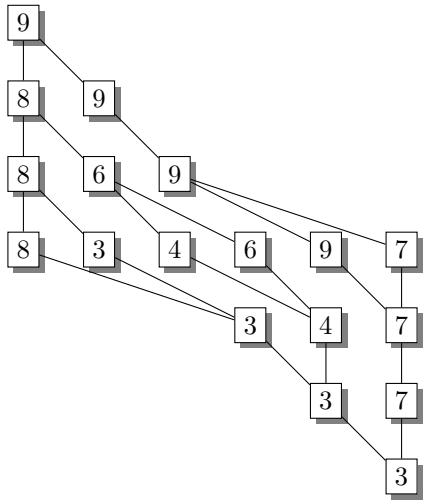
Dans le pire des cas, nous avons 2 comparaisons à chaque passage dans la boucle, donc $2(n-1)$ comparaisons à effectuer. Un même élément de la liste est comparé à `maxi` et à `mini`.

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.

```
def tri_bulles(liste):
    for j in range(1, len(liste)):
        trier = True
        for i in range(len(liste)-j):
            if liste[i] > liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                trier = False
        if trier:
            break
```

La fonction ne renvoie. On dit que la liste passée en paramètre est triée en place.

On pourrait remplacer l'instruction `break` par l'instruction `return True` ou simplement `return` qui ne renvoie rien (plus précisément, c'est la valeur `None` qui est renvoyée).

On peut aussi utiliser une boucle `while` :

```
def tri_bulles(liste):
    j = len(liste)-1
    while j > 0:
        tri = True
        for i in range(j):
            if liste[i] > liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                tri = False
        j = j - 1
        if tri:
            j = 0
```


Nous pouvons remarquer que dans le cas le plus défavorable, il y a autant de permutations de valeurs que de comparaisons. Des tris plus performants sont présentés au chapitre 8.

Notion de validité d'un algorithme

- **Terminaison**

  La question est de prouver que l'exécution du programme s'arrête après un nombre fini d'étapes. Il nous faut étudier les deux boucles. La boucle interne est une boucle `for` donc le nombre de passages est déterminé et fini. La boucle externe est une boucle `while`. Les valeurs prises par la variable `j` constituent une suite d'entiers strictement décroissante avec des valeurs allant de $n-1$ à 0, si n est la longueur de la liste. Il y a donc exactement $n-1$ passages dans la boucle `while`. L'expression égale à `j` s'appelle *un variant de boucle*.

- **Correction**

  Il s'agit de prouver que le résultat obtenu à la fin est bien celui attendu.

  On démontre pour cela qu'une propriété est vraie après chaque passage dans la boucle, propriété qu'on appelle *un invariant de boucle* : « pour chaque valeur de `j`, la liste est une permutation de la liste initiale et la liste `liste[j:len(liste)]` est triée ».

■ Recherche textuelle

La question est de déterminer la présence ou l'absence d'un motif dans un texte. Le texte et le motif sont représentés en Python par des chaînes de caractères (type str). Ils sont donc composés de caractères qui peuvent être des lettres, des signes de ponctuations, différents symboles et types d'espace. Si le motif est constitué d'un unique caractère, il s'agit d'une simple recherche séquentielle traitée au chapitre 1. Dans ce cas, le coût de la recherche est linéaire en la longueur de la chaîne.

■ Recherche naïve

On parle d'algorithme naïf quand il s'agit de la mise en œuvre d'une idée simple, l'une des premières idées à laquelle on peut penser. Le principe est le suivant :

- on cherche la présence du premier caractère du motif dans le texte ;
- si on le trouve, on vérifie si les caractères suivants du motif coïncident avec ceux du texte.
- si tous les caractères coïncident, le motif est trouvé, sinon on reprend la recherche du premier caractère.

