

**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

**NOUVEAUX
PROGRAMMES** !



PRÉPAS SCIENCES

collection dirigée par Bertrand Hauchecorne

Informatique

Tronc commun

MPSI - PCSI - PTSI

**NOUVEAUX
PROGRAMMES**

Serge BAYS

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



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

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.

OBJECTIFS

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.

- 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.

■ 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 $0, 1, \dots, n-1$, avec la boucle `while` elle prend les valeurs $0, 1, \dots, n$.

■ 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_paires(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 réussit 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 <code>for i in range(1, 5)</code> , la variable <code>i</code> prend successivement cinq valeurs.	☐	☐
2. En Python, une boucle <code>for</code> unique se termine toujours après un nombre fini d'étapes.	☐	☐
3. Avec les instructions <code>i=1</code> puis <code>while i<5: i=i+1</code> , la variable <code>i</code> prend successivement cinq valeurs.	☐	☐
4. Après les instructions <code>lst = []</code> puis <code>lst.append([1, 2])</code> , la liste <code>lst</code> a pour valeur <code>[1, 2]</code> .	☐	☐
5. Après les instructions <code>lst = []</code> puis <code>for i in range(1, 5): lst.append(i)</code> , la liste <code>lst</code> a pour valeur <code>[1, 2, 3, 4]</code> .	☐	☐
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 <code>d</code> un dictionnaire. Si la clé <code>k</code> n'est pas dans ce dictionnaire, l'instruction <code>d[k]=5</code> provoque une erreur.	☐	☐
9. Si <code>d</code> est un dictionnaire, l'instruction <code>d[5]=[10,15]</code> provoque une erreur.	☐	☐
10. Si <code>d</code> est un dictionnaire, l'instruction <code>d[[10,15]]=5</code> 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 pas être une liste. 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)
```

```
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`.

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.

OBJECTIFS

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*.

- 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]
```



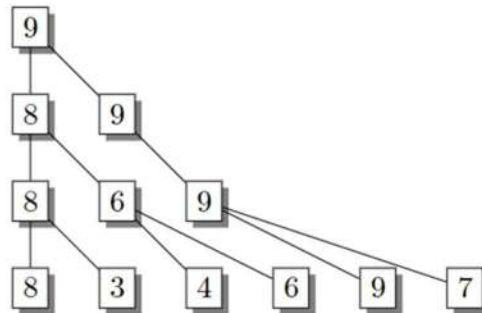
```

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

```

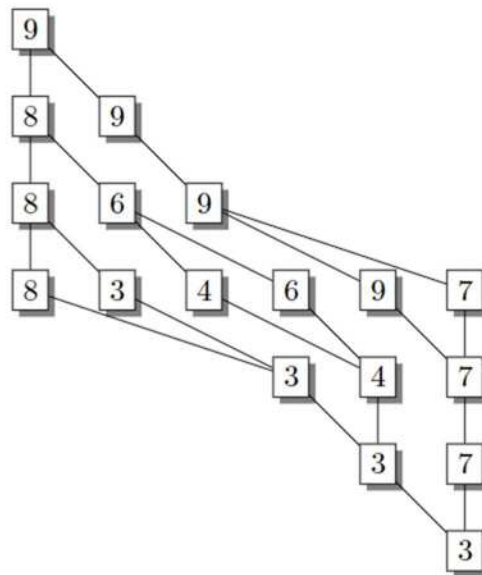
Dans le pire des cas, nous avons 2 comparaisons à chaque passage dans la boucle, donc $2(n-1)$ comparaisons à effectuer. Un même élément de la liste est comparé à `maxi` et à `mini`.

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 maximini2(liste): # la liste est supposée non vide
    n = len(liste)
    if n == 1:

```



```

    return liste[0], liste[0]
liste1 = liste[:]
liste2 = liste[:]
while n >= 2:
    k = n // 2
    mnms1 = k * [0]
    mnms2 = k * [0]
    for i in range(k):
        if liste1[2*i] > liste1[2*i+1]:
            mnms1[i] = liste1[2*i]
        else:
            mnms1[i] = liste1[2*i+1]
        if liste2[2*i] > liste2[2*i+1]:
            mnms2[i] = liste2[2*i+1]
        else:
            mnms2[i] = liste2[2*i]
    if n % 2 == 1:
        mnms1.append(liste1[n-1])
        mnms2.append(liste1[n-1])
    liste1 = mnms1[:]
    liste2 = mnms2[:]
    n = len(mnms1)
return mnms1[0], mnms2[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 est 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 , $\text{abs}(x-y)$, 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 valeurs 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 : $[n, n - 1, \dots, 3, 2, 1]$.

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 $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}$. 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 <code>while</code> 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 "ababababab" 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 ».

■ ■ Corrigé des exercices

Exercice 2.1

La variable *i* varie de 0 à 4. Si *i* vaut 0, *j* varie de 1 à 4, donc cela donne 4 appels. Puis nous avons 3 appels, puis 2 et enfin 1 appel. Le total est $4 + 3 + 2 + 1 = 10$ appels.

Exercice 2.2

1. La variable *i* prend la valeur 0, et il y a une addition.

La variable *j* prend successivement les valeurs 0, 1, 2, avec une addition à chaque passage, donc cela fait 3 additions.

La variable *i* prend la valeur 1, il y a une addition.

La variable *j* prend successivement les valeurs 0, 1, 2, donc cela fait encore 3 additions.

Le total est $2 + 2 \times 3 = 8$ additions.

2. On fait le même type de calcul que pour la question 1 en remplaçant le 3 de la boucle interne par *n*, donc le total est $2 + 2n = 2(n+1)$ additions.

3. Cette fois on remplace le 2 par *n* donc le total est : $n + n \times n = n(n+1)$ additions.

Exercice 2.3

Les valeurs de *x* sont strictement décroissantes et la valeur de *y* est remise à 0 dès qu'elle n'est plus strictement inférieure à celle de *x*. Au dernier passage dans la boucle interne, *y* vaut 0, et *x* vaut 1 : *y* prend alors la valeur 1 et *x* la valeur 0 ; on sort de la boucle interne puis de la boucle externe.

On a une boucle infinie si on remplace $y = 0$ par $y = 1$.

Exercice 2.4

La liste est parcourue du début vers la fin. Donc les différents échanges de valeurs amènent le plus grand élément à la fin. C'est la réponse 2. Exactement $(n - 1)$ comparaisons sont effectuées.

Exercice 2.5

La liste `nombres` est parcourue. Les valeurs 12 et 5 sont échangées, puis 13 et 8, puis 13 et 11 et enfin 13 et 6. La valeur finale de la liste `nombres` est [5, 12, 8, 11, 6, 13].

Exercice 2.6

On adapte le tri à bulles en comparant à chaque étape les longueurs des mots.

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

maliste = ['toto', 'bonjour', 'a', 'oui', 'non']
ordre(maliste)
print(maliste)
```

Exercice 2.7

Il faut commencer par importer la fonction `time` avec l'instruction `from time import time` et écrire le code de la fonction du tri à bulles.

1. La fonction `randint` permet de créer la liste aléatoire.

```
from random import randint

liste = [randint(1, 10000) for i in range(5000)]
top = time()
tri_bulles(liste)
print(time() - top)

liste = [i for i in range(100000)]
top = time()
tri_bulles(liste)
print(time() - top)
```

Le résultat obtenu pour le premier test est de l'ordre de 3 secondes. Pour le second, c'est environ 0,015 seconde. Ces résultats sont bien sûr légèrement variables si on répète l'expérience ou si on change de machine. La différence de coût apparaît nettement, quadratique dans le premier cas et linéaire dans le second.

- 2.

```
liste = [randint(1, 100000) for i in range(100000)]
top = time()
liste.sort()
print(time() - top)
```

Le temps affiché est environ 0.031 seconde. Il montre l'efficacité de cet algorithme.

Exercice 2.8

- 1.

```
def distance2(point):
    x, y = point
    return x * x + y * y
```

- 2.

```
def compare(p1, p2):
    d1 = distance2(p1)
    d2 = distance2(p2)
    if d1 < d2:
```



```

        return -1
    elif d2 < d1:
        return 1
    else:
        return 0

```

3. Le tri à bulles est adapté. Tester avec la liste `[[3, 5], [2, 1], [0, 7], [-2, 0]]`.

```

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

```

Exercice 2.9

On étudie la fonction `recherche`.

Les comparaisons sont marquées par les caractères en gras du motif et leur nombre figure dans la colonne de gauche. Dès qu'une comparaison échoue, le motif est décalé à chaque fois d'une unité vers la droite.

	a	b	a	b	a	b	a	b	a	b
3	a	b	c							
1		a	b	c						
3			a	b	c					
1				a	b	c				
3					a	b	c			
1						a	b	c		
3							a	b	c	
1								a	b	c
2									a	b

Le total est de 18 comparaisons, $16 + 2$. Les 16 sont obtenues par 4 fois $(3 + 1)$ ou 8 fois 2 (en moyenne nous avons 2 comparaisons par caractère du texte) puis les 2 dernières comparaisons avec le motif qui déborde du texte. Le total est donc de 9×2 comparaisons.

En suivant le même principe, pour un texte de 100 caractères, nous avons 98 fois 2 comparaisons puis les 2 dernières avec le motif qui déborde du texte, soit 99×2 comparaisons.

Avec n caractères, nous avons $(n - 1) \times 2$ comparaisons, donc de l'ordre de $2n$ comparaisons.

Exercice 2.10

1. Algorithme de Horspool.

La fonction de traitement du motif est la fonction `distances`. La valeur de `s` ne doit pas être modifiée pendant les comparaisons.

```
def horspool(texte, motif):
    m = len(motif)
    n = len(texte)
    d = distances(motif)
    s = m - 1
    sol = []
    while s < n:
        i = m - 1
        while i >= 0 and motif[i] == texte[s+i-m+1]: # modification
            i = i - 1 # modification
        if i < 0:
            sol.append(s-m+1)
            s = s + 1 # modification
        else:
            s = s + d[texte[s]]
    return sol
```

2. Avec cette fonction, on effectue une comparaison entre le "a" du texte, la troisième lettre, et le "c" du motif qui ne se correspondent pas. On décale alors le motif de deux places vers la droite pour faire correspondre le "a" fautif du texte avec le "a" du motif. Et on poursuit ainsi.

	a	b	a	b	a	b	a	b	a	b
1	a	b	c							
1			a	b	c					
1					a	b	c			
1							a	b	c	

Le total est de 4 comparaisons. Seules les lettres "a" du texte sont comparées, exceptée la première. La dernière lettre est un "b". Le nombre de comparaisons est donc $(10 - 2)/2$.

En suivant le même principe, pour un texte de 100 caractères, nous avons $(100 - 2)/2$, soit un total de 49 comparaisons.

Avec n caractères, nous avons donc de l'ordre de $n/2$ comparaisons. Ce résultat est plus intéressant que celui obtenu à l'exercice précédent.

Exercice 2.11

On écrit le code de la fonction `boyer_moore`, puis le code ci-dessous :

```
from random import randint
choix = "ACGT"

texte = [choix[randint(0, 3)] for i in range(10000)]
motif = "CAGCAG"
print(boyer_moore(texte, motif))
```

Sur quelques essais, on trouve en moyenne trois fois le motif.

```
def proches_voisins(E, x, k, d):
    voisins = []
    for i in range(k):
        voisins.append(E[i])
    for i in range(k, len(E)):
        dist = d(x, E[i])
        u = i
        for j in range(k): # recherche d'un maximum
            if d(voisins[j], x) > dist:
                dist = d(voisins[j], x)
                u = j
        if u != i:
            voisins[u] = E[i]
    return voisins
```

On définit une distance entre deux points, (ici la distance entre deux réels), et une liste de points (ici des nombres réels) :

```
def d(x, y):
    return abs(x - y)

E = list(range(1000))
```

On peut tester le programme dans l'interpréteur :

```
>>> proches_voisins(E, 15.2, 1, d)
[15]
>>> proches_voisins(E, 15.2, 4, d)
[16, 17, 14, 15]
```

Un problème de lenteur apparaît clairement.

Chapitre **3**

Modules et bibliothèques

UN SCIENTIFIQUE



Né en 1950 à Aarhus, **Bjarne Stroustrup** est un informaticien danois. Après des études de mathématiques dans sa ville natale, il se rend à Cambridge, en Angleterre où il obtient un doctorat d'informatique en 1979. Il s'installe alors dans le New Jersey et travaille dans le laboratoire de recherche sur les télécommunications *Bell Labs*. C'est alors qu'il élabore le langage C++ qu'il rend public en 1985. Ce langage évoluera et deviendra l'un des plus répandus au monde.

■ Un peu d'histoire

Depuis les débuts de l'informatique, la capacité des machines s'est démultipliée. Cependant, la plus grande des avancées est permise grâce à l'utilisation d'algorithmes et de programmes déjà créés et auxquels on peut faire appel pour concevoir des nouvelles applications. Ceci permet la gestion de services d'une immense complexité. Si ceci possède d'énormes avantages, il s'avère néanmoins que les bases de certaines applications ne sont plus connues ou mal référencées. C'est ainsi qu'au début de l'informatique, par souci de préserver de la mémoire, on notait parfois les années avec leurs deux derniers chiffres. À l'arrivée de l'an 2000 on a craint que certains programmes considèrent que l'on remonte dans le temps pour se retrouver en 1900. Chaque millénaire a ses angoisses, on souriait à celles de nos ancêtres à l'arrivée de l'an 1000, nous avons eu les nôtres, mille ans plus tard.

OBJECTIFS

L'écriture d'un programme consiste souvent à utiliser du code existant et à y ajouter son propre code pour répondre à un problème particulier.

- Savoir utiliser un module, une bibliothèque :
 - ▶ maîtriser différentes manières d'importer un module et une bibliothèque ;
 - ▶ savoir comment repérer les fonctions et constantes d'un module ;
 - ▶ être capable de trouver des informations dans une documentation.
- Comprendre la modularisation :
 - ▶ être en mesure de créer un programme et l'utiliser dans un autre programme ;
 - ▶ s'habituer à décomposer un programme en plusieurs programmes, une fonction en plusieurs fonctions ;
 - ▶ comprendre l'importance de fournir une documentation.
- Acquérir des compétences sur la gestion des fichiers textes :
 - ▶ savoir lire des données dans un fichier texte ;
 - ▶ connaître le format de fichier CSV ;
 - ▶ comprendre la nécessité des conversions de type.

Il est conseillé de décomposer un programme en particulier à l'aide de fonctions. La relecture en est facilitée et il est plus aisé de tester une à une les différentes parties. Lorsque des fonctions traitent toutes d'un même sujet, on peut les regrouper dans un fichier, un module. Différents modules peuvent être regroupés dans une bibliothèque.

■ Modularité

En général, dans l'écriture d'un programme, il n'est pas question de réécrire ce qui existe déjà. Ce serait beaucoup trop long et improductif. Un programme en Python commence souvent par des lignes contenant le mot `import` afin d'utiliser des modules et des bibliothèques.

Remarque : bibliothèque se traduit en anglais pas library.

Vocabulaire utilisé avec Python

Un *module* peut être un fichier unique (extension `py`). Il contient des variables, des fonctions. Par exemple, le fichier s'appelle `mon_module.py` et contient une fonction qui s'appelle `bonjour`. On peut alors importer ce module dans un programme avec l'instruction `import mon_module`. Pour utiliser la fonction, si elle ne prend aucun paramètre, on écrit `mon_module.bonjour()`.

En Python, un *package* est un ensemble de dossiers et sous-dossiers. Sous Windows ou Linux, on installe une bibliothèque ou un module en téléchargeant un ou plusieurs packages. Ce package contient un fichier `--init--.py` pour indiquer à Python qu'il ne s'agit pas d'un simple répertoire mais bien d'un package. Pour utiliser le module `pyplot`, livré avec le package `Matplotlib`, on écrit l'instruction `import matplotlib.pyplot`.

De manière générale, une *bibliothèque* est constituée de plusieurs modules, soit d'un ou plusieurs packages, chaque package contenant un ensemble de modules.

Une bibliothèque standard est distribuée avec Python. Elle est composée d'un grand nombre d'outils, en particulier des modules destinés à des utilisations très diverses.

Une bibliothèque apporte ce qui n'existe pas de manière basique dans le langage.

■ Modules

Pour créer un module qui est un simple fichier, il suffit donc d'écrire les définitions de fonctions et de constantes dans un fichier dont le nom a l'extension `py`. Il est important de fournir une documentation qui indique la façon de les utiliser et ce qu'elles produisent.

Par exemple, on crée un fichier qui se nomme `aires.py`, fichier dans lequel on précise la valeur utilisée pour le nombre π et les définitions de quelques fonctions permettant de calculer des aires.

```
pi = 3.14159

def disque(rayon):
    """rayon de type float est le rayon d'un disque
    renvoie l'aire du disque"""
    return pi * rayon * rayon
```



```
def rectangle(largeur, longueur):
    """largeur et longueur sont du type float
    renvoie l'aire d'un rectangle de côtés largeur et longueur"""
    return largeur * longueur

def triangle(base, hauteur):
    """base et hauteur de type float sont la base et la hauteur d'un triangle
    renvoie l'aire du triangle"""
    return base * hauteur / 2
```

Importation d'un module

Attention, si le fichier `aires.py` est enregistré dans un dossier quelconque et qu'on écrit `import aires` dans un interpréteur Python, une erreur de type `ModuleNotFoundError` est alors affichée. L'interpréteur nous dit qu'il ne connaît aucun module se nommant '`aires`'.

On pourrait préciser à l'interpréteur Python le chemin d'accès au fichier. Ceci est fait de manière définitive lorsqu'on installe un nouveau module dans la distribution Python. C'est aussi possible ponctuellement en écrivant dans l'interpréteur les commandes suivantes, où `programmes` est le dossier contenant le fichier `aires.py` :

```
>>> from os import chdir
>>> chdir("C:/Users/Toto/docs/programmes")
```

Si les fichiers sont enregistrés dans un même dossier, l'importation ne pose pas de problème.

```
import aires
print(aires.disque(5))
```

Le point entre `aires` et `disque` signifie que l'on se réfère au nom `disque` qui est défini dans le module `aires`. Lorsque le nom du module est long, on peut le renommer avec une instruction comme `import matplotlib.pyplot as plt`.

Une autre possibilité est d'écrire `from aires import disque`. Dans ce cas on peut écrire `disque(5)`. Au moment de l'importation, la fonction `disque` récupère la valeur de `pi` définie dans le fichier `aires.py` et peut l'utiliser. Mais nous n'avons pas accès à cette valeur de `pi`. La commande `print(pi)` entraîne une erreur de nom, `NameError: name 'pi' is not defined`.

On accède à la spécification d'une fonction avec la fonction `help`.

```
>>> import aires
>>> help(aires.disque)
Help on function disque in module aires:

disque(rayon)
```

rayon de type float est le rayon d'un disque
renvoie l'aire du disque

Le contenu du module s'obtient avec la fonction `dir`.

```
>>> import aires
>>> dir(aires)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'disque', 'pi', 'rectangle', 'triangle']
>>>
```

On complète le module avec les instructions suivantes :

```
print("Calcul d'aires")
print("Disque: pi * r^2")
print("Exemple avec r = 5")
print("Aire = pi * 5^2 =", disque(5))
```

Si le fichier est exécuté dans l'interpréteur Python, nous obtenons l'affichage :

```
Calcul d'aires
Disque: pi * r^2
Exemple avec r = 5
Aire = pi * 5^2 = 78.53975
>>>
```

Si le fichier est importé dans un autre programme, lorsque ce second programme est exécuté, l'affichage précédent est à nouveau obtenu alors qu'il n'est pas demandé par ce second programme.