Cet algorithme est implémenté dans le programme suivant :

```
def recherche(texte, motif):
    n = len(texte)
    m = len(motif)
    for j in range(n-m+1):
        i = 0
        while i < m and texte[j+i] == motif[i]:
            i = i + 1
        if i == m:
            return j
```

À partir de l'indice `n-m-1`, ce n'est plus la peine de chercher car il ne reste plus assez de place pour caser le motif à la fin du texte.

indice j	...	...	...	k	k+1	k+2	k+3	...	...
texte				e	x	e	m		
				↕	↕	↕			
motif				e	x	e	r		
indice i				0	1	2	3		

Étude du coût

On note n la longueur du texte et m la longueur du motif. La boucle externe `for` est parcourue $(n - m + 1)$ fois. Dans le pire des cas, la boucle interne `while` est parcourue au plus m fois, pour tester chaque caractère du motif. Le nombre total de comparaisons entre caractères est donc majoré par $m(n - m + 1)$. En particulier, si $m = n/2$, la valeur de ce produit vaut $(n^2 + 2n)/4$.

■ Pré-traitement du motif

Considérons le motif `'abcd'` et un texte commençant par `'abce'`. Avec l'algorithme précédent, les quatre caractères du motif sont comparés aux quatre premiers caractères du texte avec un échec sur le dernier. Le motif est alors décalé d'une place et on peut prévoir que la correspondance entre le `'a'` du motif et le `'b'` du texte va échouer. Il en est de même au décalage suivant.

Les informaticiens Morris et Pratt ont eu séparément l'idée d'effectuer un pré-traitement du motif qui permet de déterminer à partir de quelle place la recherche doit se poursuivre et éviter ainsi de contrôler les mêmes caractères plusieurs fois. Boyer et Moore, informaticiens également, ont aussi eu cette idée d'un pré-traitement du motif. Mais leur approche a été différente. La différence fondamentale avec d'autres algorithmes est que *la comparaison entre le motif et une partie du texte se fait de droite à gauche en commençant par la fin du motif.*

Exemple de pré-traitement du motif

Dans une version publiée par l'informaticien Nigel Horspool en 1980, le pré-traitement du motif consiste à construire un tableau de distances. Pour chaque caractère du motif, c'est la distance entre la dernière occurrence du caractère dans le motif, exceptée la dernière place, et le dernier caractère du motif. Pour tous les autres caractères de l'alphabet, c'est la longueur du motif.

Nous utilisons pour cela un dictionnaire. Par exemple si le motif est le mot "exemple", alors le dictionnaire est {`"e": 4, "x": 5, "m": 3, "p": 2, "l": 1, ...`}. Les autres couples `clé: valeur` sont tous de la forme `"*": 7` où `"*"` représente un caractère quelconque ne figurant pas dans le motif. Pour l'alphabet, on peut utiliser les 256 caractères codés par l'ASCII étendu.

- On crée une fonction qui construit un dictionnaire. Les clés sont tous les caractères dont le code ASCII varie de 0 à 255. Les valeurs correspondant à ces clés sont toutes initialisées à la valeur de `m`, la longueur du motif.
- La fonction prend en paramètre une chaîne de caractères, le motif. Chaque clé qui correspond à un caractère de la chaîne, excepté le dernier, est associée à la valeur `m-1-i`, où `i` est l'indice du caractère dans la chaîne, le dernier s'il est présent à plusieurs endroits de la chaîne, sans tenir compte de la dernière place.
- Cette fonction renvoie le dictionnaire qui permet de déterminer le décalage à appliquer au motif lors de la recherche.

```
def distances(motif):
    m = len(motif)
    dic = {chr(i): m for i in range(256)}
    for i in range(m-1):
        dic[motif[i]] = m - 1 - i
    return dic
```

Cette fonction permet d'obtenir le tableau des décalages : `d = distances(motif)`.

Il reste à écrire la fonction de recherche proprement dite que l'on nomme `boyer_moore`.