Pour modifier ce type de comportement, qui en général n'est pas souhaité lorsqu'une fonction est importée dans un programme uniquement pour pouvoir être utilisée par le programme, il suffit de compléter le fichier `aires.py` en écrivant à la fin une fonction `main` de la façon suivante :

```
def main():
    """formules de calcul d'aires"""
    print("Calcul d'aires")
    print("Disque: pi x r^2")
    print("Exemple avec r = 5")
    print("Aire = pi x 5^2 =", disque(5))

if __name__ == "__main__":
    main()
```


Principe : une variable `__name__` est créée automatiquement à l'exécution d'un programme. Cette variable contient le nom du programme courant quand on importe un programme. Mais le nom du programme principal est toujours `__main__`. Donc la fonction `main` est exécutée seulement si le fichier `aires.py` constitue le programme principal.

Ainsi, si ce dernier fichier est exécuté tout seul, l'affichage est identique au précédent car la variable `__name__` a la valeur `__main__`.

Mais si ce fichier est importé dans un autre fichier, à l'exécution le test `__name__ == __main__` n'est pas vérifié et la fonction `main` n'est pas exécutée.

Dans cet exemple très simple, les fonctions `disque`, `rectangle`, et `triangle` constituent la partie du programme qui peut être utilisée. Le principe de base est que la logique sous-jacente n'a pas besoin d'être connue du programme utilisateur. Par contre cela nécessite une documentation suffisamment fournie décrivant son utilisation.

Un point important est qu'il est possible de modifier le code de ces fonctions sans qu'un changement ne génère un problème quelconque pour les utilisateurs. Par exemple, si on se rend compte que les valeurs des aires sont très souvent arrondies avec quelques décimales, il est intéressant d'offrir cette possibilité.

```
def disque(rayon, dec = -1):
    """rayon de type float est le rayon d'un disque
    renvoie l'aire du disque"""
    aire = pi * rayon * rayon
    if dec >= 0:
        aire = round(aire, dec)
    return aire
```

Grâce au paramètre facultatif `dec`, la fonction peut être utilisée comme précédemment. Mais pour les utilisateurs intéressés, cela offre une nouvelle possibilité.

```
from aires import disque
print("rayon = 4, aire =", disque(4))
print("rayon = 4, aire =", disque(4, 2))
```

On obtient dans l'interpréteur les valeurs 50.26544 et 50.27.

■ Exemples de modules et bibliothèques

■ Module math

La fonction `dir` nous permet d'explorer le contenu du module `math`.

L'instruction `import math` est obligatoire. Si elle n'est pas écrite, l'appel `dir(math)` provoque une erreur de nom : `"name 'math' is not defined"`.

Ce module contient des constantes comme π , noté `pi`, et des fonctions mathématiques. Nous pouvons obtenir des informations sur une fonction particulière à l'aide de la fonction `help`.

Dans la liste des fonctions, nous remarquons la fonction `hypot`. Qu'obtient-on avec `help`?


```
>>> help(math.hypot)
Help on built-in function hypot in module math:

hypot(...)
    hypot(x, y)

    Return the Euclidean distance, sqrt(x*x + y*y).
```

C'est bien la longueur de l'hypoténuse d'un triangle rectangle de petits côtés `x` et `y`.

■ Module random

Pour obtenir des informations sur un module, des documentations en ligne sont consultables, par exemple pour le module `random` : <https://docs.python.org/3/library/random.html>.

Comme précédemment, nous pouvons observer dans l'interpréteur le contenu de ce module avec les instructions `import random`, puis `dir(random)`.

Quelques fonctions qui renvoient des nombres de manière aléatoire sont souvent utilisées.

```
>>> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

La fonction `randint` renvoie un entier compris entre `a` et `b`. Nous sommes informés que les bornes `a` et `b` sont incluses alors que la borne `b` est exclue avec `randrange(a, b)`.

L'expression `randint(a, b)` a exactement pour valeur `randrange(a, b+1)`.

Une autre fonction souvent utilisée est `random` qui renvoie un nombre de type float compris entre 0 et 1, la borne 1 étant exclue.

■ Module Turtle

Le module `turtle` permet de dessiner en programmant. On commence par tout importer. Puis on utilise des instructions simples comme `forward(x)` pour avancer de `x` pixels, donc tracer un segment de longueur `x` pixels, `left(d)` pour tourner à gauche de `d` degrés.

```
from turtle import *

def triangle(pixel):
    for i in range(3):
        forward(pixel)
        left(120)

triangle(200)
```

Nous exécutons ce programme et obtenons dans la fenêtre de dessin un triangle équilatéral.

Voici quelques fonctions dont la signification est presque évidente :

`forward(distance)`, `backward(distance)`, pour avancer ou reculer ;

`left(angle)`, `right(angle)`, pour tourner à gauche ou à droite ;

`reset()`, pour effacer ;

`goto(x, y)`, `up()`, `down()`, pour aller à un point donné, lever ou baisser le crayon ;

`color('couleur')`, `width(épaisseur)`, pour la couleur et l'épaisseur du trait ;

`write('texte')`, pour afficher du texte ;

`begin_fill()`, `end_fill()`, pour remplir une figure fermée avec une couleur.

Le point central a pour coordonnées (0;0). Le programme est écrit dans un fichier puis exécuté et on observe le résultat. L'utilisation de l'interpréteur est intéressante pour tester des instructions.

```
>>> from turtle import *
>>> forward(100)
>>> goto(-80,150)
```

Le résultat de chaque instruction est immédiatement visible sur la figure.

Ce module permet de mesurer l'intérêt de définir des fonctions et d'utiliser des boucles pour obtenir des figures complexes.

On trouve de nombreux exemples sur internet et une documentation est disponible à l'adresse : <https://docs.python.org/fr/3.8/library/turtle.html>.

Un dessin simple :

```
couleurs = ['blue', 'green', 'yellow', 'orange', 'red', 'purple']
bgcolor('black')

def dessin():
    for i in range(180):
        color(couleurs[i%6])
        forward(i)
        right(59)

dessin()
```

La tortue, ou le crayon, que l'on voit se déplacer et dessiner est une instance de la classe `Turtle`. Les mots *instance* et *classe* font partie du vocabulaire de la programmation orienté objet. Disons simplement que la tortue est un objet.

Le dessin commence lorsqu'on exécute le programme. Remplaçons l'instruction `dessin()` par les deux instructions qui suivent et exécutons le programme.

```
listen()
onkey(dessin, "a")
```


La fenêtre de dessin s'ouvre mais il ne se passe rien. Le programme est en attente d'un évènement. Plus précisément il est à l'écoute avec la fonction `listen` qui fixe le focus sur la fenêtre de dessin. La fonction `onkey` prend deux paramètres : une fonction sans arguments et une chaîne de caractères qui représente une touche du clavier. Avec la chaîne "a" comme dans l'exemple, l'appui sur la touche A du clavier entraîne l'exécution de la fonction `dessin`.

- ▶ Le dessin commence lorsque la touche A du clavier est relâchée.
- ▶ À chaque appui et relâchement de la touche, le dessin recommence mais à partir de la position et de la direction courantes du crayon.

Ce module permet ainsi de découvrir très simplement la programmation événementielle, style de programmation très répandu sur Internet.

■ Bibliothèques

Une bibliothèque est en général une collection de modules. Il en existe un très grand nombre en Python qui viennent compléter suivant les besoins la bibliothèque standard. Celle-ci contient les modules `math` et `random` parmi beaucoup d'autres.

Parmi ces bibliothèques, deux sont importantes à ce niveau.

- ▶ **Matplotlib** pour tout ce qui concerne les graphiques.

La bibliothèque Matplotlib contient le module `pyplot` utilisé pour le tracé de courbes. Il est aussi possible de tracer des histogrammes et toutes sortes de représentations graphiques.

- ▶ **Pillow ou PIL** pour la manipulation et le traitement d'images. (Voir chapitre 7).

Une documentation précise et de très nombreux exemples d'utilisation se trouvent facilement par une recherche sur Internet.

Le principe pour tracer une courbe

Le tracé d'une courbe s'obtient avec des points dont on précise les coordonnées dans deux listes. Ce tracé nécessite donc la constitution de deux listes, la liste des abscisses et celle des ordonnées.

Voici une syntaxe minimale pour obtenir un tracé :

```
import matplotlib.pyplot as plt

def f(x): # une fonction
    return x ** 2 - 5 * x + 1

x = [i * 0.01 for i in range(101)] # la liste des abscisses
y = [f(u) for u in x] # la liste des ordonnées

plt.plot(x, y) # construction de la courbe
plt.grid() # un quadrillage facultatif
plt.show() # affichage de la courbe
```

La fonction `plot` construit la courbe, la fonction `show` l'affiche à l'écran. Nous pouvons ainsi obtenir plusieurs courbes sur la même figure.


```
def f(x): # une fonction
    return x ** 2 - 3 * x + 2

def g(x): # une fonction
    return - x ** 2 + x + 1

x = [1 + i * 0.01 for i in range(101)]
y1 = [f(u) for u in x]
y2 = [g(u) for u in x]

plt.plot(x, y1) # construction de la courbe 1
plt.plot(x, y2) # construction de la courbe 2
plt.show() # affichage de la courbe
```

Pour avoir une idée de l'étendue des possibilités de représentation graphique, il suffit de regarder les exemples présentés à l'adresse <https://matplotlib.org/gallery/index.html>.

■ Lecture et écriture de fichiers textes

■ Ouverture et fermeture d'un fichier

La fonction `open` prend deux paramètres, le nom du fichier et le mode d'ouverture : 'w' pour le mode "écriture", 'r' pour le mode "lecture" et 'a' pour le mode "ajout", (write, read, append).

La syntaxe est `fic = open('fichier', 'w')` pour écrire, `fic = open('fichier', 'r')` pour lire, `fic = open('fichier', 'a')` pour ajouter du texte à la fin du fichier.

Il est essentiel de fermer un fichier qui a été ouvert. L'instruction est : `fic.close()`.

■ Écriture et lecture

La méthode `write` prend en paramètre une chaîne de caractères (type `str`), par exemple : `fic.write("J'écris dans un fichier")`.

Pour écrire sur plusieurs lignes, on utilise le caractère `\n` qui force un retour à la ligne. Par exemple : `fic.write("J'écris dans un fichier\nA la fin, je le ferme.")`, ou en plus lisible `fic.write("J'écris dans un fichier" + "\n" + "A la fin, je le ferme.")`.

Si le fichier est fermé puis à nouveau ouvert en écriture, c'est un nouveau fichier qui est écrit. Ce qui était écrit auparavant est perdu. Pour écrire à nouveau dans un fichier déjà fermé, on l'ouvre en mode "ajout". Le texte est écrit à la suite du précédent.

```
fic = open('fichier.txt', 'w')
fic.write("J'écris dans un fichier\nA la fin, je le ferme.")
fic.close()

fic = open('fichier.txt', 'a')
fic.write("Je continue.")
fic.close()
```

L'extension "txt" peut être remplacée par "dat" ou "csv". Le fichier écrit peut être ouvert dans un éditeur de texte. Par défaut, il est créé dans le dossier où est enregistré le fichier Python.

Si les données à écrire sont de type numérique, il faut les convertir au préalable en type str. Dans le format CSV, les données sont séparées par un point-virgule (une virgule chez les anglophones).

```
a, b, c = 2, 5, 8
fic = open('fichier.csv', 'w')
fic.write(str(a) + ";" + str(b) + ";" + str(c) + "\n")
fic.write(str(2*a) + ";" + str(2*b) + ";" + str(2*c))
fic.close()
```

Lecture

L'ouverture d'un fichier en mode lecture s'effectue avec la méthode `read`.

Plusieurs instructions sont disponibles : `ch = fic.read(n)` lit `n` caractères, `ch = fic.read()` lit tout le fichier, `ch = fic.readline()` lit la ligne courante et passe à la suivante. Dans les trois cas, la variable `ch` obtenue est une chaîne de caractères.

L'instruction `ch = fic.readlines()` lit toutes les lignes. Dans ce cas, la variable `ch` obtenue est une liste de chaînes de caractères. Chaque élément est une ligne du fichier.

L'instruction `ch = [x for x in fic]` produit le même résultat.

Deux méthodes sur les chaînes de caractères sont importantes dans le traitement des données lues. Si `ch` est une chaîne de caractères, alors :

`ch.rstrip()` supprime le caractère de fin de ligne (par exemple `"\n"`);

`ch.split(sep)` coupe la chaîne `ch` suivant le délimiteur `sep`, et renvoie une liste de sous-chaînes de `ch`. Avec `ch = "un,deux,trois"`, `ch.split(',')` renvoie `['un', 'deux', 'trois']`.

Le séparateur est une chaîne de caractères. Par défaut, c'est l'espace.

Si les données à lire sont des nombres qui doivent être utilisés dans des calculs, il faut les convertir en type `int` ou en type `float` suivant les besoins.

Voici un exemple :

```
fic = open("fichier.dat", "w")
fic.write(str(5) + "\t" + str(8.3) + "\t" + str(1e-4) + "\n")
fic.write(str(8) + "\t" + str(32.7) + "\t" + str(1e2))
fic.close()

fic = open("fichier.dat", "r")
for ligne in fic:
    liste = ligne.rstrip().split("\t")
    a, b, c = [float(x) for x in liste]
    print(a, b, c)

fic.close()
```

■ ■ Vrai/Faux

	Vrai	Faux
1. Pour importer deux fonctions <code>f1</code> et <code>f2</code> d'un module <code>mod</code> , on écrit <code>from mod import f1 and f2</code> .	☐	☐
2. Pour importer une fonction <code>f1</code> d'un module <code>mod</code> , on écrit <code>import f1 from mod</code> .	☐	☐
3. Dans l'écriture <code>mystere.myst</code> , <code>mystere</code> peut être une fonction ou une constante et <code>myst</code> un module.	☐	☐
4. Une bibliothèque peut contenir de nombreux modules.	☐	☐
5. La fonction <code>plot</code> du module <code>pyplot</code> de Matplotlib permet de construire une courbe mais pas de l'afficher.	☐	☐
6. Afin de construire les points de coordonnées (1;2) et (3;4), on écrit l'instruction <code>plt.plot([1, 2], [3, 4])</code> .	☐	☐
7. Pour ouvrir en lecture un fichier nommé <code>fichier.txt</code> , on peut écrire <code>f = open(fichier.txt, r)</code> .	☐	☐
8. L'instruction <code>readline(3)</code> permet de lire trois lignes dans un fichier.	☐	☐
9. Pour lire ou écrire des nombres entiers dans un fichier il faut les convertir au type <code>str</code> .	☐	☐
10. Seul un tableur permet de créer un fichier au format CSV.	☐	☐

■ ■ Énoncé des exercices

Exercice 3.1 : Pour importer dans un fichier la fonction `sqrt` du module `math`, on peut écrire :

1. `import sqrt from math`
2. `insert sqrt from math`
3. `from math import sqrt`
4. `from math insert sqrt`

Exercice 3.2 : On simule une marche aléatoire. Le point de départ est l'origine du repère. Les abscisses représentent les numéros (entiers) des n étapes. La distance à l'origine est représentée en ordonnée. À chaque étape, cette distance augmente de 1 ou diminue de 1 de manière équiprobable.

Tracer une figure avec Matplotlib en prenant par exemple $n = 500$ (n est un paramètre).

Exercice 3.3 :

1. Écrire une fonction `maximum` qui renvoie le maximum de deux nombres et une fonction `minimum` qui renvoie le minimum de deux nombres, sans utiliser les fonctions `min` et `max`. Ces deux fonctions sont écrites dans un fichier. Ce fichier est enregistré sous le nom `mesfonctions.py` et constitue un module.
2. Écrire dans un autre fichier, après avoir importé le module précédent, une fonction qui renvoie le maximum de quatre nombres.

Exercice 3.4 : Créer un module `stats` qui contient deux fonctions `somme` et `moyenne`. Ces deux fonctions prennent une liste de nombres non vide en paramètre. La fonction `somme` renvoie la somme des éléments de la liste et la fonction `moyenne` renvoie la moyenne des éléments de la liste. Tester le module en l'important dans un autre fichier où les deux fonctions sont utilisées.

Il est possible ensuite de compléter le module `stats` avec d'autres fonctions qui peuvent être utiles en statistique.

Exercice 3.5 : Une série statistique est donnée par une liste dont les éléments sont des listes de deux nombres représentant une valeur et un effectif. Écrire une fonction qui prend en paramètre une liste représentant une série statistique et renvoie la moyenne pondérée de cette série. Le programme doit utiliser le module créé dans l'exercice précédent.

Exercice 3.6 : Variance

Écrire une fonction `variance` qui prend en paramètres une liste de nombres et renvoie la variance de ces nombres. Utiliser la fonction `moyenne` du module `stats`.

Pour rappel, le calcul de la variance s'effectue ainsi : on calcule la somme des carrés des valeurs, somme que l'on divise par le nombre d'éléments, autrement dit, on calcule la moyenne des carrés des valeurs. Ensuite on retranche à cette moyenne le carré de la moyenne des valeurs.

Exercice 3.7 : On dispose de données écrites dans un fichier nommé "data.csv". La première ligne contient le texte `largeur;longueur;hauteur`. Chaque ligne suivante du fichier contient trois valeurs, qui représentent des flottants, séparées par des virgules, par exemple `25.3,37.5,13.8`.

1. Écrire un code permettant de lire les lignes de données et de stocker l'ensemble des lignes dans une liste dont chaque élément est une liste constituée par les trois valeurs de type float d'une ligne.
2. Utiliser la fonction `moyenne` du module `stats` (exercice 4) pour calculer la hauteur moyenne.
3. Modifier le code pour ne récupérer qu'une ligne de données sur dix (la première ligne de données, la onzième, etc).

Exercice 3.8 : Création d'un module

On considère un fichier `dessins.py` qui contient le code suivant :

```
from turtle import *
import figures as f

up()
goto(50, -60)
down()
f.rectangle(100, 75, 'red')

up()
goto(-80, -20)
down()
f.triangle(85, 'blue')

up()
goto(20, 60)
down()
f.cercle(60, 'green')
```

Il s'agit d'écrire le fichier `figures.py` qui est importé dans le programme ci-dessus. Lorsque le programme est exécuté, on obtient dans la fenêtre graphique du module Turtle un rectangle, un triangle équilatéral et un cercle avec des couleurs différentes. De plus le périmètre et l'aire de chaque figure sont affichés avec le dessin.

Les principales commandes du module Turtle sont résumées à ces deux adresses :

https://fr.wikibooks.org/wiki/Programmation_Python/Turtle

<https://docs.python.org/fr/3.8/library/turtle.html>.

Exercice 3.9 : L'objectif est d'écrire une fonction qui trace, à l'aide de la bibliothèque Matplotlib, la courbe représentant une fonction f sur un intervalle $[a; b]$, en utilisant n points. On importe au préalable le module `pyplot` de Matplotlib.

Écrire une fonction `trace` qui prend en arguments deux nombres `a` et `b`, une fonction `f` et un entier `n`. L'appel `trace(a, b, f, n)` permet d'obtenir le tracé de la courbe.

Exercice 3.10* : Image bitmap.

Le format PBM est l'un des plus simples pour définir une image en noir et blanc. Il s'agit d'un fichier texte dont le contenu est :

- ▶ sur la première ligne, P1 qui définit le format ;
- ▶ sur la ligne suivante, facultative, un commentaire qui commence par le symbole # ;

- sur la ligne suivante, la largeur et le hauteur de l'image en pixel ;
- sur les lignes suivantes, des 0 et des 1 qui représentent chacun un pixel en blanc ou en noir.

Par exemple :

```
P1
# Un exemple d'image bitmap
10 12
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0
0 0 1 1 0 0 1 1 0 0 0 0
0 0 1 1 0 0 1 1 0 0 0 0
0 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 0 0
1 1 0 0 0 0 0 0 1 1 1 1
1 1 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0
```

Un fichier est enregistré avec l'extension pbm, comme `image.pbm`. Pour visualiser l'image, il suffit de l'ouvrir avec un logiciel comme XnView ou GIMP.

Une image est représentée par une matrice, une liste de listes. Pour l'exemple ci-dessus, on obtient : `[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0], [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0], [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0], [1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1], [1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]`

On définit la matrice d'une image exemple avec la fonction suivante :

```
def matrice(n, p):
    m = [[0 for j in range(p)] for i in range(n)]
    for i in range(n): # n est la hauteur de l'image
        for j in range(p): # p est la largeur de l'image
            if (j // 20) % 2 == 0:
                m[i][j] = 1
    return m
```

Écrire une fonction `fichier` qui prend en paramètres une matrice comme celle définie ci-dessus, et une chaîne de caractères représentant le nom d'un fichier, sans l'extension. La fonction crée le fichier image, avec l'extension pbm, et y écrit les lignes définissant l'image avec les valeurs données par la matrice.

Après avoir créer une matrice avec l'instruction `m1 = matrice(240, 180)`, entrer l'instruction `fichier(m1, "image1")` et vérifier qu'un fichier `image1.pbm` a été créé. Visualiser l'image avec par exemple le logiciel XnView ou le logiciel GIMP.

■ ■ Indications

Ex. 3.2

Construire deux listes, une pour les abscisses et l'autre pour les ordonnées.

Ex. 3.4

Commencer par écrire une fonction, la tester, puis écrire la seconde fonction et la tester. Ensuite tester l'utilisation du module dans un autre fichier.

Ex. 3.7

Pour garder une ligne sur dix, penser à une expression comme $i \% 10$.

Ex. 3.8

Procéder pas à pas en écrivant une fonction simple et en la testant pour améliorer le code avant de passer à la suite. Quand toutes les fonctions sont correctes, tester alors le module.

Ex. 3.9

La fonction construit les deux listes de n valeurs (abscisses et ordonnées), appelle la fonction `plot` puis la fonction `show`.

■ ■ Corrigé du vrai/faux

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

Quelques explications

1. On écrit `from mod import f1, f2`. Le mot `and` est un *et* logique.
3. C'est le contraire.
5. C'est la fonction `show` qui provoque l'affichage.
9. Seule l'écriture nécessite cette conversion.
10. N'importe quel éditeur de texte peut être utilisé.

❑ Erreurs classiques et conseils.—

- Une erreur classique est de confondre le *et* utilisé dans la langue courante et le *et* (`and`) utilisé en logique.
- Les méthodes d'importation d'un module et d'utilisation d'éléments du module doivent être parfaitement maîtrisées. La notation avec un point entre le nom d'un module et le nom d'une fonction doit être connue.
- La lecture de données dans un fichier est une activité importante dans de nombreux domaines. Il faut pratiquer les différentes méthodes régulièrement.

■ ■ Corrigé des exercices

Exercice 3.1

De manière générale, on écrit : `from module import fonction`. C'est la réponse 3.

Exercice 3.2

```
import matplotlib.pyplot as plt
from random import randint

def marche(n): # n nombre d'étapes
    y = 0
    yliste = [0] # la liste des ordonnées
    for i in range(n):
        y = y + 2 * randint(0, 1) - 1 # pour choisir -1 ou 1 "au hasard"
        yliste.append(y)
    xliste = list(range(n+1))
    plt.plot(xliste, yliste, "r")
    plt.grid()
    plt.show()

marche(500)
```

Exercice 3.3

1. Les fonctions du module.

```
def maximum(a, b):
    """ a et b sont des nombres de type int ou float
    la fonction renvoie le maximum des deux nombres """
    if a > b:
        return a
    else:
        return b

def minimum(a, b):
    """ a et b sont des nombres de type int ou float
    la fonction renvoie le minimum des deux nombres """
    if a < b:
        return a
    else:
        return b
```

2. Il faut importer le module ou la fonction utilisée (`from mesfonctions import maximum`). Le fichier suivant et le fichier `mesfonctions.py` sont placés dans le même répertoire.


```
import mesfonctions as mf

def maxi(a, b, c, d):
    """ a, b, c et d sont des nombres de type int ou float
    la fonction renvoie le maximum des quatre nombres"""
    m1 = mf.maximum(a, b)
    m2 = mf.maximum(c, d)
    m = mf.maximum(m1, m2)
    return m
```

Exercice 3.4

On écrit dans un fichier nommé `stats.py` les codes des deux fonctions.

```
def somme(liste):
    s = 0
    for x in liste:
        s = s + x
    return s

def moyenne(liste):
    s = somme(liste)
    n = len(liste)
    return s / n
```

Le fichier est enregistré dans un répertoire. Dans ce même répertoire on écrit un fichier `test.py`.

```
import stats as st

liste = [1, 2, 3, 4, 5]
print(st.somme(liste))
print(st.moyenne(liste))
```

Exercice 3.5

Il faut créer la liste qui est donnée en paramètre à la fonction `moyenne` du module `stats`.

```
import stats as st

def moyenne_pond(liste):
    liste2 = []
    for [x, n] in liste:
        for i in range(n):
```

```
        liste2.append(x)
    return st.moyenne(liste2)
```

Exercice 3.6

```
import stats as st

def variance(liste):
    n = len(liste)
    m1 = st.moyenne(liste)
    carres = [x ** 2 for x in liste]
    m2 = st.moyenne(carres)
    return m2 - m1 ** 2
```

Comme pour le calcul de la moyenne, le coût est linéaire en n la taille de la liste.

Exercice 3.7

1.

```
fic = open('data.csv', 'r')
fic.readline() # pour séparer la première ligne de texte
liste = []
for ligne in fic:
    valeurs = ligne.rstrip().split(',')
    liste.append([float(x) for x in valeurs])
fic.close()
```

2. Importer le module comme précédemment `import stats as st`. Construire la liste des hauteurs en écrivant par exemple `hauteurs = [x[2] for x in liste]` et utiliser la fonction `moyenne` avec `st.moyenne(hauteurs)`.

3. On ajoute un compteur. En commençant le compte à 0, on garde les lignes 1, 11, 21, etc. Les numéros des lignes sont donc de la forme $10 \times k + 1$.

```
fic = open('data.csv', 'r')
liste = []
i = 0
for ligne in fic:
    if i % 10 == 1:
        valeurs = ligne.rstrip().split(',')
        liste.append([float(x) for x in valeurs])
    i = i + 1
fic.close()
```

Exercice 3.8

Les possibilités sont multiples. Penser à enregistrer les fichiers `dessins.py` et `figures.py` dans le même répertoire.

```
from turtle import *

def rectangle(long, larg, coul='black'):
    color(coul)
    for i in range(2):
        forward(long)
        left(90)
        forward(larg)
        left(90)
    p = 2 * (long + larg)
    a = long * larg
    color('black')
    write("aire = " + str(a) + "\n" + "périmètre = " + str(p))

def triangle(long, coul='black'):
    color(coul)
    for i in range(3):
        forward(long)
        left(120)
    p = 3 * long
    a = long * long * 1.732 / 2
    color('black')
    write("aire = " + str(a) + "\n" + "périmètre = " + str(p))

def cercle(r, coul='black'):
    color(coul)
    circle(r)
    p = 2 * 3.14159 * r
    a = 3.14159 * r * r
    color('black')
    write("aire = " + str(a) + "\n" + "périmètre = " + str(p))
```

Exercice 3.9

```
import matplotlib.pyplot as plt

def trace(a, b, f, n):
    xliste = []
    yliste = []
    h = (b - a) / (n - 1)
    for i in range(n):
        x = a + i * h
```



```
xliste.append(x)
yliste.append(f(x))
plt.plot(xliste, yliste)
plt.show()
```

Si le nombre de points utilisés est souvent le même, on peut lui donner une valeur par défaut en écrivant : `def trace(a, b, f, n=50)`.

Après avoir défini une fonction `g`, l'appel de la fonction `trace` peut s'écrire de deux manières :

```
>>> trace(-1, 4, g) # par défaut n vaut 50
>>> trace(-1, 4, g, 40) # la valeur de n est fixée à 40
```

Cette fonction peut ensuite être importée et utilisée dans un autre fichier.

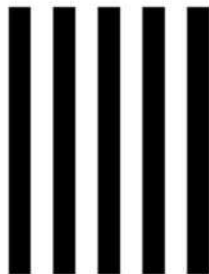
Exercice 3.10

La fonction fichier :

```
def fichier(mat, nom):
    hauteur = len(mat)
    largeur = len(mat[0])
    nom = nom + '.pbm'
    f = open(nom, 'w')
    f.write('P1\n' + str(largeur) + ' ' + str(hauteur) + '\n')
    for i in range(hauteur):
        for j in range(largeur):
            bit = str(mat[i][j])
            f.write(bit + ' ')
    f.close()

fichier(m1, "image1")
```

On obtient l'image suivante :



Algorithmes dichotomiques

UN SCIENTIFIQUE



Après des études en Californie, le mathématicien américain **Derrick Lehmer** (1905-1991) devient un spécialiste de la théorie des nombres.

Il perfectionne un test imaginé par Édouard Lucas pour déterminer si un nombre est premier.

Dès 1945, il collabore à l'élaboration de programmes pour l'ENIAC, l'un des tout premiers ordinateurs. Il élabore alors différents algorithmes et améliore la méthode de dichotomie pour établir celui qui porte le nom de Lehmer-Schur.

■ Un peu d'histoire

La méthode de dichotomie existe depuis l'Antiquité. Le mot *dichotomie* est formé sur le grec, *dikha*, en deux et de *tomia* qui signifie division. On a retrouvé des exemples de tri par dichotomie dès l'Antiquité, chez les Babyloniens et les peuples de la mer Égée. Cette méthode est utilisée depuis longtemps dans des démonstrations mathématiques comme celle du théorème de Bolzano-Weierstrass qui affirme que de toute suite bornée de réels, on peut extraire une sous-suite convergente.

Les premiers algorithmes de tri à visée informatique apparaissent en 1957 avec William Peterson et en 1962 avec Derrick Lehmer. En 1986 les mathématiciens Bernard Chazelle et Leonidas Guibas ont introduit des méthodes tout à fait nouvelles. Depuis, l'élaboration d'algorithmes toujours plus rapides permet de gérer l'immense masse de données dont nous disposons.

OBJECTIFS

Les algorithmes présentés dans ce chapitre utilisent un principe simple mais qui n'est pas toujours facile à mettre en œuvre. Ils illustrent la nécessité de tester et de valider un programme.

- Comprendre le principe de la recherche dichotomique :
 - ▶ connaître les conditions d'utilisation ;
 - ▶ savoir programmer une recherche dichotomique sur un tableau trié ;
 - ▶ produire une preuve de terminaison et de correction.
- Comprendre l'intérêt d'un algorithme dichotomique :
 - ▶ envisager une stratégie dichotomique pour remplacer un algorithme naïf ;
 - ▶ appréhender la notion de complexité logarithmique ;
 - ▶ être capable d'évaluer la complexité d'un algorithme dichotomique.
- Acquérir des compétences sur la manière de tester et de prouver :
 - ▶ découvrir et comprendre comment organiser un jeu de tests ;
 - ▶ s'habituer à mettre en œuvre des outils de validation.

Le mot *dichotomie* vient du grec et signifie *division en deux parties*. Une méthode dichotomique est une stratégie de type *diviser pour régner*.

■ Recherche dichotomique

Trier des nombres est une activité ancienne qui remonte à plus de deux mille ans.

D'après Donald Knuth, John Mauchly qui travaillait sur le projet ENIAC, dont l'objectif était la construction d'un ordinateur entièrement programmable, est le premier à avoir mentionné la recherche dichotomique dans un papier en 1946. Mais sa méthode n'était prévue que pour des tableaux dont la taille était de la forme $n = 2^p - 1$. Quand on retire l'élément central du tableau, il reste $2^p - 2$ éléments qui sont répartis dans deux parties de même taille, $2^{p-1} - 1$, et ainsi de suite jusqu'à obtenir un seul élément. Cette méthode était utilisée par les programmeurs. Elle traite le cas le plus simple. Par exemple avec 127 éléments, on passe à 63 éléments, puis 31, 15, 7, 3 et 1. Remarquons que les nombres d'éléments successifs s'écrivent en binaire 1111111, 111111, 11111, 1111, 111, 11, 1. Le coût de l'algorithme est de l'ordre du nombre de divisions, donc de l'ordre du nombre de chiffres dans l'écriture binaire de n , soit à peu près $\log_2(n)$.

Il semble que Derrick Henry Lehmer est le premier en 1960 à avoir publié un algorithme de recherche dichotomique qui fonctionne pour tout n . D. H. Lehmer était un mathématicien, travaillant également autour du projet ENIAC, qui spécialisé dans la théorie des nombres.

Le coût d'une recherche linéaire d'un élément dans un tableau de taille n est linéaire en n dans le pire des cas qui consiste à tester tous les éléments du tableau. Le cas d'un tableau dont les éléments sont triés, par exemple si les éléments sont rangés dans l'ordre croissant, n'améliore pas cette complexité.

L'intérêt de la méthode dichotomique, est d'accélérer grandement la recherche. Mais cette méthode n'est applicable qu'à une condition : que le tableau soit trié. *Le principe est de comparer l'élément avec celui se trouvant au milieu du tableau. S'ils sont égaux, c'est terminé, sinon on recommence avec la partie du tableau pouvant contenir l'élément.*

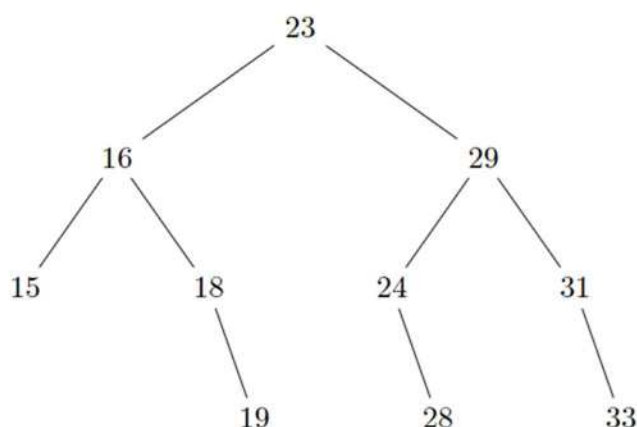
Voici une application de cette méthode sur un exemple. On cherche l'élément 28 qui est compris entre les valeurs extrêmes du tableau 15 et 33. Les « milieux » utilisés pour les tests sont grisés et désignés par les flèches verticales. Les extrémités des sous-tableaux sont soulignées.

15	16	18	19	23	24	28	29	31	33	28 > 23
15	16	18	19	23	24	28	29	31	33	28 < 29
15	16	18	19	23	24	28	29	31	33	28 > 24
15	16	18	19	23	24	28	29	31	33	28 = 28

Si on cherche l'élément 27, le processus est le même, seule la conclusion change.

15	16	18	19	23	24	28	29	31	33	27 > 23
15	16	18	19	23	24	28	29	31	33	27 < 29
15	16	18	19	23	24	28	29	31	33	27 > 24
15	16	18	19	23	24	28	29	31	33	27 ≠ 28

Nous pouvons représenter cette recherche dichotomique, en anglais « binary search », sous la forme d'un arbre avec les éventuels milieux successifs utilisés :



Cette représentation nous montre bien qu'en passant par les « milieux » on atteint n'importe lequel des dix éléments en au plus quatre tests.

■ Programmation

Les tableaux sont représentés en Python par des listes. Nous écrivons le code d'une fonction qui prend en paramètres une liste triée et l'élément cherché. La fonction peut renvoyer `True` si l'élément est trouvé ou son indice. La difficulté est de prévoir les cas, qu'il vaut mieux tester après l'écriture d'un code, d'une liste à deux éléments, un élément, ou d'une liste vide, avec la recherche d'un élément qui appartient à la liste ou pas.

Une recherche par dichotomie s'effectue dans un tableau trié. Un tableau est représenté par une liste (type `list`). Si la liste n'est pas triée, nous disposons en Python de la fonction `sorted` qui prend en argument une liste et renvoie la liste triée. La syntaxe est `lst2 = sorted(lst1)`. La liste initiale `lst1` n'est pas modifiée. Python propose aussi la méthode `sort` qui trie en place la liste à laquelle elle s'applique. La syntaxe est `lst.sort()`.

■ Principe de l'algorithme :

à chaque étape, on coupe le tableau en deux et on effectue un test pour savoir dans quelle partie se trouve l'élément cherché.

■ Première version

```
def dichotomie(x, liste):
    g = 0
    d = len(liste)
    while g < d - 1:
        k = (g + d) // 2
        if x < liste[k]:
            d = k
        else:
            g = k
    if x == liste[g]:
        return g
    else:
        return False
```

Preuve de la terminaison

Nous utilisons l'expression $d - g$ qui à chaque étape représente la taille de la sous-liste qui peut contenir x . Cette expression est *un variant de boucle*.

Soit p le plus grand entier tel que la taille n du tableau est inférieure à 2^p . Avant d'entrer dans la boucle $d - g \leq 2^p$.

Après k itérations, $d - g \leq \frac{2^p}{2^k}$, soit $d - g \leq 2^{p-k}$. La suite des valeurs de l'expression $d - g$ est strictement décroissante et après p étapes, $d - g \leq 1$, donc $g \geq d - 1$ et la boucle est terminée.

Par exemple, il faut sept étapes pour une taille de tableau égale à 100 et 10 étapes pour une taille égale à 1000.

Ceci prouve aussi que le coût de cette recherche dichotomique est de l'ordre du nombre de chiffres dans l'écriture binaire de n , donc de l'ordre de $\log_2(n)$. Ce coût est nettement inférieur à celui d'une recherche linéaire. On dit que la complexité est *logarithmique*.

Avant l'entrée dans la boucle, g a pour valeur 0 et d a pour valeur `len(liste)`. Si $x < \text{liste}[0]$, ou si $x > \text{liste}[d-1]$, alors x n'est pas dans la liste et la recherche n'aboutira pas.

Il est possible de vérifier avant de commencer la recherche que le nombre cherché est bien compris entre les deux bornes de la liste. On utilise pour cela une *assertion* qui arrête le programme si la condition à vérifier n'est pas remplie. Ce procédé est développé au chapitre 9.

La syntaxe est `assert liste[g] <= x <= liste[d-1]`, à placer avant la boucle.

Si l'assertion est vérifiée, alors `liste[g] ≤ x ≤ liste[d-1]` avant l'entrée dans la boucle.

Preuve de la correction

Nous ajoutons un élément en fin de liste, strictement supérieur au dernier élément, par exemple égal à `liste[d-1]+1`. Nous allons démontrer que la propriété `liste[g] ≤ x < liste[d]` qui est vraie avant l'entrée dans la boucle reste encore vraie après chaque passage dans la boucle. Cette propriété est *un invariant de la boucle*.

La propriété est vraie avant l'entrée dans la boucle d'après ce qui précède.

Supposons qu'elle soit vraie avant un passage dans la boucle : `liste[g] ≤ x < liste[d]`.

D'après le choix de k , `liste[g] ≤ liste[k] ≤ liste[d]` puisque la liste est triée.

Donc, si $x < \text{liste}[k]$, on obtient `liste[g] ≤ x < liste[k]`. Dans ce cas la nouvelle valeur de d est k , et donc `liste[g] ≤ x < liste[d]` après le passage dans la boucle.

Sinon, $\text{liste}[k] \leq x$, et on obtient $\text{liste}[k] \leq x < \text{liste}[d]$. Dans ce cas la nouvelle valeur de g est k , et donc $\text{liste}[g] \leq x < \text{liste}[d]$ après le passage dans la boucle.

Il reste à examiner l'état des variables après la sortie de la boucle.

Lorsque le dernier passage dans la boucle est effectué, $d - g \geq 2$. Donc $g < k < d$. Si d prend la valeur k ou si g prend la valeur k , alors $d - g > 0$. Mais s'il n'y a plus de passage dans la boucle, cela signifie que $d - g \leq 1$. La seule possibilité est donc $d - g = 1$.

Or $\text{liste}[g] \leq x < \text{liste}[d]$, donc $\text{liste}[g] \leq x < \text{liste}[g+1]$. En conclusion, nous n'avons que deux possibilités : soit x est la valeur de $\text{liste}[g]$, soit x n'est pas dans la liste.

Remarques

- Il est intéressant de tester la fonction avec plusieurs types de listes. Une liste contenant un seul élément, deux éléments, un nombre pair d'éléments et un nombre impair d'éléments avec une valeur cherchée présente ou non dans chaque liste.
- Le principe de dichotomie s'applique en mathématiques pour trouver une solution approchée d'une équation du type $f(x) = 0$ sur un intervalle $[a; b]$ à epsilon près. La fonction f est supposée monotone continue.