- On définit une fonction `boyer_moore` qui prend en paramètres un texte et un motif (les deux du type str). La comparaison entre le motif et le texte s'effectue de droite à gauche à partir du dernier caractère du motif.
- Si le motif est trouvé, on poursuit la recherche en décalant le motif d'un cran vers la droite et on stocke dans une liste l'indice de réussite, marquant le début d'une présence.

- Si le motif n'est pas trouvé, on poursuit la recherche en décalant le motif vers la droite de manière à faire coïncider le caractère fautif du texte avec sa dernière occurrence dans le motif. S'il n'y en a pas, on décale de la longueur du motif.
- La fonction renvoie la liste des indices trouvés.

Ce programme est une version simplifiée de l'algorithme de Boyer et Moore.

Si X, soit `texte[s]`, est le caractère fautif du texte, et si ce caractère est présent dans le motif à gauche du caractère comparé et pas à droite exceptée l'extrémité, alors `d[texte[s]]` notée d, est la distance dans le motif entre X et l'extrémité droite. Donc en ajoutant cette distance d à `s`, le caractère X du texte se trouve à la distance d du premier caractère du texte qui sera comparé après le décalage avec l'extrémité droite du motif, donc sera en face du X du motif.

indice s					j-k		j			j-k+d	
texte					X	Z	Z				
					↕	↕	↕				
motif		X			Y	Z	Z				
indice i		m-d-1	...	...	m-k-1		m-1				

```
def boyer_moore(texte, motif):
    m = len(motif)
    n = len(texte)
    d = distances(motif)
    s = m - 1  # on commence face au dernier caractère du motif
    sol = []
    while s < n:
        i = m - 1
        while i >= 0 and motif[i] == texte[s]:
            s = s - 1
            i = i - 1
        if i < 0:
            sol.append(s+1)
        s = s + max(d[texte[s]], m-i) # le décalage
    return sol
```

Remarque : si on remplace la ligne effectuant le décalage `s = s + max(d[texte[s]], m-i)` par la ligne `s = s + m - i`, le motif est décalé à chaque fois d'une unité vers la droite. On retrouve l'algorithme naïf utilisé en parcourant le motif de droite à gauche.

Cette simplification de l'algorithme de Boyer et Moore a pour origine une autre méthode publiée par Horspool où le décalage n'est pas effectué par rapport au caractère fautif du texte mais toujours par rapport au caractère comparé le plus à droite du texte. Cette méthode est proposée en exercice.

Coût de l'algorithme de Boyer et Moore

On note n la longueur du texte et m la longueur du motif. Le calcul du coût est difficile. Cependant dans le meilleur des cas, il suffit de remarquer que si la dernière lettre du motif est absente du texte, alors on compare avec la dernière lettre du motif une seule lettre du texte sur m lettres. Le nombre total de comparaisons est donc de l'ordre de n/m. De manière générale le coût diminue quand la longueur du motif augmente.

On montre que dans le pire des cas le coût est de l'ordre de $n + m$.

Vrai/Faux

	Vrai	Faux
1. Si le coût d'un algorithme est quadratique, c'est qu'il y a au moins deux boucles imbriquées.	☐	☐
2. La recherche simultanée d'un maximum et d'un minimum a un coût quadratique.	☐	☐
3. La recherche d'une médiane dans une liste déjà triée a un coût linéaire.	☐	☐
4. L'utilisation du tri à bulles pour trier une liste de taille n nécessite $n(n-1)/2$ permutations dans le cas le moins favorable.	☐	☐
5. Tous les algorithmes qui résolvent le même problème ont la même complexité.	☐	☐
6. Avec deux boucles imbriquées, la complexité peut être linéaire.	☐	☐
7. Un algorithme quelconque de recherche d'un motif composé de deux caractères dans une liste de longueur n a une complexité quadratique en fonction de n.	☐	☐
8. Si un algorithme ne contient pas d'erreur de syntaxe, alors il est correct.	☐	☐
9. Si on trouve une propriété invariante, on peut affirmer que l'algorithme est correct.	☐	☐
10. On utilise un invariant pour prouver qu'une boucle `while` se termine.	☐	☐

Énoncé des exercices

Exercice 2.1 : Combien de fois la fonction `print` est-elle appelée dans le code suivant ?

```
for i in range(5):
    for j in range(i+1, 5):
        print(i + j)
```

Exercice 2.2 : Pour chacun des trois scripts, déterminer le nombre d'additions effectuées. Exprimer ce nombre en fonction de `n` pour les deux derniers.