```
def solution(f, a, b, epsilon):
    while b - a > epsilon:
        m = (a + b) / 2
        if f(a) * f(m) <= 0:
            b = m
        else:
            a = m
    return (a + b) / 2
```

À chaque étape, l'amplitude de l'intervalle contenant la solution est divisée par deux. Ceci signifie que la précision gagne un chiffre dans l'écriture binaire. (10 chiffres dans l'écriture binaire correspondent à environ 3 chiffres dans l'écriture décimale).

■ Deuxième version

```
def dichotomie(liste, x):
    g, d = 0, len(liste) - 1
    while g <= d:
        m = (g + d) // 2
        if x == liste[m]:
            return True
        else:
            if x > liste[m]:
                g = m + 1
            elif x < liste[m]: # ou else ...
                d = m - 1
    return False
```

Les choix $g = m + 1$ et $d = m - 1$ permettent d'obtenir dans le cas où la longueur de la liste est de la forme $2^n - 1$ des sous-listes de taille $2^k - 1$. Une explicitation des détails de ce programme, en particulier l'usage des indices $m-1$ et $m+1$, est développée en exercice.

La terminaison et la correction de l'algorithme se démontrent comme avec la première version. La complexité est aussi évaluée d'une manière similaire à celle déjà utilisée.

■ Exponentiation rapide

Il s'agit d'écrire une fonction `puissance` qui prend en paramètres un flottant x non nul et un entier naturel n et qui renvoie x^n .

Dans une version naïve, on calcule successivement x^2 , x^3 , etc, en se basant sur la définition mathématique : pour $n \geq 0$, $x^n = 1 \times x \times \dots \times x$ où n est le nombre de facteur égaux à x .

```
def puissance1(x, n):
    p = 1
    for i in range(n):
        p = p * x
    return p
```

La fonction `puissance1` effectue n produits et $n+1$ affectations. La complexité est donc linéaire en fonction de n .

Remarque

Dans le calcul du coût, un autre facteur ne doit pas être négligé, la taille de x . En Python, il n'y a pas de limite de taille pour les entiers. Donc si x est de type `int`, le temps de calcul peut être nettement allongé selon la taille de x .

Méthode dichotomique

Pour tout n , si n est pair, $x^n = (x^2)^{n/2}$, sinon $x^n = x(x^2)^{n/2}$, où $n/2$ désigne le quotient entier de n par 2.

Nous en déduisons l'algorithme suivant :

- ▶ p vaut 1
- ▶ Tant que $n/2$ n'est pas nul
 si n est impair nous remplaçons p par px
 nous remplaçons x par x^2 et n par $n/2$.

Le principe d'un algorithme dichotomique est de diviser un problème en deux sous-problèmes.

```
def puissance2(x, n):
    p = 1
    while n != 0:
        if n % 2:
            p = p * x
        x = x * x
        n = n // 2
    return p
```


Complexité logarithmique.

Considérons l'écriture binaire de n (des précisions sont données au chapitre 11).

$$n = n_0 + n_1 \times 2 + n_2 \times 2^2 + \dots + n_k \times 2^k = \sum_{i=0}^k n_i 2^i, \text{ avec pour tout } i, n_i \text{ égal à } 0 \text{ ou à } 1.$$

Nous pouvons alors écrire : $x^n = x^{\sum_{i=0}^k n_i 2^i} = x^{n_0} (x^2)^{n_1} (x^2)^{n_2} \dots (x^2)^{n_k}$.

Dans le cas le moins favorable, nous pouvons considérer que nous avons de l'ordre de $4k$ opérations à effectuer, où k est le nombre de chiffres dans l'écriture binaire de n , soit environ $\log_2(n)$ opérations (k multiplications, k élévations au carré, et k calculs de quotient et de reste).

Le logarithme en base b d'un nombre entier n est de l'ordre du nombre de chiffres dans l'écriture en base b de n .

Si $c_{10}(n)$ est le nombre de chiffres dans l'écriture décimale de n , alors nous pouvons en déduire que $c_{10}(n) - 1 \leq \log_{10}(n) < c_{10}(n)$, soit $\log_{10}(n) < c_{10}(n) \leq \log_{10}(n) + 1$.

Si $c_2(n)$ est le nombre de chiffres dans l'écriture binaire de n , alors nous pouvons en déduire que $c_2(n) - 1 \leq \log_2(n) < c_2(n)$, soit $\log_2(n) < c_2(n) \leq \log_2(n) + 1$.

■ Concaténation de chaînes

On donne une liste de n chaînes de caractères, chacune de longueur p . L'objectif est d'écrire une fonction qui renvoie la chaîne obtenue par concaténation des n chaînes et d'étudier sa complexité en fonction de n et de p . Pour cette étude nous considérons qu'une chaîne est lue et écrite en mémoire caractère par caractère, qu'il n'y a pas plus de lectures que d'écritures et donc que les opérations *élémentaires* à compter sont uniquement les écritures de caractères. On suppose enfin que l'affectation `ch1 = ch1 + ch2`, nécessite de réécrire en mémoire les caractères de la chaîne `ch1` suivis des caractères de la chaîne `ch2`.

- 1er cas : les n chaînes sont identiques de longueur p .

En Python, on peut utiliser l'opérateur `*`.

```
def concat1(liste):  
    return n * liste[0]
```

La longueur de la chaîne renvoyée est prévisible, c'est $n \times p$.

La fonction écrit les caractères de la chaîne `liste[0]` l'un après l'autre, n fois à la suite. Le nombre d'écritures est donc $p \times n$.

Il est clair qu'on peut difficilement faire mieux.

- 2e cas : les n chaînes sont distinctes.

En Python, on peut utiliser l'opérateur `+`.

```
def concat2(liste):  
    ch = ''  
    for chaine in liste:  
        ch = ch + chaine  
    return ch
```


À chaque passage dans la boucle, pour concaténer `ch` et `chaine`, tous les caractères de la valeur courante de `ch` sont écrits. La première chaîne de la liste est donc écrite n fois, la deuxième $n-1$ fois, etc, jusqu'à la dernière écrite une fois. Le nombre total d'écritures est donc $pn + p(n-1) + \dots + p = pn(n+1)/2$. Le coût est donc de l'ordre de $p \times n^2/2$. Ceci signifie une complexité quadratique.

Peut-on être plus efficace ? Dans ce chapitre, il apparaît que pour certains algorithmes, on peut passer d'un coût linéaire à un coût logarithmique en utilisant une méthode par dichotomie.

Cela signifie que l'on coupe la liste en deux parties de même longueur à une unité près. On concatène les chaînes de la première moitié, les chaînes de la deuxième moitié, puis on concatène les deux résultats. Pour concaténer les chaînes de chaque partie, on recommence la procédure avec chaque partie, et ainsi de suite.

Cette procédure décrit une approche dite *descendante* qui peut être programmée de manière récursive (voir chapitre 5). L'algorithme programmé ci-dessous suit une approche dite *ascendante*. Les chaînes consécutives de la liste sont concaténées deux par deux. Nous avons donc $n//2$ concaténations de deux chaînes de longueur p . La dernière chaîne si elle est seule est ajoutée à cette liste. On recommence alors l'opération avec $n//4$ concaténations de deux chaînes de longueur $2p$, et on continue ce processus jusqu'à obtenir une seule chaîne de longueur np .

Les concaténations au niveau 1 coûtent $2p \times n/2 = p \times n$ opérations.

Les concaténations au niveau 2 coûtent $4p \times n/4 = p \times n$ opérations.

Et ainsi de suite. Le nombre de niveaux étant environ $\log_2(n)$, le nombre total d'opérations est de l'ordre de $p \times n \log_2(n)$.

Nous pouvons donc envisager une complexité de l'ordre de $p \times n \log_2(n)$.

Pour simplifier l'écriture du programme proposé ci-dessous, on suppose que la taille de la liste est une puissance de 2. La variable `taille` représente la longueur des chaînes à concaténer divisée par p .

```
def concat3(liste):
    n = len(liste)
    taille = 1
    while taille < n:
        for g in range(0, n-taille, 2*taille):
            liste[g] = liste[g] + liste[g+taille]
        taille = 2 * taille
    return liste[0]
```

Remarques

- Les chaînes créées sont nombreuses et il apparaît ici une autre forme de coût, que l'on appelle un coût en espace. En effet ces chaînes occupent de l'espace en mémoire et cet espace doit être évalué dans certains cas.
- On obtiendrait le même type de résultat en remplaçant les chaînes par des listes. Mais Python propose des méthodes plus efficaces, liées au caractère *mutable* des listes. Par exemple, il faut éviter d'écrire la concaténation `lst = lst + [elt]` et utiliser plutôt `lst.append(elt)`.

■ ■ Vrai/Faux

	Vrai	Faux
1. Une recherche dichotomique sur un tableau non trié a un coût inférieur à celui d'une recherche linéaire.	☐	☐
2. S'il faut environ un temps t pour trouver par dichotomie une valeur dans une liste triée de taille n , alors il faut environ le temps $2t$ pour une liste de taille $2n$.	☐	☐
3. S'il faut environ un temps t pour trouver par dichotomie une valeur dans une liste triée de taille n , alors il faut environ le temps $10t$ pour une liste de taille $1000n$.	☐	☐
4. Avec une liste triée, une recherche dichotomique est toujours plus rapide qu'une recherche séquentielle.	☐	☐
5. Dans une liste triée contenant un million d'entiers, on peut vérifier la présence d'un élément avec au plus vingt tests.	☐	☐
6. Avec une liste triée de taille n , le nombre d'étapes dans la recherche dichotomique d'une valeur est de l'ordre du nombre de chiffres dans l'écriture binaire de n .	☐	☐
7. Pour toutes les listes triées de longueur n tel que $2^{p-1} < n \leq 2^p$, p est le nombre maximum d'étapes dans la recherche dichotomique d'une valeur.	☐	☐
8. Pour prouver la terminaison de l'algorithme de recherche dichotomique, on peut examiner les variations d'une expression qui représente la taille de la partie du tableau contenant éventuellement l'élément.	☐	☐
9. C'est une propriété invariante qui a permis dans le cours de prouver la correction de l'algorithme de la recherche dichotomique.	☐	☐
10. On peut calculer x^8 avec trois multiplications.	☐	☐

■ ■ Énoncé des exercices

Exercice 4.1 : On utilise la première version de la fonction `dichotomie` présentée dans le cours.

1. On recherche dans la liste `[2, 4, 6]` la valeur 6 puis la valeur 7 à l'aide de cette fonction. Expliciter dans chaque cas les valeurs successives des variables `g`, `d` et `k`
2. Peut-on avoir une des valeurs successives de `liste[d]` égale à la valeur cherchée `x` ?

Exercice 4.2 : Une recherche est effectuée, à l'aide de la deuxième version de la fonction `dichotomie` du cours, dans la liste `[12, 15, 16, 18, 21, 24, 25, 27, 31, 33, 35, 36, 38]`.

Donner l'ordre de parcours des différentes valeurs dans une recherche dichotomique.

Exercice 4.3* : On étudie le code de la deuxième version de la fonction `dichotomie` présentée dans la partie cours. Il s'agit d'expliciter certains détails de ce programme.

1. Quel est le problème si on remplace la condition `g <= d` par la condition `g < d` ?
2. Quel est le problème si on remplace l'affectation `g = m+1` par l'affectation `g = m` ?
3. Quel est le problème si on remplace l'affectation `d = m-1` par l'affectation `d = m` ?
4. Lorsque la condition `g <= d` n'est plus satisfaite, quelle relation peut-on écrire entre `g` et `d` ? Que peut-on dire alors de `liste[g]`, `x` et `liste[d]` ?

Pour les questions 1, 2 et 3, on peut considérer une liste à deux éléments comme `[2, 4]` et la recherche de la valeur 4 pour les questions 1 et 2, celle de la valeur 3 pour la question 3.

Exercice 4.4* : Il s'agit de simuler un jeu où un nombre entier est choisi au hasard par la machine, entre 1 et 100, ou 1 et 1000 par exemple, et le joueur doit deviner ce nombre en un minimum de coups. À chaque étape le joueur propose un nombre, entré au clavier, et la machine répond en affichant un message disant si le nombre est trop grand, trop petit, ou si le joueur a gagné.

1. Écrire une fonction `devine` qui prend en argument un entier `n` et réalise ce jeu où l'entier à deviner est compris entre 1 et `n`

Quelle est la stratégie optimale permettant de gagner avec le minimum de coups ?

2. Améliorer le jeu en affichant à chaque étape le nombre de coups joués, ou le nombre de coups qui restent à jouer si un nombre de coups maximal a été défini.

Exercice 4.5 : Tester puis corriger la fonction suivante qui doit renvoyer une solution approchée sur un intervalle $[a; b]$ de l'équation $f(x) = 0$. La fonction f est supposée continue monotone et a une solution unique sur $[a; b]$.

```
def dichotomie(f, a, b, p):
    while b-a > p:
        m = (a + b) / 2
        if f(a) * f(m) < 0:
            b = m
        else:
            a = m
    return (a + b) / 2
```


Exercice 4.6* : Écrire une fonction `insertion` qui prend en paramètres une liste de nombres triée et un nombre x et insère le nombre x dans la liste de manière à ce que la liste reste triée. La fonction utilise un algorithme dichotomique pour trouver la bonne place et ne renvoie rien.

Exercice 4.7 : On considère le code d'exponentiation rapide de la fonction `puissance2` du cours. Décrire les valeurs successives des variables pour obtenir le résultat `puissance2(2, 9)`.

Exercice 4.8 : On considère le code d'exponentiation rapide de la fonction `puissance2` du cours. Remplacer `p` par `s` en initialisant `s` à la valeur 0. Remplacer ensuite `p = p * x` par `s = s + x` et `x = x * x` par `x = x + x`. On nomme cette fonction `mult_russe`.

```
def mult_russe(x, n):
    s = ...
    while n != 0:
        if n % 2:
            s = ...
        x = ...
        n = n // 2
    return s
```

À quoi sert cette fonction ?

Exercice 4.9 : On considère le code de la fonction de l'exercice précédent. Décrire les valeurs successives des variables pour obtenir le résultat `mult_russe(3, 7)`.

Exercice 4.10 : On considère le code de la fonction `puissance2` et l'écriture binaire d'un entier.

1. Écrire une fonction qui calcule un encadrement entre deux entiers consécutifs du logarithme en base 2 de n avec $n > 0$.
2. Vérifier la correction du programme.
3. Écrire ensuite sur le même modèle une fonction qui calcule un encadrement entre deux entiers consécutifs du logarithme décimal de n avec $n > 0$.

Exercice 4.11 : La méthode de Héron, connue des babyloniens et des égyptiens, mais exposée par Héron d'Alexandrie il y a environ 2000 ans, est une méthode qui permet de calculer une valeur approchée de la racine carrée d'un nombre positif, c'est-à-dire une valeur approchée de la solution positive de l'équation $x^2 - a = 0$. C'est une concurrente très sérieuse de la méthode par dichotomie.

Le principe est de construire une suite d'abscisses telles que x_0 est la valeur initiale choisie, par exemple $x_0 = 1$, et $x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k} \right)$ pour tout entier k .

1. Écrire une fonction `heron` qui prend en arguments un nombre positif `a` et un nombre `epsilon`. La fonction `heron` renvoie une solution approchée positive x de l'équation $x^2 = a$ telle que $|x^2 - a| \leq \epsilon$ où ϵ est la valeur de `epsilon`.
2. Tester le programme pour différentes valeurs de `a` et de `epsilon`.

■ ■ Indications

Ex. 4.3

Pour la question 4, examiner séparément les cas qui conduisent à une situation $g > d$.

Ex. 4.4

Voici deux instructions utiles pour choisir un nombre au hasard et pour enregistrer un nombre entré au clavier :

```
from random import randint
nombre = randint(1, n)
# n est défini préalablement

# choix du joueur
choix = int(input("Entrer un nombre: "))
```

Ex. 4.5

Utiliser des fonctions simples pour les tests, comme f et g définies respectivement par $f(x) = x$ et $g(x) = x^2$.

Ex. 4.6

Utiliser un code semblable à celui d'une recherche dichotomique.

Ex. 4.10

Si $2^p \leq n < 2^{p+1}$, alors $p \leq \log_2(n) \leq p+1$ et si $10^p \leq n \leq 10^{p+1}$, alors $p \leq \log_{10}(n) < p+1$.

■ ■ Corrigé du vrai/faux

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

Quelques explications

1. Le tableau doit être trié pour pouvoir utiliser une recherche dichotomique.
2. Ce serait le cas si le coût était linéaire.
3. En dix étapes on se ramène à une liste de taille inférieure à n . Le temps est environ $t + 10$.
4. Ce n'est pas le cas par exemple si l'élément cherché est le premier de la liste.
8. Cette expression s'appelle *un variant de boucle*.

❑ Erreurs classiques et conseils.—

- Une recherche dichotomique sur une liste ne s'effectue que si la liste est triée. Il faut prendre soin de vérifier que cette condition est satisfaite.
- Les puissances de 2 jouent un rôle important. Il faut connaître quelques approximations, en particulier $2^{10} \simeq 10^3$, $2^{20} \simeq 10^6$.

■ ■ Corrigé des exercices

Exercice 4.1

1. Pour la recherche de la valeur 6, les valeurs successives de g et d sont : 0 et 3 au début ; k prend la valeur 1 donc nous avons ensuite 1 et 3 ; k prend la valeur 2 donc nous avons 2 et 3 et la sortie de la boucle. Il en est de même pour les valeurs successives de g et d dans la recherche de la valeur 7. Seule la conclusion est différente.

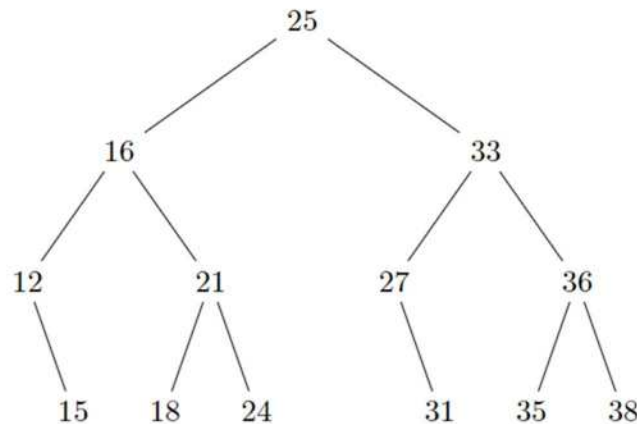
2. Si la valeur initiale de d ne change pas, ceci signifie que la valeur cherchée est supérieure ou égale à la valeur de $liste[d-1]$. C'est la valeur de g qui est modifiée à chaque passage dans la boucle jusqu'à obtenir $g = d-1$. Et $liste[d]$ n'existe pas.

Si la valeur initiale de d est modifiée, cela intervient après une comparaison $x < liste[k]$. Et d prend alors la valeur de k . Donc on a bien $x < liste[d]$ et la réponse est non.

Exercice 4.2

Au début, la valeur de l'indice gauche g est 0 et celle de l'indice droit d est 12. L'indice de m est donc 6 et correspond à l'élément 25.

Au niveau suivant on prend le « milieu » de la partie gauche, (indices 0 et 5), soit l'indice 2 qui correspond à l'élément 16 et le milieu de la partie droite, (indice 7 et 12), soit l'indice 9 qui correspond à l'élément 33. On continue ainsi de suite.



Exercice 4.3

1. Nous remplaçons la condition $g \leq d$ par la condition $g < d$.

Considérons la liste $[2, 4]$. Nous cherchons l'élément x qui a pour valeur 4.

Les valeurs initiales de g et d sont 0 et 1. La condition $g < d$ est satisfaite, et donc m prend la valeur 0. C'est ensuite la condition $x > liste[m]$ qui est satisfaite, donc g prend la valeur de $m+1$, soit 1. Les valeurs de g et d sont alors identiques et on sort de la boucle `while`. Donc la fonction renvoie `False` alors qu'elle doit renvoyer `True`.

Avec la condition $g \leq d$, on reste dans la boucle `while`. La variable m prend la valeur 1, puis la condition $x == liste[m]$ est satisfaite et la fonction renvoie `True`.

2. Nous remplaçons l'affectation $g = m + 1$ par l'affectation $g = m$.

Considérons à nouveau la liste $[2, 4]$ et nous cherchons l'élément x qui a pour valeur 4.

Les valeurs initiales de g et d sont 0 et 1. La condition $g \leq d$ est satisfaite, donc m prend la valeur 0. C'est ensuite la condition $x > liste[m]$ qui est satisfaite, donc g prend la valeur de m soit 0. Les valeurs de g et d n'ont pas changé et le programme entre dans une boucle infinie.

3. Nous remplaçons l'affectation $d = m - 1$ par l'affectation $d = m$.

Nous considérons encore la liste [2, 4] mais nous cherchons cette fois un élément x qui a pour valeur 3 et qui n'est donc pas présent dans la liste.

Les valeurs initiales de g et d sont 0 et 1. La condition $g \leq d$ est satisfaite, donc m prend la valeur 0. C'est ensuite la condition $x > \text{liste}[m]$ qui est satisfaite, donc g prend la valeur de $m+1$, soit 1. La condition $g \leq d$ est encore satisfaite, donc m prend la valeur 1. C'est ensuite la condition $x < \text{liste}[m]$ qui est satisfaite, donc d garde la valeur 1 et le programme entre dans une boucle infinie.

4. La situation $g > d$ peut se produire dans deux cas :

- soit après l'affectation $g = m+1$ si $g = m = d$;
- soit après l'affectation $d = m-1$ si $g = m = d$ ou si $g = m = d-1$.

Nous construisons un exemple pour chaque cas qui doit conduire à $g = m = d$ ou $g = m = d-1$.

- Premier cas : nous sommes dans le cas $g = m = d$.

L'instruction $g = m+1$ est exécutée si $x > \text{liste}[g]$, et donc $x > \text{liste}[d]$. Ceci signifie que x est strictement compris entre $\text{liste}[m]$ et $\text{liste}[m+1]$, donc n'est pas dans la liste. Nous obtenons donc à la fin : $\text{liste}[d] < x < \text{liste}[g]$.

Exemple avec la liste [2, 4, 6] et la recherche de la valeur 3 : les valeurs successives du couple d'indices (g, d) sont (0, 2) et m prend la valeur 1, puis (0, 0) et m prend la valeur 0 (nous en sommes à $g = d = m$), et enfin (1, 0). La fonction renvoie False.

- Deuxième cas : l'instruction $d = m-1$ est exécutée si $x < \text{liste}[d]$, et donc $x < \text{liste}[g]$. Ceci signifie que x est strictement compris entre $\text{liste}[m-1]$ et $\text{liste}[m]$, donc n'est pas dans la liste. Nous avons donc à la fin : $\text{liste}[d] < x < \text{liste}[g]$.

Exemple avec la liste [2, 4] et la recherche de la valeur 3 : les valeurs successives du couple d'indices (g, d) sont (0, 1) et m prend la valeur 0, puis (1, 1) et m prend la valeur 1 (nous en sommes à $g = d = m$), et enfin (1, 0). La fonction renvoie False.

Avec la liste [2, 4, 6, 8] et la recherche de 5 : les valeurs successives du couple d'indices (g, d) sont (0, 3) et m prend la valeur 1, puis (2, 3) et m prend la valeur 2 (nous en sommes à $g = d-1 = m$), et enfin (2, 1). La fonction renvoie False.

Exercice 4.4

1. Programme de base :

```
from random import randint

def jeu(n):
    nombre = randint(1, n)
    choix = -1
    while choix != nombre:
        choix = int(input("Entrer un nombre: "))
        if choix < nombre:
            print("Trop petit")
        elif choix > nombre:
            print("Trop grand")
        else:
            print("Gagné !")
```


La stratégie optimale permettant de gagner avec le minimum de coups utilise la dichotomie.

2. Avec un compteur :

```
def jeu(n):
    nombre = randint(1, n)
    cpt = 0
    choix = -1
    while choix != nombre:
        choix = int(input("Entrer un nombre: "))
        cpt = cpt + 1
        if choix < nombre:
            print("Trop petit")
        elif choix > nombre:
            print("Trop grand")
        else:
            print("Gagné en", cpt, "coups")
```

On peut ajouter une instruction comme `print("Nombre de coups joués:", cpt)` ou, si `cmax` est le nombre maximal de coups, `print("Il vous reste", cmax-cpt, "coups à jouer")`. On ajoute alors la condition `cpt < cmax` pour la boucle `while` et l'affichage de "Perdu" si `cpt > cmax`.

Exercice 4.5

Plusieurs types de tests sont à envisager avec des fonctions simples comme la fonction identité ou la fonction carrée. Tester les cas où la solution est a , b , $(a + b)/2$.

Quelques exemples de tests qui montrent un problème :

```
def f(x):
    return x

def g(x):
    return x ** 2 - 9

print(dicho(f, -1, 2, 0.01))
print(dicho(f, -3, 3, 0.01))
print(dicho(f, 0, 3, 0.01))
print(dicho(g, 0, 10, 0.01))
print(dicho(g, 0, 6, 0.01))
```

Les deux premiers affichages sont -0.0009765625 et 2.9970703125. Dans le premier cas, on approche bien la solution 0 et dans le second cas on approche la borne 3. Il y a donc une erreur dans le programme.

À chaque passage dans la boucle, la solution x_0 doit appartenir à $[a; b]$. Or, si $f(a)$ vaut 0, lors d'un passage dans la boucle, a prend la valeur m et la solution n'appartient plus à $[a; b]$.

On peut corriger le code en remplaçant $f(a) * f(m) < 0$ par $f(a) * f(m) \leq 0$


```
def insertion(liste, x):
    liste.append(x)
    n = len(liste)
    g, d = 0, n-1
    while g <= d:
        m = (g + d) // 2
        if x < liste[m]:
            d = m - 1
        else:
            g = m + 1
    for j in range(g+1, n):
        liste[n+g-j] = liste[n+g-j-1]
    if g < n:
        liste[g] = x
```

Exercice 4.7

La valeur initiale de p est 1, la valeur initiale de n est 9, donc on entre dans la boucle.

1er passage : n est impair donc p prend la valeur 2, x prend la valeur 4, n prend la valeur 4.

2e passage : n est pair donc p garde la valeur 2, x prend la valeur 16, n prend la valeur 2.

3e passage : n est pair donc p garde la valeur 2, x prend la valeur 256, n prend la valeur 1.

4e passage : n est impair donc p prend la valeur 512, x prend la valeur 65536, et enfin n prend la valeur 0, c'est la sortie de boucle.

La valeur de p ($512 = 2^9$) est renvoyée.

Exercice 4.8

Il s'agit de l'algorithme de multiplication à la russe. On n'utilise que des additions et des divisions par 2.

Par exemple : $5 \times 8 = 8 + 4 \times 8 = 8 + 2 \times 16 = 8 + 1 \times 32 = 40$.

```
def mult_russe(x, n):
    s = 0
    while n != 0:
        if n % 2:
            s = s + x
        x = x + x
        n = n // 2
    return s
```

Un autre exemple : 16×7 avec 16 additions ou alors avec 4 divisions par 2 et 4 multiplications par 2 : $16 \times 7 = 8 \times 14 = 4 \times 28 = 2 \times 56 = 1 \times 112$.

Avec l'exponentiation rapide, on calcule $x \times x \times \dots \times x$ avec n facteurs. Pour la multiplication à la russe, on calcule $x + x + \dots + x$ avec n termes. C'est le même algorithme où on remplace l'opérateur $\times$ par l'opérateur $+$.

Exercice 4.9

La valeur initiale de s est 0, la valeur initiale de n est 7, donc on entre dans la boucle.

1er passage : n est impair donc s prend la valeur 3, x prend la valeur 6, n prend la valeur 3.

2e passage : n est impair donc s prend la valeur 9, x prend la valeur 12, n prend la valeur 1.

3e passage : n est impair donc s prend la valeur 21, x prend la valeur 24, n prend la valeur 0 et c'est la sortie de boucle.

La valeur de s ($21 = 3 \times 7$) est renvoyée.

Exercice 4.10

1. On appelle la fonction `loga2`, pour logarithme approché en base 2.

```
def loga2(n):
    s = 0
    while n != 1:
        s = s + 1
        n = n // 2
    return s, s+1
```

2. Les valeurs successives de n constituent une suite strictement décroissante d'entiers naturels qui converge donc vers 1 en un nombre fini d'étapes. Ceci prouve la terminaison.

L'expression n constitue un variant de boucle.

Il existe un entier p tel que la propriété $2^p \leq 2^s \times n < 2^{p+1}$ est vraie après chaque passage dans la boucle. (p est la partie entière de $\log_2(n)$). Soit s_1 et n_1 les valeurs de s et n avant un passage dans la boucle, s_2 et n_2 les valeurs de s après le passage. Si $2^p \leq 2^{s_1} \times n_1 < 2^{p+1}$, alors $2^{p-s_1} \leq n_1 < 2^{p+1-s_1}$. Donc $2^{p-s_1-1} \leq n_1/2 < 2^{p+1-s_1-1}$, d'où $2^{p-s_2} \leq n_2 < 2^{p+1-s_2}$, donc $2^p \leq 2^{s_2} \times n_2 < 2^{p+1}$. À la fin de la boucle, la valeur de n est 1, donc $2^p \leq 2^s < 2^{p+1}$ et $s = p$.

3. Sur le même modèle que `loga2`, la fonction `loga10`.

```
def loga10(n):
    s = 0
    while n != 1:
        s = s + 1
        n = n // 10
    return s, s+1
```

Exercice 4.11

1. On programme dans une boucle `while` le calcul des termes de la suite.

```
def heron(a, epsilon):
    x = 1
    while abs(x * x - a) > epsilon:
        x = (x + a / x) / 2
    return x
```

2. Voici quelques tests pour vérifier la fonction :

```
>>> heron(36, 0.01)
6.0002529841194185
>>> heron(2, 0.001)
1.4142156862745097
>>> heron(0.25, 0.0001)
0.5000000232305737
```


Fonctions récursives

UN SCIENTIFIQUE



Mathématicien hors pair, **Jacques Herbrand** (1908-1931) entre à dix-sept ans à l'École normale supérieure et il est reçu premier à l'agrégation de mathématiques à vingt ans. Ses contributions en logique mathématique sont fondamentales, en particulier sur la théorie de la démonstration. Dans un échange de lettres avec Kurt Gödel, il suggère au logicien autrichien la notion de fonction récursive. Il se tue à vingt-trois ans lors d'une randonnée en montagne.

■ Un peu d'histoire

Au début du XX^e siècle, des logiciens tels Frege ou Russel entamèrent des études cherchant à étayer les mathématiques sur des bases théoriques et non sur la réalité physique qui avait été un guide des thèmes de recherche pendant des siècles ce qui amène au développement de la logique mathématique.

Dans ce cadre-là, Jacques Herbrand imagine la notion de fonction récursive l'année même de sa mort accidentelle. Les logiciens Kurt Gödel, Alonso Church et Emil Post prennent le relais.

Alan Turing conçoit en 1936 une méthode abstraite qui cherche à formaliser la notion d'algorithme et pour laquelle celle de fonction récursive prend toute sa pertinence.

Il ne manquait plus que des machines très puissantes pour donner un sens concret à ces recherches. En particulier, les fonctions récursives trouvent toute leur place dans le cadre de l'intelligence artificielle.

OBJECTIFS

Les algorithmes proposés dans les chapitres précédents sont itératifs. Cela signifie qu'ils sont conçus à l'aide de boucles. Ce chapitre présente une autre manière d'écrire un algorithme, en se passant des boucles.

- Comprendre la notion de récursivité :
 - ▶ comprendre le principe d'une fonction récursive ;
 - ▶ reconnaître un programme récursif et le distinguer d'un programme itératif ;
 - ▶ savoir écrire une fonction récursive (principe du test d'arrêt) ;
 - ▶ parvenir à formuler une solution récursive à un problème ;
 - ▶ être capable de transformer un programme itératif en un programme récursif et réciproquement.
- Continuer à appréhender les notions de validité et de complexité d'un algorithme :
 - ▶ savoir comment établir la correction d'un programme récursif simple ;
 - ▶ faire le lien avec la terminaison d'un programme itératif ;
 - ▶ pouvoir évaluer le nombre d'appels récursifs sur des exemples classiques.

■ Notion de récursivité

■ Introduction

Considérons une fonction qui dessine des carrés dont les dimensions des côtés sont contenus dans une liste. Nous disposons d'une fonction qui dessine un carré et d'une fonction qui l'appelle de manière répétitive à l'aide d'une boucle. Nous utilisons le module Turtle.

```
from turtle import *
def carre(n):
    for i in range(4):
        forward(n)
        left(90)

def dessine1(liste, angle):
    for d in liste:
        left(angle)
        carre(d)
```

Pour dessiner n carrés, le processus est donc : dessine un carré, dessine un carré, dessine un carré, etc. On peut résumer le processus par : *dessine un carré puis dessine $n - 1$ carrés*. Ainsi, la fonction qui dessine n carrés peut être la même que celle qui dessine 1 carré ou $n - 1$ carrés. C'est juste la valeur de son paramètre qui change. Après avoir dessiner un premier carré, elle s'appelle elle-même pour dessiner $n - 1$ carrés. C'est le principe de récursivité.

```
def dessine2(liste, i, angle):
    if i >= 0 and i < len(liste):
        left(angle)
        carre(liste[i])
        dessine2(liste, i+1, angle)
```

On écrit les instruction `lst = [100, 90, 80, 70, 60], a = 20`, puis `dessine1(lst, a)` pour obtenir un premier dessin ou `dessine2(lst, 0, a)` pour obtenir un dessin identique.

Considérons le code suivant où la variable `i` prend successivement les valeurs entières de 0 à 9 qui sont ensuite affichées :

```
for i in range(10):
    print(i)
```


Notons qu'à la fin de ce code, une variable `i` continue d'exister et a la valeur 9.

Considérons alors la fonction `affiche` définie ainsi :

```
def affiche(k):
    if k < 10:
        print(k)
        affiche(k+1)
```

Si nous écrivons l'instruction `affiche(0)`, la fonction `affiche` est appelée avec le paramètre 0 qui vérifie le test `k < 10`. Après l'affichage de 0, la fonction `affiche` est à nouveau appelée mais avec le paramètre 1, puis après l'affichage de 1, la fonction `affiche` est à nouveau appelée mais avec le paramètre 2, et ceci continue avec l'affichage de `k` et l'appel qui suit tant que `k < 10`.

Dans ce code, nous n'avons ni boucle de type `while` ou `for`, ni instruction d'affectation.

Dans les chapitres précédents, les programmes ont été écrits avec des boucles et des affectations. C'est un style de programmation qui est :

- impératif, les séquences d'instructions sont exécutées l'une après l'autre ;
- itératif, des instructions sont exécutées dans des boucles `while` ou `for`.

Comme cela est visible sur les exemples précédents, une fonction qui s'appelle elle-même permet de remplacer une boucle. On parle alors de programme récursif par opposition à itératif.

Considérons la fonction `rebours1` qui prend en argument un entier naturel `n` et effectue un compte à rebours, par exemple affiche 5, 4, 3, 2, 1, 0.

```
def rebours1(n):
    """n est un entier naturel
       affiche les entiers de n à 0"""
    while n >= 0:
        print(n)
        n = n - 1
```

Cette fonction peut s'écrire avec une boucle `for`, mais quel que soit le type de boucle, la fonction `print` est appelée avec un paramètre prenant successivement la valeur `n`, `n-1`, ..., 0.

Sur le modèle de la fonction `affiche`, nous pouvons écrire une fonction `rebours` qui produit le même résultat que la fonction `rebours1`.

```
def rebours(n):
    if n >= 0:
        print(n)
        rebours(n-1)
```

Quel est le déroulement de l'exécution avec par exemple le paramètre `n` prenant la valeur 3 ?

Si le paramètre de la fonction `rebours` est positif, le test est vérifié et l'exécution produit :

- affichage de 3, puis appel `rebours(2)`,
- > affichage de 2, puis appel `rebours(1)`,
- - > affichage de 1, puis appel `rebours(0)`
- - - > affichage de 0, puis appel `rebours(-1)`,

Le nombre `-1` est strictement négatif, donc la condition `n >= 0` n'est plus vérifiée et l'exécution du programme est terminée. La même fonction `rebours` a été appelée cinq fois, et cette fonction a appelé la fonction `print` quatre fois pour les affichages des nombres de 3 jusqu'à 0.

■ Fonction récursive

Définitions

- Une fonction est dite *récursive* si elle s'appelle elle-même.

Nous parlons alors d'*appel récursif*. La fonction `rebours` est récursive.

- Par opposition, la fonction `rebours1` est *itérative*.

Voici deux autres exemples avec le module `Turtle` :

```
from turtle import *

def dessine(n):
    if n > 0:
        forward(n)
        right(90)
        dessine(n-5)
```

Si `n` est un entier strictement positif, la tortue trace un segment de longueur `n` pixels puis tourne à droite de 90 degrés. Dans l'appel récursif le nouveau paramètre est `n-5`, donc la tortue trace un segment de longueur `n-5` pixels puis tourne à droite de 90 degrés, et ainsi de suite. Ce processus continue tant que `n > 0`. Tester par exemple avec l'instruction `dessine(200)`.

Le deuxième exemple produit la courbe du dragon (ou « fractale du dragon » ou « courbe de Heighway » ou « dragon de Heighway ») qui a été étudiée pour la première fois par les physiciens de la NASA John Heighway, Bruce Banks, et William Harter.