1.

```
x = 0
for i in range(2):
    x = x + i
    for j in range(3):
        x = x + j
```

2.

```
x = 0
for i in range(2):
    x = x + i
    for j in range(n):
        x = x + j
```

3.

```
x = 0
for i in range(n):
    x = x + i
    for j in range(n):
        x = x + j
```

Exercice 2.3 : Après le code Python qui suit, quelles sont les valeurs finales de `x` et de `y` ?

```
x = 4
while x > 0:
    y = 0
    while y < x:
        y = y + 1
        x = x - 1
```

Exercice 2.4 : Voici la définition d'une fonction qui prend en paramètre une liste de nombres :

```
def remonte(liste):
    for i in range(len(liste)-1):
        if liste[i] > liste[i+1]:
            liste[i], liste[i+1] = liste[i+1], liste[i]
```

On appelle la fonction `remonte` avec une liste de longueur n.

Quelle affirmation est vraie ?

1. Dans la liste modifiée, le plus petit élément est placé au début.
2. Dans la liste modifiée, le plus grand élément est placé à la fin.
3. Le nombre de comparaisons effectuées est exactement égal à n.
4. Le nombre d'échanges effectués est strictement inférieur à $n - 1$.

Exercice 2.5 : On reprend la fonction `remonte` définie dans l'exercice précédent.

On définit une liste `nombres = [12, 5, 13, 8, 11, 6]`. Si on appelle la fonction `remonte` avec en paramètre la liste `nombres`, quel est l'état final de cette liste ?

Exercice 2.6* : Écrire une fonction `ordre` qui prend en argument une liste de mots et modifie la liste en ordonnant les mots en fonction du nombre de lettres. La fonction ne renvoie rien.

Tester la fonction avec la liste `["toto", "bonjour", "a", "oui", "non"]`.

Exercice 2.7 : L'objectif est de comparer les temps d'exécution du tri à bulles sur deux types de listes : une liste de nombres au hasard et une liste de nombres déjà triée. Utiliser le code du tri à bulles et pour mesurer le temps d'exécution, procéder comme dans l'exercice 14 du chapitre 1 en important la fonction `time` du module `time`.

1. Construire une liste de 5000 entiers pris au hasard entre 1 et 10000, bornes comprises, puis la liste des 100000 entiers de 0 à 99999, bornes comprises. Mesurer les temps d'exécution du programme du tri à bulles pour trier chacune de ces listes. Quel commentaire peut-on faire sur ces temps ?
2. Comparer avec le temps d'exécution de la méthode `sort` sur une liste de 100000 entiers, choisis de manière aléatoire entre 1 et 100000. La syntaxe est `liste.sort()`.

Exercice 2.8* : Trier des points

On dispose de points dans le plan muni d'un repère orthonormé d'origine $\mathcal{O}$. Chaque point possède un couple de coordonnées $(x; y)$ représenté par la liste `[x, y]`. Il s'agit de trier ces points en fonction de leur distance à $\mathcal{O}$, de la plus petite à la plus grande.

1. Écrire une fonction `distance2` qui prend en paramètre une liste de deux nombres nommée `point` qui représente un point du plan, (`point` est la liste des coordonnées d'un point P), et renvoie le carré de la distance euclidienne entre ce point et $\mathcal{O}$.
2. Écrire une fonction `compare` qui prend en paramètres deux listes `p1` et `p2` représentant deux points P_1 et P_2 et qui renvoie -1 si P_1 est plus proche de $\mathcal{O}$ que P_2, 1 si P_2 est plus proche de $\mathcal{O}$ que P_1, et 0 si les deux points sont équidistants de $\mathcal{O}$.
3. Écrire une fonction `tri_points` qui prend en paramètre une liste composée de listes de deux nombres représentant des points du plan et qui trie cette liste suivant les distances entre chacun des points et $\mathcal{O}$. Utiliser le tri à bulles.