```
def dragon(n, s):
    if n == 0:
        forward(5)
    else:
        dragon(n-1, 1)
        right(s * 90)
        dragon(n-1, -1)

speed('fastest')
ht()
dragon(10, 1)
```

Deux exemples avec la fonction `print` :

```
T = 8 # exécuter les fonctions avec pour paramètre n = T

def affiche1(n):
    if n > 0:
        print(n*' ' + '/' + 2*(T-n)*" " + '\\')
        affiche1(n-1)

def affiche2(n):
    if n > 0:
        print(n*' ' + '/' + 2*(T-n)*" " + '\\')
        affiche2(n-1)
    print(n*' ' + '\\ ' + 2*(T-n)*" " + '/')
```

Un exemple de calcul avec la multiplication de deux entiers naturels qui est effectuée uniquement à l'aide d'additions :

```
def produit(n, p):
    """n et p sont deux entiers naturels
    renvoie le produit de n par p"""
    if p == 0:
        return 0
    else:
        return n + produit(n, p-1)
```

Examinons l'exécution du programme pour obtenir `produit(4, 3)` :

le calcul de `4 + produit(4, 2)` est en attente du résultat de `produit(4, 2)` ;
- > le calcul de `4 + produit(4, 1)` est en attente du résultat de `produit(4, 1)` ;
- - - > le calcul de `4 + produit(4, 0)` est en attente du résultat de `produit(4, 0)` ;
- - - - - > la valeur de `produit(4, 0)` est obtenue, c'est 0 ;
- - - > l'addition `4 + produit(4, 0)` est effectuée, le résultat est 4, $(4 + 0)$;
- > l'addition `4 + produit(4, 1)` est effectuée, le résultat est 8, $(4 + 4)$;
l'addition `4 + produit(4, 2)` est effectuée, le résultat est 12, $(4 + 8)$;
la valeur renvoyée est 12 (le produit de 4 par 3).

Des visualisations de codes sont proposées sur le site Python Tutor <http://pythontutor.com/>.

Deux remarques importantes déduites du programme précédent doivent être généralisées.

- Le test `p == 0` est une condition d'arrêt. Il peut y avoir plusieurs conditions. Au moins une condition d'arrêt est obligatoire afin que le programme ne boucle pas indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs constituent une suite de nombres qui décroît vers la valeur d'arrêt. Dans l'exemple, la valeur `n - 1` passée en paramètre permet de faire décroître la valeur du paramètre. Pour que le programme ne boucle pas indéfiniment, il est impératif que le ou les paramètres atteignent une valeur d'arrêt après un nombre fini d'appels.

Principes généraux

- Une fonction récursive doit contenir une ou des conditions d'arrêt. Sinon le programme boucle indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs doivent être différentes. Sinon la fonction s'exécute à chaque appel de manière identique et continue donc de s'exécuter indéfiniment.
- Après un nombre fini d'appels, la ou les valeurs passées en paramètres doivent permettre de satisfaire la condition d'arrêt.

Dans le corps d'une boucle **while**, des instructions d'affectation permettent de modifier les valeurs des variables afin d'assurer l'arrêt de la boucle. Dans une fonction récursive, c'est un changement des valeurs des paramètres utilisés dans l'appel récursif qui assure l'arrêt des appels.

Un point essentiel est à noter concernant l'exécution des fonction **rebours** et **produit**. L'exemple de la fonction **rebours** nous montre que le code :

```
Si condition:
    appel récursif
```

s'exécute de la même manière que le code :

```
Tant que condition:
    instructions
```

Mais avec la fonction **produit**, des calculs ne peuvent pas être effectués avant d'obtenir les valeurs renvoyées par les appels récursifs et la mémoire de la machine est donc sollicitée pour stocker des instructions en attente. On parle dans ce cas de récursivité profonde. Dans le cas de la fonction **rebours**, on parle de récursivité terminale (ce qui est semblable à une boucle **while**).

Dans le code de la fonction **rebours**, l'affichage avec la fonction **print** est suivi de l'appel récursif qui est la dernière instruction à être exécutée. Si nous permutons ces deux lignes de code, nous obtenons la fonction **compte** écrite ci-dessous. Son comportement est similaire à celui de la fonction **produit**.

```
def compte(n):
    if n >= 0:
        compte(n-1)
        print(n)
```

■ Algorithmes dichotomiques

Recherche dichotomique

On cherche une valeur dans un tableau trié. Le tableau est coupé en deux et on cherche la valeur dans l'une des deux parties. La récursivité semble donc bien adaptée pour écrire un algorithme.

Le programme qui suit est une traduction en récursif du programme itératif présenté dans le chapitre précédent. Son coût est du même ordre que celui de l'algorithme itératif. La terminaison se démontre comme pour l'algorithme itératif en étudiant les valeurs successives des expressions **d-g**, **d** et **g** étant ici des paramètres de la fonction récursive.

La liste passée en paramètre est supposée non vide.

```
def dichorec(liste, x, g, d):
    if g == d:
        return liste[g] == x
    m = (g + d) // 2
    if x == liste[m]:
        return True
    elif x < liste[m]:
        if g == m:
            return False
        else:
            return dichorec(liste, x, g, m-1)
    else:
        return dichorec(liste, x, m+1, d)
```

On utilise la fonction avec un premier appel `dichorec(lst, x, 0, len(lst)-1)`. À chaque appel récursif, l'un des deux paramètres `g` ou `d` est modifié.

Une explication de ce programme, en particulier l'intérêt d'utiliser des indices `m-1` et `m+1`, est détaillée en exercice.

D'autres algorithmes de type dichotomique sont proposés en exercices, en particulier le calcul des puissances d'un nombre effectué en itératif au chapitre 4.

■ Exemples classiques

Les deux exemples qui suivent sont issus des mathématiques.

■ Premier exemple

L'un des exemples les plus classiques est une fonction mathématique, la fonction factorielle. On dit « factorielle n » et on note avec un point d'exclamation $n!$.

Cette fonction est définie « par récurrence » sur les entiers naturels : $0! = 1$ et pour tout entier naturel n non nul, $n! = n \times (n-1)!$

Autrement dit, pour tout n non nul, $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$.

Dans l'écriture $0! = 1$ et $n! = n \times (n-1)!$ pour n non nul, nous voyons les deux aspects d'une fonction récursive : un cas simple qui correspond à la condition d'arrêt et un cas complexe, celui de l'appel récursif pour obtenir la valeur de $(n-1)!$ afin de calculer $n!$.

L'écriture d'une fonction récursive consiste à traduire la définition mathématique.

```
def factorielle(n):
    """ n est de type int positif
        renvoie n! """
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1
```


La condition $n > 0$ au début de l'exécution de la fonction sert à déterminer s'il faut procéder à un appel récursif ou pas. Dès que la valeur de n est négative ou nulle, la valeur 1 est renvoyée. À cet effet, les valeurs passées en paramètres lors des appels récursifs constituent une suite décroissante qui en n étapes prend une valeur nulle si n est un entier positif.

Voici deux exemples en Python de fonctions qui calculent $n!$ de manière itérative.

- ▶ Avec une boucle `for` où les produits sont calculés dans l'ordre $1 \times 2 \times 3 \times \dots \times n$.
- ▶ Avec une boucle `while`, où les produits sont calculés dans l'ordre $n \times (n-1) \times (n-2) \times \dots \times 1$.

```
def factorielle2(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f

def factorielle3(n):
    f = 1
    while n > 1:
        f = f * n
        n = n - 1
    return f
```

Il est important de savoir passer d'une fonction récursive à une fonction itérative utilisant une boucle `while` ou une boucle `for`, et réciproquement.

Dans la fonction récursive, la dernière instruction est une multiplication par n qui ne peut être effectuée qu'après avoir obtenu la valeur renvoyée par l'appel récursif. Les différentes multiplications sont donc mises en attente comme les additions avec la fonction `produit` (page 90).

La fonction peut être modifiée en utilisant un paramètre supplémentaire, un accumulateur, par lequel des informations sont passées dans les appels récursifs. On munit ainsi la fonction d'un peu de mémoire. C'est l'accumulateur qui est renvoyé lorsque la condition d'arrêt des appels récursifs est satisfaite. Il doit donc contenir le résultat.

```
def factorielle4(n, acc):
    if n > 0:
        return factorielle4(n-1, acc*n)
    else:
        return acc
```

Cette fonction renvoie la valeur de $acc \times n!$. Donc `factorielle4(n, 1)` a pour valeur $n!$. Elle peut s'interpréter plus facilement comme une fonction itérative avec une boucle `while`. Il n'y a aucune opération en attente hormis les appels récursifs.

Terminaison et correction

De manière générale, la terminaison d'une fonction récursive est assurée par la condition d'arrêt des appels récursifs et les valeurs passées en paramètres constituant une suite qui converge en un

nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. La correction se démontre à l'aide d'un raisonnement « par récurrence ». On vérifie qu'une propriété est vraie pour une valeur entière initiale puis on prouve que la propriété est héréditaire. Une propriété est héréditaire si la véracité pour un entier n quelconque entraîne la véracité pour l'entier suivant $n + 1$.

Prouvons que `factorielle4(n, acc)` vaut $\text{acc} \times n!$ pour tout entier n .

► Initialisation

Pour $n = 0$, `factorielle4(0, acc)` vaut `acc`, soit $\text{acc} \times 0!$. La propriété est donc vraie pour $n = 0$.

► Hérédité

Nous supposons que la propriété est vraie pour un entier n quelconque, c'est-à-dire que `factorielle4(n, acc)` vaut $\text{acc} \times n!$.

Alors `factorielle4(n+1, acc)` vaut `factorielle4(n, acc*(n+1))`. D'après l'hypothèse, cette valeur est $\text{acc} \times (n+1) \times n!$. Donc `factorielle4(n+1, acc)` vaut $\text{acc} \times (n+1)!$ et la propriété est vraie pour $n + 1$.

La propriété est donc démontrée pour tout entier n .

■ Deuxième exemple

Dans les exemples précédents, le corps de la fonction récursive ne contient qu'un seul appel récursif. Mais il y a des cas où le corps de la fonction contient deux appels récursifs ou plus.

Considérons la suite de Fibonacci. Il s'agit d'une suite d'entiers naturels définie par récurrence.

Les deux premiers termes sont 0 et 1, puis un terme est la somme des deux termes précédents.

On obtient ainsi la suite de nombres 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

La définition mathématique est : $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$.

Le n -ième terme peut se calculer avec $n-2$ additions. On peut donc écrire une fonction, utilisant une boucle `while` ou une boucle `for`, qui prend en paramètre un entier `n` et renvoie le terme f_n . Exemple avec une boucle `for` :

```
def fibo1(n):
    u, v = 0, 1
    for i in range(n):
        u, v = v, u + v
    return u
```

Pour écrire une version récursive, il suffit de « traduire » la formule de calcul.

```
def fibo2(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo2(n-1) + fibo2(n-2)
```

Cette version récursive est clairement moins efficace que la version itérative. En effet, le nombre d'appels récursifs est « presque multiplié par deux » chaque fois qu'on augmente `n` d'une unité.

Il suffit de constater que $2^{30} \simeq 10^9$, pour conclure qu'avec des valeurs supérieures à 30, le calcul devient lent, et sûrement trop long pour des valeurs supérieures à 40.

Il est intéressant d'effectuer des tests dans l'interpréteur Python avec différentes valeurs à partir de 35. On peut arrêter l'exécution avec « ctrl+C » si l'attente est trop longue!

Validité de l'algorithme

Prouvons que la valeur de `fibo2(n)` est bien celle du nombre de Fibonacci f_n pour tout $n \in \mathbb{N}$.

► Initialisation

Pour $n = 0$, `fibo2(0)` vaut 0, et pour $n = 1$, `fibo2(1)` vaut 1. La propriété est donc vraie pour $n = 0$ et pour $n = 1$.

► Hérité

Supposons que pour un entier k quelconque `fibo2(k)` vaut f_k et `fibo2(k+1)` vaut f_{k+1} , alors : `fibo2(k+2)` vaut `fibo2(k+1) + fibo2(k)`, soit $f_{k+1} + f_k$ égal à f_{k+2} . La propriété est donc vraie pour $k + 2$.

Nous pouvons conclure que la propriété est vraie pour tout entier naturel n .

On peut transformer cette fonction récursive comme cela a été fait pour `factorielle`. Il convient ici d'ajouter deux accumulateurs dans les paramètres. Avec des valeurs par défaut, l'appel initial de la fonction est plus simple. On obtient alors, par exemple, le nombre f_{300} sans souci.

```
def fibo3(n, a=0, b=1):
    if n == 0:
        return a
    else:
        return fibo3(n-1, b, a+b)
```

On peut aussi écrire une version itérative qui construit un tableau des nombres f_n .

```
def fibo4(n): #tab est une liste ou un dictionnaire
    if n == 0:
        return [0] #ou return {0: 0}
    else:
        tab = [0 for i in range(n+1)] # ou tab = {i: 0 for i in range(n+1)}
        tab[1] = 1
        for i in range(2, n+1):
            tab[i] = tab[i-1] + tab[i-2]
        return tab
```

■ Coût d'un algorithme récursif

Le coût en temps d'un algorithme récursif est lié au nombre d'appels récursifs en fonction de n représentant le nombre ou la taille de l'objet en entrée. Ce coût peut généralement s'exprimer par une relation de récurrence et il n'est pas souvent facile à obtenir de manière exacte.

Prenons l'exemple de la fonction `factorielle`. Nous notons c_n le coût en fonction de n et nous comptons le nombre de tests et d'opérations arithmétiques.

Pour $n = 0$, nous avons un test, soit $c_0 = 1$.

Pour $n > 0$, nous avons un test, une multiplication et l'appel de la fonction effectué avec le paramètre $n - 1$, soit $c_n = 2 + c_{n-1}$ (ou $c_n = 3 + c_{n-1}$ si nous comptons aussi l'appel).

Les nombres c_n pour n entier naturel constituent ce qu'on appelle en mathématiques une suite arithmétique : 1, 3, 5, ... On montre que si $u_n = u_{n-1} + r$, alors $u_n = r \times n + u_0$. Nous obtenons donc $c_n = 2n + 1$.

Le coût de la fonction factorielle est linéaire en n .

Plus précisément, pour tout n , nous avons n appels récurifs, n multiplications et $n + 1$ tests.

De manière générale, si le programme a un coût constant c_0 lorsque la condition d'arrêt est satisfaite et si c_n est de la forme $c_{n-1} + k$ où k est le coût constant des opérations effectuées en dehors de l'appel récursif, alors on obtient pour tout n , $c_n = kn + c_0$, donc un coût linéaire.

Le coût de la fonction `fib2` pour le calcul des nombres de Fibonacci est plus complexe.

Les calculs du nombre d'additions, du nombre de tests et du nombre d'appels récurifs sont faits séparément.

Soit a_n le nombre d'additions pour obtenir f_n , alors $a_n = a_{n-1} + 1 + a_{n-2}$. On ajoute 1 des deux côtés de l'égalité : $a_n + 1 = a_{n-1} + 1 + a_{n-2} + 1$.

On pose alors $b_n = a_n + 1$, et on obtient : $b_n = b_{n-1} + b_{n-2}$, avec $b_0 = a_0 + 1 = 1$ et $b_1 = a_1 + 1 = 1$.

Nous constatons alors que $b_n = f_{n+1}$ et donc $a_n = f_{n+1} - 1$. Par exemple, pour calculer f_5 le nombre d'additions à effectuer est $f_6 - 1 = 7$.

Pour une addition, nous avons deux appels récurifs. Le nombre d'appels récurifs pour obtenir f_n est donc $2f_{n+1} - 2$. Par exemple pour calculer f_5 , nous appelons la fonction `fib2` qui procède à 14 appels récurifs. Le nombre de tests est égal au nombre d'appels de la fonction `fib2`, soit $2f_{n+1} - 1$. Nous comptons un test pour l'appel initial et un test pour chaque appel récursif. Pour calculer f_5 , nous procédons donc à 15 tests.

En conclusion, le coût du calcul de f_n en terme de nombre d'additions, de tests et d'appels récursif est $5f_{n+1} - 4$. Or, on démontre en mathématiques que le nombre f_n est égal à l'entier le plus proche de $\frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n$. Nous disons que le coût en fonction de n est exponentiel.

Lorsqu'un calcul exact du coût n'est pas possible, il faut envisager d'établir un encadrement.

■ Lien avec les piles

Il existe deux types de structures linéaires courantes en informatique, les files et les piles. Pour parler de la structure de file, on peut penser à une file d'attente comme dans une boulangerie : premier entré, premier sorti ! Pour une pile, on peut penser à une pile de livres déposés sur une table. On peut prendre un livre sur la pile et le déposer ailleurs, par exemple sur une autre pile. Les livres ne sont manipulés qu'un par un : dernier entré, premier sorti.

On considère à nouveau la fonction récursive `factorielle`. Pour son exécution, les instructions en attente sont placées dans une pile. Exemple avec `factorielle(3)` :

- `n` vaut 3 : empilement de `return n * factorielle(n-1)`;
- `n` vaut 2 : empilement de `return n * factorielle(n-1)`;
- `n` vaut 1 : empilement de `return n * factorielle(n-1)`.

Ensuite `n` vaut 0 donc renvoi de `factorielle(0)` : `return 1`.

Les instructions sont alors retirées de la pile et les produits sont effectués.

- n vaut 1 : return 1;
- n vaut 2 : return 2;
- n vaut 3 : return 6.

Les produits sont effectués ainsi : $3 \times (2 \times (1 \times 1))$.

En *notation polonaise inverse*, l'expression s'écrit : 3 2 1 1 × × × et se lit de gauche à droite. Cela revient à empiler tous les opérandes. Puis à chaque signe d'opération, on dépile deux opérandes, on effectue l'opération et on empile le résultat. À la fin, on dépile le résultat.

En Python le nombre d'appels récursifs en attente est limité. Par défaut, cette limite est de l'ordre du millier d'appels. S'il y a un dépassement, un message d'erreur est affiché, et le programme ne termine pas : *"RuntimeError : maximum recursion depth exceeded in comparison"*. Nous pouvons modifier le nombre maximal d'appels récursifs avec les instructions : `import sys`, puis `sys.setrecursionlimit(5000)` (par exemple). Ce nombre maximal doit rester raisonnable.

La limite du nombre d'appels est observable en exécutant une fonction comme `limite` :

```
def limite(n):
    if n > 0:
        limite(n-1)
    else:
        print("terminé")
```

On peut se demander pourquoi avec la fonction `fibonacci` qui nécessite un grand nombre d'appels récursifs par exemple pour calculer f_{30} , le programme a un temps d'exécution relativement long mais ne provoque pas d'erreur. Examinons le cas de `fibonacci(4)`, soit le calcul de f_4 . Les instructions en attente sont empilées, mais il y a au maximum 3 niveaux d'instructions en attente.

Chaque « demande » est empilée :

```
f4 demande f3 + f2
- - > f3 demande f2 + f1
- - - - > f2 demande f1 + f0
- - - - - > calcul de f2
- - - > calcul de f3
- - > f2 demande f1 + f0
- - > calcul de f2
calcul de f4
```

■ Permutations et parties d'un ensemble fini

■ Génération des permutations d'un ensemble fini

```
def permute(liste, ind, resultats):
    """Les éléments d'indice 0 à ind-1 sont fixés, les autres sont permutés,
    resultats contient les différentes permutations"""
    if ind == len(liste)-1: #cas liste vide
        resultats.append(list(liste))
```

```

    else:
        for i in range(ind, len(liste)):
            liste[i], liste[ind] = liste[ind], liste[i]
            permute(liste, ind+1, resultats)
            liste[i], liste[ind] = liste[ind], liste[i]

def permutations(L):
    res = []
    permute(L, 0, res)
    return res

```

On peut tester dans l'interpréteur Python avec `ensemble = ['a', 'b', 'c']`.

```

>>> permutations(ensemble)
[['a', 'b', 'c'], ['a', 'c', 'b'], ['b', 'a', 'c'], ['b', 'c', 'a'],
 ['c', 'b', 'a'], ['c', 'a', 'b']]

```

■ Génération des parties d'un ensemble fini

On distingue un élément, par exemple le dernier. Une partie contient ou pas cet élément.

```

def parties(liste, resultats):
    """énumère les parties de liste"""
    n = len(liste)
    if n == 0:
        resultats.append([])
    else:
        sousliste = liste[0:n-1]
        parties(sousliste, resultats) #sans le dernier élément
        for k in range(len(resultats)): #avec le dernier élément
            partie = resultats[k] + [liste[n-1]]
            resultats.append(partie)

def ens_parties(ensemble):
    rep = []
    parties(ensemble, rep)
    return rep

```

On teste la fonction avec `ensemble = [1, 2, 3, 4]` :

```

>>> ens_parties(ensemble)
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3], [4], [1, 4], [2, 4],
 [1, 2, 4], [3, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]

```

■ ■ Vrai/Faux

	Vrai	Faux
1. La récursivité peut s'utiliser pour remplacer une boucle <code>while</code> mais pas pour remplacer une boucle <code>for</code> .	☐	☐
2. On peut parler d'un programme récursif par opposition à un programme itératif.	☐	☐
3. Les valeurs des paramètres d'une fonction récursive doivent être identiques à chaque appel pour assurer la terminaison.	☐	☐
4. Dans une fonction récursive, il peut y avoir plusieurs conditions d'arrêt.	☐	☐
5. Les valeurs passées en paramètres pour les appels récursifs ne doivent pas obligatoirement constituer une suite décroissante.	☐	☐
6. Pour calculer la factorielle d'un nombre, une fonction récursive nécessite moins de calculs qu'une fonction itérative.	☐	☐
7. Si le code d'une fonction récursive contient deux appels récursifs alors la complexité de la fonction est toujours quadratique.	☐	☐
8. On utilise généralement une file pour placer des appels récursifs successifs en attente.	☐	☐
9. L'appel d'une fonction récursive pour calculer le n -ième nombre de Fibonacci nécessite deux appels récursifs donc le nombre total d'appels en fonction de n est de l'ordre de $2n$.	☐	☐
10. Si le nombre d'appels récursifs en attente est limité à 1000, on ne peut pas calculer le 40-ième nombre de Fibonacci avec une fonction récursive.	☐	☐

■ ■ Énoncé des exercices

Exercice 5.1 : Les seules opérations autorisées sur les nombres entiers sont soit ajouter 1, soit retrancher 1. La fonction `somme` renvoie la somme des entiers positifs `a` et `b`.

```
def somme(a, b):  
    while b > 0:  
        a = a + 1  
        b = b - 1  
    return a
```

Écrire une version récursive de cette fonction.

Exercice 5.2 : En s'inspirant de l'exercice précédent, il est demandé d'écrire une fonction récursive qui renvoie la somme de deux entiers quelconques. Les seules opérations autorisées sur les nombres entiers sont ajouter ou retrancher 1.

Exercice 5.3 : La fonction qui suit permet de savoir si un entier naturel est pair ou non.

```
def pair(n):  
    while n > 0:  
        n = n - 2  
    return n == 0
```

Écrire une version récursive de cette fonction.

Exercice 5.4 : On écrit en récursif une fonction du chapitre 4 qui calcule x^n .

Écrire une fonction récursive `puissance` qui prend en paramètres un flottant x non nul et un entier naturel n et qui renvoie x^n en se basant sur la définition mathématique : $x^0 = 1$ et $x^n = x \times x^{n-1}$ pour $n > 0$.

Exercice 5.5 : Calcul de x^n

1. Écrire, comme à l'exercice précédent, une fonction récursive `puissance1` prenant en paramètres un flottant x non nul et un entier naturel n et qui renvoie x^n . Mais, comme au chapitre 4, appliquer la remarque suivante :

$x^0 = 1$, si $n = 2k$ alors $x^n = x^{2k} = (x^k)^2$ et si $n = 2k + 1$ alors $x^n = x^{2k+1} = x(x^k)^2$.

2. Même question pour une fonction `puissance2`, mais en utilisant la remarque : $x^0 = 1$, si $n = 2k$ alors $x^n = x^{2k} = (x^2)^k$ et si $n = 2k + 1$ alors $x^n = x^{2k+1} = x(x^2)^k$.

Exercice 5.6 :

1. Écrire une fonction récursive `somme` qui prend en paramètre une liste de nombres et renvoie la somme des termes de cette liste.

2. Que penser du coût en temps et en espace d'une telle fonction ?
3. Expliquer avec l'exemple `somme([4, 7, 2])` le déroulement de l'exécution de la fonction.

Exercice 5.7* :

1. Expliquer quel est le résultat renvoyé par la fonction `mystere`.

```
def mystere(n):
    if n < 2:
        return str(n)
    else:
        return mystere(n//2) + str(n%2)
```

2. Écrire une fonction récursive `binaire` qui prend en paramètres un entier relatif `r` et un entier naturel `n` strictement positif, et qui renvoie la représentation en machine de `r` sur `n` bits. La méthode utilisée est celle du complément à deux et il est demandé d'écrire les 0 au début afin d'obtenir un résultat composé de `n` caractères (voir chapitre 11). Pour ce dernier point, on peut utiliser un compteur passé en paramètre.

Exercice 5.8 : La valeur du pgcd (plus grand commun diviseur) de deux entiers naturels a et b est calculable à l'aide de l'algorithme d'Euclide : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste dans la division euclidienne de a par b . En voici une version itérative :

```
def pgcd(a, b):
    if b == 0:
        return a
    while b != 0:
        a, b = b, a % b
    return a
```

Écrire une version récursive de cette fonction.

Exercice 5.9* : Une fonction `nettoie` prend en paramètre une liste triée et élimine les éléments identiques. Si la liste est `liste=[1, 1, 2, 6, 6, 6, 8, 8, 9]`, alors cette liste a pour valeur `[1, 2, 6, 8, 9]` après l'instruction `nettoie(liste)`. Écrire une version récursive.

```
def nettoie(L):
    n = len(L)
    k = 0
    while k < n - 1:
        if L[k] != L[k+1]:
            k = k + 1
        else:
            del L[k] # instruction del (delete): supprime l'élément L[k]
            n = len(L)
```

Exercice 5.10 : Utilisation du module Turtle

Tester le programme ci-dessous puis en écrire une version récursive.

```
from turtle import *

couleurs = ['blue', 'green', 'yellow', 'orange', 'red', 'purple']
bgcolor('black')

def dessin():
    for i in range(180):
        color(couleurs[i%6])
        forward(i)
        right(59)

dessin()
```

Exercice 5.11 : On considère la fonction récursive `dicho_rec` définie dans le cours.

1. On définit une liste par : `liste = [2, 5, 7, 8, 10, 13, 15, 16, 17, 20, 23, 24]`.

On écrit `dicho_rec(10, liste, 0, len(liste)-1)`.

Détailler l'exécution de la fonction en précisant les valeurs prises successivement par les trois variables `g`, `d` et `m`.

2. On conserve la même liste et on écrit `dicho_rec(18, liste, 0, len(liste)-1)`. Expliquer avec précision l'exécution de la fonction.

Exercice 5.12 : La fonction `palindrome` est écrite ci-dessous de manière itérative avec une boucle `for` et une interruption de boucle. Écrire une version récursive.

```
def palindrome(ch):
    n = len(ch)
    for i in range(n//2):
        if ch[i] != ch[n-1-i]:
            return False
    return True
```

■ ■ Indications

Ex. 5.2

Pour effectuer la somme $a + b$, distinguer les cas $b > 0$, $b < 0$ et $b = 0$.

Ex. 5.9

La fonction récursive prend deux paramètres, la liste et un indice k qui a pour valeur initiale par défaut 0.

■ ■ Corrigé du vrai/faux

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

Quelques explications

5. Ce qui est obligatoire est que les valeurs convergent en un nombre fini d'étapes vers des valeurs satisfaisant les conditions d'arrêt.
7. La complexité peut être exponentielle comme pour le calcul des nombres de Fibonacci.
8. On utilise une pile : dernier entré, premier sorti.
10. Le nombre d'appels est très important, donc il faudra du temps de calcul, mais peu d'appels sont en attente d'un résultat.

❑ Erreurs classiques et conseils.—

- Il ne faut pas opposer une boucle `while` et une boucle `for`. On peut passer de l'une à l'autre et elles constituent un programme itératif.
- Le code d'une fonction récursif contient au moins un appel récursif et une condition d'arrêt des appels.
- Il faut faire attention dans l'écriture d'une fonction récursive à préciser soigneusement les conditions d'arrêt qui assurent la terminaison de l'algorithme.

■ ■ Corrigé des exercices

Exercice 5.1

Deux versions : dans la première, un seul paramètre est modifié, dans la seconde les deux. Dans la première version, les additions (+ 1) sont en attente des retours des appels récursifs.

```
def somme_rec(a, b):
    if b == 0:
        return a
    else:
        return somme_rec(a, b-1) + 1

def somme_rec(a, b):
    if b == 0:
        return a
    else:
        return somme_rec(a+1, b-1)
```

Le nombre d'appels récursifs est égal à b . Il est donc important de permuter a et b si nécessaire afin que b soit le plus petit.

Exercice 5.2

Deux versions comme dans l'exercice précédent :

```
def somme_rec1(a, b):
    if b == 0:
        return a
    elif b > 0:
        return somme_rec1(a, b-1) + 1
    else:
        return somme_rec1(a, b+1) - 1

def somme_rec2(a, b):
    if b == 0:
        return a
    elif b > 0:
        return somme_rec2(a+1, b-1)
    else:
        return somme_rec2(a-1, b+1)
```

Comme dans l'exercice 1, le nombre d'appels récursifs est égal à b . Il est donc important de permuter a et b si nécessaire afin que b soit le plus petit en valeur absolue.

Exercice 5.3

```
def pair(n):
    if n > 0:
        return pair(n-2)
    else:
        return n == 0
```

Exercice 5.4

Une fonction puissance récursive :

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n-1)
```

On peut visualiser les appels récursifs et les résultats renvoyés :

```
def puissance(x, n, ch=""):
    if n == 0:
        return 1
    else:
        ch = ch + "--"
        param = str(x) + ", " + str(n-1)
        print(ch + "> appel de puissance(" + param + ")")
        y = puissance(x, n-1, ch)
        print(ch + "> résultat de puissance(" + param + "): " + str(y))
        return x * y
```

```
>>> puissance(2, 4)
--> appel de puissance(2, 3)
----> appel de puissance(2, 2)
-----> appel de puissance(2, 1)
-----> appel de puissance(2, 0)
-----> résultat de puissance(2, 0): 1
-----> résultat de puissance(2, 1): 2
----> résultat de puissance(2, 2): 4
--> résultat de puissance(2, 3): 8
16
```


Exercice 5.5

1. Une fonction `puissance1` :

```
def puissance1(x, n):
    if n == 0:
        return 1
    else:
        p = puissance1(x, n//2)
        if n % 2 == 0:
            return p * p
        else:
            return x * p * p
```

2. Une fonction `puissance2` :

```
def puissance2(x, n):
    if n == 0:
        return 1
    else:
        if n % 2 == 0:
            return puissance2(x**2, n//2)
        else:
            return x * puissance2(x**2, n//2)
```

Exercice 5.6

1. Deux exemples d'écriture d'une fonction `somme`.

```
def somme1(liste):
    if liste == []:
        return 0
    else:
        return liste[0] + somme1(liste[1:])

def somme2(liste):
    if liste == []:
        return 0
    else:
        x = liste.pop() # la liste est modifiée et vide à la fin !
        return somme2(liste) + x
```

2. Avec la première fonction, une liste est construite à chaque appel récursif. Le coût est quadratique en la longueur de la liste. Avec la deuxième fonction, le coût de la fonction `pop` est constant et la liste est modifiée en place. Le coût total est donc linéaire en la longueur de la liste mais la liste est vide après l'exécution de la fonction.

3. Exécution de la première fonction avec le paramètre [4, 7, 2] :
- > en attente, doit renvoyer : 4 + somme([7, 2])
 - - > en attente, doit renvoyer : 4 + (7 + somme([2]))
 - - - > en attente, doit renvoyer : 4 + (7 + (2 + somme([])))
 - - - - > arrêt des appels récursifs
 - - - - > renvoi de somme([]) : 4 + (7 + (2 + 0))
 - - - - > renvoi de (2 + 0) : 4 + (7 + 2)
 - - > renvoi de (7 + 2) : 4 + 9
 - > renvoi de 13

Exercice 5.7

- On reconnaît les divisions euclidiennes successives par 2. À chaque étape, on garde le reste et on recommence avec le quotient. Cette fonction renvoie l'écriture binaire de l'entier naturel n sous la forme d'une chaîne de caractères.
- Pour les entiers relatifs, on complète simplement avec l'ajout de 2^n à r si r est négatif et on ajoute un compteur en paramètre qui compte le nombre de caractères écrits.

```
def binaire(r, n, cpt=0):
    if r < 0:
        r = r + 2 ** n
    if r < 2:
        return (n-cpt-1)*'0' + str(r)
    else:
        return binaire(r//2, n, cpt+1) + str(r%2)
```

Exercice 5.8

```
def pgcd(a, b):
    if b == 0:
        return a
    else:
        return pgcd(b, a%b)
```

Exercice 5.9

```
def nettoie_rec(liste, k=0):
    if k < len(liste) - 1:
        if liste[k] != liste[k+1]:
            nettoie_rec(liste, k+1)
        else:
            del liste[k]
            nettoie_rec(liste, k)
```

Exercice 5.10

La fonction récursive prend un paramètre *i* prenant les valeurs de 0 à 180.

```
def dessin(i):
    if i < 180:
        color(couleurs[i%6])
        forward(i)
        right(59)
        dessin(i+1)

dessin(0)
```

Exercice 5.11

1. On examine les valeurs successives des variables.

- Nous avons *g* et *d* qui valent respectivement 0 et 11. Donc *m* prend la valeur 5 et *liste*[5] vaut 13. Puisque $10 < 13$, on procède à l'appel `mystere(10, liste, 0, 4)`.
- Nous avons *g* et *d* qui valent respectivement 0 et 4. Donc *m* prend la valeur 2 et *liste*[2] vaut 7. Puisque $7 < 10$, on procède à l'appel `mystere(10, liste, 3, 5)`.
- Nous avons *g* et *d* qui valent respectivement 3 et 5. Donc *m* prend la valeur 4 et *liste*[4] vaut 10. Donc la fonction renvoie l'indice 4.

2. On examine encore les valeurs successives des variables.

- Nous avons *g* et *d* qui valent respectivement 0 et 11. Donc *m* prend la valeur 5 et *liste*[5] vaut 13. Puisque $18 > 13$, on procède à l'appel `mystere(18, liste, 6, 11)`.
- Nous avons *g* et *d* qui valent respectivement 6 et 11. Donc *m* prend la valeur 8 et *liste*[8] vaut 17. Puisque $18 > 17$, on procède à l'appel `mystere(18, liste, 9, 11)`.
- Nous avons *g* et *d* qui valent respectivement 9 et 11. Donc *m* prend la valeur 10 et *liste*[10] vaut 23. Puisque $18 < 23$, on procède à l'appel `mystere(18, liste, 9, 10)`.
- Nous avons *g* et *d* qui valent respectivement 9 et 10. Donc *m* prend la valeur 9 et *liste*[9] vaut 20. Puisque $18 < 20$, et le test `g == m` est vérifié, la fonction renvoie `False`.

Exercice 5.12

```
def palindrome(ch):
    """ ch est de type str
        renvoie True si ch est un palindrome, False sinon """
    n = len(ch)
    if n <= 1:
        return True
    else:
        return ch[0] == ch[n-1] and palindrome(ch[1:n-1])
```


Algorithmes gloutons

UN SCIENTIFIQUE



Après des études à l'université de Chicago, **Joseph Kruskal** (1928-2010) entreprend des recherches à l'université de Princeton. Le mathématicien Paul Erdős lui soumet un problème sur les ensembles ordonnés, thème sur lequel il obtient son doctorat en 1954. Ceci le conduit à des études sur la théorie des graphes et à élaborer des algorithmes qui s'en inspirent. C'est ainsi qu'il conçoit en 1956 un algorithme d'arbre recouvrant de poids minimum qui est un bel exemple d'algorithme glouton.

■ Un peu d'histoire

Au XVI^e siècle, le savant flamand Gerhard Mercator a conçu très habilement des cartes en projection dite conforme, c'est-à-dire qu'elle conserve les angles des courbes. Aussi, il est facile de parcourir les mers avec ces cartes, en gardant un cap constant avec le nord, grâce à une boussole. Cependant, cette méthode n'est optimale qu'au niveau local mais ne l'est plus du tout pour traverser l'Atlantique.

Au début de l'informatique, on a conçu des algorithmes optimaux, examinant tous les cas de manière systématique. Malgré la puissance des machines récentes, mais à cause des masses énormes de données à traiter, certains d'entre eux se sont avérés trop lents. On préfère parfois des méthodes plus rapides, qui sautent des étapes ou même qui utilisent des démarches aléatoires, mais qui permettent d'aboutir à des résultats rapides et tout à fait satisfaisants.

Les algorithmes sont regroupés selon le type de problème qu'ils aident à résoudre ou selon les méthodes employées. Les algorithmes gloutons suivent une stratégie simple : lorsqu'à chaque étape, un choix doit être fait, c'est le choix optimal à ce moment qui est fait.

OBJECTIFS

- Comprendre le principe d'un algorithme glouton :
 - ▶ distinguer un choix optimum local d'un choix optimum global ;
 - ▶ reconnaître une stratégie gloutonne dans un programme ;
 - ▶ être capable d'écrire un algorithme glouton pour résoudre un problème.
- Savoir utiliser un algorithme glouton dans des situations diverses :
 - ▶ être capable d'appliquer cette stratégie avec quelques problèmes classiques ;
 - ▶ connaître au moins un exemple où un algorithme glouton est optimal ;
 - ▶ connaître au moins un exemple où un algorithme glouton n'est pas optimal.

Un problème d'optimisation a deux caractéristiques : une fonction que l'on doit maximiser ou minimiser et une série de contraintes auxquelles il faut satisfaire. On peut essayer de résoudre ce type de problème en écrivant un algorithme qui énumère les possibilités de manière exhaustive afin de trouver la meilleure. C'est un algorithme très simple mais souvent inutilisable en machine à cause de son coût. L'objectif d'un algorithme glouton (en anglais « greedy algorithm ») est d'obtenir une solution rapidement. Mais celle-ci n'est pas toujours la solution optimale.

Un choix peut être globalement optimal, c'est le meilleur de tous, ou localement optimal, c'est le meilleur parmi un ensemble restreint de choix. À chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. Un choix glouton est donc un choix localement optimal. La question est de savoir si en faisant une série de choix localement optimaux, on fini par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.

■ Problème du rendu de monnaie

Pour simplifier nous supposons que nous n'avons que des pièces. Un système de pièces est alors un n -uplet $S = (p_0, p_1, \dots, p_{n-1})$, où p_i représente la valeur de la pièce d'indice i . Ces valeurs constituent une suite de nombres entiers strictement croissante avec $p_0 = 1$. Si $p_0 > 1$, alors certaines sommes ne peuvent pas être rendues.

Le problème du rendu de monnaie consiste à trouver une liste d'entiers positifs $[x_0, x_1, \dots, x_{n-1}]$ qui vérifie $x_0 p_0 + x_1 p_1 + \dots + x_{n-1} p_{n-1} = r$ où r est la somme à rendre en minimisant la somme $x_0 + x_1 + \dots + x_{n-1}$, c'est-à-dire le nombre de pièces utilisées.

La condition $x_0 p_0 + x_1 p_1 + \dots + x_{n-1} p_{n-1} = r$ est appelée contrainte.

Dans le système de la zone euro, par exemple, nous avons en centimes les pièces suivantes :

$S = (1, 2, 5, 10, 20, 50, 100, 200, 500)$ où 100, 200 et 500 représentent respectivement les pièces de 1, 2 et 5 euros.

Il y a de nombreuses manières de rendre huit centimes. Les pièces qui peuvent être utilisées sont les pièces de 1, 2 et 5 centimes. Nous n'écrivons que les triplets possibles : $[8, 0, 0]$, $[6, 1, 0]$, $[4, 2, 0]$, $[3, 0, 1]$, $[2, 3, 0]$, $[1, 1, 1]$ et $[0, 4, 0]$.