Exercice 2.9 : On considère le texte "abababababab" et le motif "abc". Déterminer le nombre de comparaisons effectuées par la fonction `recherche` du cours (algorithme naïf). En déduire le nombre de comparaisons effectuées si le texte "ababab ..." contient 100 caractères, (50 fois "ab"), puis une expression de ce nombre en fonction de n si le texte contient n caractères sur le même modèle.

Exercice 2.10* : Algorithme de Horspool.

1. Reprendre la fonction de traitement d'un motif `distances` et la fonction `boyer_moore`.

On considère l'algorithme de Horspool, pour lequel les décalages ne sont pas effectués par rapport au caractère fautif mais toujours par rapport au caractère le plus à droite du texte parmi les caractères comparés.

Écrire une fonction de recherche `horspool` en effectuant les modifications nécessaires à partir de la fonction `boyer_moore`.

2. On reprend l'énoncé de l'exercice précédent. Procéder à la même étude, (même texte et même motif), du nombre de comparaisons avec la fonction `horspool`.

Exercice 2.11 : Créer une liste de 10000 caractères choisis de manières aléatoires parmi les caractères "A", "C", "G" et "T". Rechercher dans cette liste le motif "CAGCAG" à l'aide de l'une des fonctions de recherche présentées dans le cours, par exemple la fonction `boyer_moore`.

Exercice 2.12 : On se donne un ensemble E de n points réels, un entier k inférieur à n, et un point x qui n'est pas dans E. Il s'agit de trouver parmi les points de E les k plus proches de x. Le mot « proche » sous-entend une notion de distance.

Voici un exemple d'algorithme où on construit une liste appelée voisins qui contient les k plus proches voisins d'un point x parmi les éléments d'un ensemble E représenté par une liste :

```
Pour i allant de 0 à k − 1
    placer le point E[i] dans la liste voisins
Fin pour
Pour i allant de k à n − 1
    Si la distance entre x et E[i] est inférieure à la distance entre x et
    un point de la liste voisins
        supprimer de la liste voisins le point le plus loin de x
        placer dans la liste voisins le point E[i]
    Fin si
Fin pour
```

Définir une fonction `distance`. Écrire une fonction `proches_voisins` qui prend en paramètres x, k et la liste des points.

Indications

Ex. 2.6

Adapter l'algorithme du tri à bulles.

Ex. 2.10

Faire un dessin pour gérer les indices correctement. Les seules différences entre les deux algorithmes concernent l'indice `s`*.*

Ex. 2.11

Les programmes de recherche textuelle peuvent être utilisés avec des chaînes de caractères ou des listes de caractères.

Corrigé du vrai/faux

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

Quelques explications

3. Si la liste est triée, la médiane est « au milieu ».

5. Les algorithmes de recherche textuelle présentés dans le cours constituent un contre exemple.

7. Avec l'algorithme naïf, le nombre de comparaisons est $2(n-1)$.

9. Cela dépend de la propriété et il faut s'assurer de la terminaison de l'algorithme.

10. On utilise un variant de boucle.

❑ **Erreurs classiques et conseils.—**

- Deux boucles `for` imbriquées n'entraînent pas forcément un coût quadratique. Chaque cas particulier doit être étudié.
- Ne pas confondre les notions de variant et d'invariant précisées au chapitre 10. Une boucle `while` s'arrête lorsqu'une condition n'est plus satisfaite. Ceci signifie qu'une expression a changé de valeur donc peut « varier ». Un algorithme est correct si une certaine propriété est vraie. Si cette propriété est vraie après chaque passage dans une boucle, cela signifie qu'elle ne varie pas. Elle est « invariante ».