Un programme permet de tester de manière exhaustive tous les triplets :

```
p = (1, 2, 5)
for i in range(9):
    for j in range(5):
        for k in range(2):
            s = i * p[0] + j * p[1] + k * p[2]
            if s == 8:
                print([i, j, k])
```

Le triplet qui minimise le nombre de pièces est le triplet $[1, 1, 1]$ avec 3 pièces.

Avec le système donné en exemple, un algorithme glouton fournit la solution optimale. On cherche, par valeur décroissante en partant de la pièce qui a la plus forte valeur, la première pièce qui a une valeur inférieure ou égale à la somme à rendre r . On prend cette pièce, on retranche sa valeur v à r . On recommence en partant de la pièce prise en cherchant celle qui a une valeur inférieure ou égale à la nouvelle somme à rendre $r - v$. On prend cette pièce, et ainsi de suite jusqu'à arriver à une somme à rendre nulle.

Les entrées sont p pour le système de pièces et r pour la somme à rendre.

```
def monnaie(p, r):
    n = len(p)
    i = n - 1
    solution = n * [0]
    while r > 0:
        while p[i] > r:
            i = i - 1
        solution[i] = solution[i] + 1
        r = r - p[i]
    return solution
```

Test de la fonction :

```
>>> p = (1, 2, 5, 10, 20, 50, 100, 200, 500)
>>> monnaie(p, 8)
[1, 1, 1, 0, 0, 0, 0, 0, 0]
```

Dans le cas général, avec un système différent de celui montré en exemple, c'est un problème difficile à résoudre. Mais dans presque tous les systèmes de monnaie, l'algorithme glouton est optimal. Pour rendre la monnaie, on rend la pièce (ou le billet) de valeur maximale, et on continue tant qu'il reste quelque chose à rendre.

Le système monétaire utilisé en Angleterre jusque dans les années 1960 comportait une multitude de pièces : des pièces de 1 penny, 3 pence, 4 pence, 6 pence, 12 pence soit 1 shilling, etc. (Le mot penny est le singulier de pence.)

Pour rendre 8 pence par exemple, le choix glouton donne $[2, 0, 0, 1]$, soit une pièce de 6 pence et deux pièces d'un penny, alors que le choix optimal est $[0, 0, 2, 0]$, soit 2 pièces de 4 pence.

■ Plus court chemin dans le plan

Supposons que nous disposons de n points placés dans le plan muni d'un repère orthonormé. Ces points sont donnés par leurs coordonnées. Étant donné un point P quelconque du plan, la question est de trouver un chemin qui commence en P et qui passe par les n points sans jamais passer deux fois par le même point. Un chemin entre deux points est un segment. Quel est le plus court chemin ?

On pourrait envisager de tester tous les chemins possibles et de retenir le plus court. Ce type de méthode s'appelle méthode par « force brute ». Avec trois points A, B, C , nous avons six chemins

possibles : $P - A - B - C$, $P - A - C - B$, $P - B - A - C$, $P - B - C - A$, $P - C - A - B$, $P - C - B - A$. Avec 4 points, nous avons 4 fois plus de chemins, soit 24. Avec 10 points nous avons $10 \times 9 \times \dots \times 2 \times 1 = 3628800$ chemins possibles. Il faut écrire tous ces chemins et déterminer le plus court en faisant autant de tests que de chemins. Admettons que cela puisse se faire en une seconde. Avec 15 points, il y a 1307674368000 chemins possibles, soit environ 3×10^6 fois plus de chemins qu'avec 10 points. Il faudrait donc 3×10^6 secondes pour trouver le plus court, soit plus de 4 jours. Avec 16 points, il y a 16 fois plus de chemins et il faudrait donc plus de 64 jours, et ainsi de suite. On comprend bien que cette méthode n'est utilisable qu'avec un nombre restreint de points. Une méthode gloutonne par contre est envisageable.

Nous supposons que les n points appartiennent à un carré dont la longueur du côté est stockée dans une variable `dim`.

Nous disposons de la distance euclidienne dans le plan et commençons par définir une fonction qui renvoie la distance entre deux points. Un point est représenté par une liste contenant l'abscisse et l'ordonnée du point.

```
from math import sqrt

def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
```

Nous construisons un tableau contenant les distances entre les différents points considérés et entre chacun de ces points et le point P .

La fonction `distances` prend en paramètre la liste des n points et le point de départ P . Le tableau renvoyé `tab` est une liste de $n + 1$ listes de longueurs $n + 1$. L'élément `tab[i][j]` est la distance entre les points d'indices i et j , P ayant pour indice n . Ce tableau est bien sûr symétrique : la valeur de `tab[i][j]` est égale à celle de `tab[j][i]`.

```
def distances(pts, dep):
    n = len(pts)
    tab = [(n+1)*[0] for i in range(n+1)]
    for i in range(n):
        for j in range(i):
            tab[i][j] = distance(pts[i], pts[j])
            tab[j][i] = tab[i][j]
        tab[n][i] = distance(dep, pts[i])
        tab[i][n] = tab[n][i]
    return tab
```

Notons que la fonction calcule $n(n + 1)/2$ distances et a donc un coût qui est quadratique en fonction de n .

Nous disposons des distances entre deux points quelconques. Nous pouvons donc écrire une fonction `longueur` qui prend en paramètres un chemin et un tableau des distances, et qui renvoie la longueur du chemin (qui commence en P).


```
def longueur(chemin, dist):
    d = 0
    id_pt = len(dist) - 1
    for point in chemin:
        d = d + dist[id_pt][point]
        id_pt = point
    return d
```

Nous construisons un chemin en choisissant à chaque étape le point le plus proche de la position courante parmi les points disponibles (ceux par lesquels le chemin n'est pas encore passé). On dit aussi le « plus proche voisin ». Autrement dit nous appliquons une stratégie gloutonne.

Pour cela nous écrivons une fonction `indice` qui prend en paramètres la position courante, le tableau des distances et les indices des points disponibles. La fonction renvoie l'indice du point le plus proche.

Pour avoir la liste des points disponibles on pourrait commencer avec la liste de tous les points et supprimer à chaque étape le point choisi. Mais il est plus simple de construire une liste dont les n éléments sont égaux à `True`, l'indice d'un élément correspondant à l'indice d'un point, et de passer la valeur de l'élément d'indice i à `False` quand le point d'indice i est choisi.

```
def indice(position, dist, dispo):
    n = len(dist) - 1
    mini = 3 * dim # supérieur à la diagonale du carré
    for i in range(n):
        if dispo[i]:
            d = dist[position][i]
            if d < mini:
                mini = d
                ind = i
    return ind
```

Une dernière fonction prend en paramètre le tableau des distances et construit le chemin.

```
def plus_court(dist):
    n = len(dist) - 1
    chemin = []
    dispo = n * [True]
    position = n
    while len(chemin) < n:
        position = indice(position, dist, dispo)
        chemin.append(position)
        dispo[position] = False
    return chemin
```


Pour tester le programme, nous choisissons un nombre de points et la dimension du carré dans lequel ces points sont choisis au hasard. Une fonction `points` crée la liste des n points tous distincts.

```
from random import randint

nbpoints, dim = 10, 20

def points(n, c):
    liste = []
    while len(liste) < n:
        x = randint(-c, c)
        y = randint(-c, c)
        if [x, y] not in liste:
            liste.append([x, y])
    return liste
```

On crée le point de départ et nous pouvons représenter ces points sur un graphique afin de vérifier que le programme est correct.

```
depart = (randint(-dim, dim), randint(-dim, dim))

import matplotlib.pyplot as plt

pts = points(nbpoints, dim)
x = [u[0] for u in pts]
y = [u[1] for u in pts]
plt.plot(x, y, "x")
plt.plot(depart[0], depart[1], "+")
plt.show()
```

Le test final permet de déterminer le chemin, de l'afficher dans un graphique et d'afficher la longueur du chemin.

```
tableau = distances(pts, depart)
ch = plus_court(tableau)
print(longueur(ch, tableau))

xliste = [depart[0]] + [pts[k][0] for k in ch]
yliste = [depart[1]] + [pts[k][1] for k in ch]
plt.plot(x, y, "x") # affichage des points
plt.plot(depart[0], depart[1], "+")
plt.plot(xliste, yliste) # affichage du chemin
plt.show()
```

Il reste deux questions auxquelles il faut répondre : cet algorithme glouton nous permet-il d'obtenir le chemin le plus court et quel est son coût ?

La réponse est non à la première question. Prenons trois points $A(0; 0)$, $B(5, 1; 0, 2)$ et $C(10; 0)$. Soit P de coordonnées $(2, 6; 1)$. Avec l'algorithme du plus proche voisin nous obtenons le chemin $P - B - C - A$. La longueur de ce chemin est $PB + BC + CA \simeq 2,6 + 4,9 + 10 = 17,5$. Mais le plus court chemin est $P - A - B - C$ avec une longueur égale à $PA + AB + BC = 2,8 + 5,1 + 4,9 = 12,8$. En général l'algorithme du plus proche voisin fournit un chemin plus long que le plus court chemin et la différence augmente avec le nombre de points. Il est conseillé de faire un dessin.

Pour la deuxième question, nous évaluons le coût de chaque fonction.

La fonction `distances` construit un tableau de $(n + 1)^2$ valeurs et a un coût de l'ordre de n^2 . La fonction `indice` comporte une boucle et a un coût linéaire. La fonction `plus_court` comporte une boucle avec à chaque passage un appel à la fonction `indice`, donc son coût est quadratique. En conclusion, cet algorithme a un coût quadratique en fonction de n .

■ Problème du sac à dos

Nous considérons la variante « entière » du problème du sac à dos.

Nous sommes devant un ensemble de n objets. Chaque objet noté o_i a une valeur notée v_i et un poids noté p_i . Il s'agit d'emporter dans son sac à dos l'ensemble d'objets qui a la plus grande valeur sachant que le sac supporte un poids maximum P . Comment résoudre ce problème, quels objets doit-on prendre ?

Pour appliquer une stratégie gloutonne, nous devons définir ce que nous entendons par le meilleur choix à chaque étape.

Il y a trois manières ici de définir un meilleur choix. Parmi les objets qui n'ont pas encore été pris, soit on choisit un objet qui a la valeur maximale, soit un objet qui a le poids minimal, soit un objet qui a le rapport valeur/poids maximal.

Nous ne considérons que les objets ayant un poids $p_i \leq P$. L'algorithme glouton consiste, à chaque étape, à choisir parmi ces objets celui qui représente le choix optimal. Nous le notons $O1$, sa valeur $V1$ et son poids $P1$. Ensuite, nous recommençons parmi les objets de poids $p_i \leq P - P1$. Et ainsi de suite.

Cette variante est dite entière parce que chaque objet est pris ou pas. Il existe une version fractionnaire. Le principe de base est le même, mais cette fois il est possible de prendre des fractions d'objets. Un algorithme glouton donne une solution optimale à cette variante fractionnaire.

Prenons un exemple : le sac à dos peut contenir 15 Kgs. Les poids des objets sont en Kg, les valeurs en euro.

Objet	Valeur	Poids	Valeur/Poids
Objet 1	126	14	9
Objet 2	32	2	16
Objet 3	20	5	4
Objet 4	5	1	5
Objet 5	18	6	3
Objet 6	80	8	10

Un objet est représenté par une liste comme `['objet 1', 126, 14]`.

Nous définissons trois fonctions qui prennent en paramètre un objet et renvoient respectivement la valeur, l'inverse du poids et le rapport valeur/poids de l'objet.

Remarque : si le poids est minimal, son inverse est maximal.


```
def valeur(obj):
    return obj[1]

def poids(obj):
    return 1 / obj[2]

def rapport(obj):
    return obj[1] / obj[2]
```

Nous définissons ensuite une fonction `glouton` qui prend en paramètres une liste d'objets, un poids maximal (celui que peut supporter le sac à dos) et le type de choix utilisé (par valeur, par poids, ou par valeur/poids). La première chose à faire est de trier la liste suivant le type de choix utilisé par ordre décroissant. Nous utilisons la fonction `sorted`. Une variable réponse sert à stocker les objets choisis.

```
def glouton(liste, poids_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    reponse = []
```

Ensuite nous parcourons la liste triée et ajoutons les noms des objets un par un tant que le poids total ne dépasse pas le poids maximal. La valeur totale et le poids total sont stockés dans deux variables `valeur` et `poids`.

```
def glouton(liste, poids_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    reponse = []
    valeur = 0
    poids = 0
    i = 0
    while i < len(liste) and poids <= poids_max:
        nom, val, pds = copie[i]
        if poids + pds <= poids_max:
            reponse.append(nom)
            poids = poids + pds
            valeur = valeur + val
        i = i + 1
    return reponse, valeur
```

Il reste à définir la liste d'objets puis à exécuter la fonction `glouton` en précisant le type de choix utilisé.


```
objets = [['objet 1', 126, 14], ['objet 2', 32, 2],
          ['objet 3', 20, 5], ['objet 4', 5, 1],
          ['objet 5', 18, 6], ['objet 6', 80, 8]]

print(glouton(objets, 15, valeur))
print(glouton(objets, 15, poids))
print(glouton(objets, 15, rapport))
```

Nous exécutons le programme et obtenons des résultats différents suivant le type de choix.

Par valeur : (['objet 1', 'objet 4'], 131).

Par poids : (['objet 4', 'objet 2', 'objet 3', 'objet 5'], 75).

Par valeur/poids : (['objet 2', 'objet 6', 'objet 4'], 117).

Le critère « valeur » est le plus intéressant, puisque la valeur totale est 131. Mais est-ce une solution optimale ?

La réponse à cette question est non et une étude exhaustive nous montre que la solution optimale est (['objet 2', 'objet 3', 'objet 6'], 132).

Dans une version fractionnaire du problème où l'on peut choisir une fraction du poids maximal autorisé pour chaque objet (par exemple s'il s'agit de poudre ou de liquide), le meilleur choix est dicté par le critère valeur/poids. Une stratégie gloutonne ferait donc prendre en premier les 2 kilos d'objet 2, puis les 8 kilos d'objet 6 et 5 kilos d'objet 1. Dans ce cas, nous obtenons finalement une valeur totale de $32 + 80 + 45 = 157$.

■ Stations d'essence

Voici un problème pour lequel un algorithme glouton donne une solution optimale.

Un automobiliste part en vacances et doit parcourir un long trajet. Il prend la route avec le plein de carburant. Son véhicule peut parcourir une distance maximale d avec un plein. La route empruntée comporte n stations services, $s_0, s_1, \dots, s_{n-1}$, rangées dans l'ordre rencontré en suivant le parcours. La première est à une distance d_0 du départ, la deuxième est à une distance d_1 de la première, et ainsi de suite. Le point d'arrivée est à une distance d_n de la dernière station.

Une condition nécessaire et suffisante pour que l'automobiliste puisse effectuer le parcours est que la distance entre le point de départ et la première station, les distances entre deux stations consécutives et la distance entre la dernière station et le point d'arrivée soient toutes inférieures ou égales à d . Nous supposons que cette condition est remplie.

L'objectif de l'automobiliste est de s'arrêter pour faire le plein le moins souvent possible.

Un algorithme glouton consiste chaque fois que le plein est fait à parcourir le plus long trajet possible sans refaire le plein. Le plein est donc fait chaque fois à la dernière station avant de tomber en panne.

Le programme consiste en une fonction prenant deux paramètres : une liste de distances et la distance maximale qui peut être parcourue avec un plein. Les éléments de la liste sont la distance entre le point de départ et la première station, puis les distances entre deux stations consécutives et enfin la distance entre la dernière station et le point d'arrivée.

```
def glouton(distances, dmax):
    n = len(distances)
    d = dmax
    stations = []
    i = 0
    while i != n:
        while i < n and distances[i] <= d:
            d = d - distances[i]
            i = i + 1
        stations.append(i-1)
        d = dmax
    return stations
```

Pour tester le programme, on construit une liste de distances au hasard en modifiant la dernière valeur pour que le total des distances soit égal à la longueur du trajet. On précise la longueur totale du trajet et la distance qui peut être parcourue avec un plein.

```
from random import randint
trajet = 2300 # km
tab = [randint(25, 100)] # distance première station
while sum(tab) < trajet:
    tab.append(randint(25, 100)) # distances inter stations
tab[len(tab)-1] = trajet - (sum(tab) - tab[len(tab)-1]) # la dernière distance
res = 600 # 600 km avec le plein du réservoir

print(glouton(tab, res))
```

Sur un test, on obtient : [7, 17, 26, 35], ce qui correspond à trois arrêts pour faire le plein aux stations d'indices 7, 17, 26. La dernière valeur est le point d'arrivée.

Les distances parcourues sont :

```
>>> sum(tab[:8])
590
>>> sum(tab[8:18])
594
>>> sum(tab[18:27])
560
>>> sum(tab[27:])
556
```

Nous allons démontrer que la stratégie utilisée est optimale.

Pour cela, considérons une solution $s = [s_1, s_2, \dots, s_p]$ obtenue avec notre algorithme et une solution optimale $S = [S_1, S_2, \dots, S_q]$.

Les deux solutions représentent des suites de stations triées de celle la plus proche du point de départ à celle la plus éloignée.

Si S est optimale, alors $q \leq p$ et nous devons prouver que $p = q$.

Soit k le plus petit entier tel que $s_k \neq S_k$. Par définition de l'algorithme glouton, $s_k > S_k$ et de plus la solution S peut s'écrire $S = [s_1, s_2, \dots, s_{k-1}, S_k, S_{k+1}, \dots, S_q]$.

Montrons alors que la suite $S' = [s_1, s_2, \dots, s_{k-1}, s_k, S_{k+1}, \dots, S_q]$ est elle aussi une solution optimale.

S' et S ont la même taille k , donc il suffit de vérifier que S' est une solution, c'est-à-dire qu'il n'y a aucun risque de panne. Or, puisque s est une solution, $s_k - s_{k-1} \leq d$. (d est la distance maximale qui peut être parcourue avec un plein). S est aussi une solution, donc $S_{k+1} - S_k \leq d$, d'où $S_{k+1} - s_k < S_{k+1} - S_k \leq d$, car $s_k > S_k$.

Remarquons que $S_{k+1} < s_k$, signifie qu'on pourrait supprimer S_{k+1} de S' et dans ce cas, on obtiendrait une meilleure solution, ce qui est impossible.

On itère ensuite ce raisonnement jusqu'à obtenir une solution $s = [s_1, s_2, \dots, s_q]$ qui est une solution optimale. D'où $p = q$, et la démonstration est terminée.

■ ■ Vrai/Faux

	Vrai	Faux
1. Une succession de choix localement optimaux conduit à un choix globalement optimal.	☐	☐
2. Un algorithme glouton consiste à étudier tous les cas possibles.	☐	☐
3. Si on utilise un algorithme glouton pour résoudre un problème, c'est parce-qu'il n'existe pas d'autres types d'algorithmes pour le faire.	☐	☐
4. Un algorithme glouton donne une solution optimale plus rapidement qu'un algorithme par force brute.	☐	☐
5. Pour le problème du sac à dos, un algorithme glouton ne donne pas toujours une solution optimale.	☐	☐
6. Pour le problème du rendu de monnaie, un algorithme glouton fournit une solution optimale.	☐	☐
7. Avec un système de pièces (1, 3, 4), un algorithme glouton n'est pas optimal pour rendre la monnaie.	☐	☐
8. Avec un système de pièces (1, 5, 10), un algorithme glouton est optimal pour rendre la monnaie.	☐	☐
9. Un algorithme glouton permet de trouver le chemin le plus court passant par trois points non alignés du plan.	☐	☐
10. Un algorithme glouton permet de trouver le chemin le plus court passant par quatre points non alignés du plan.	☐	☐

■ ■ Énoncé des exercices

Exercice 6.1 : On doit rendre la monnaie avec les pièces dont on dispose dans la caisse. Quelle affirmation est exacte ?

1. Avec uniquement des pièces de (1, 4, 5, 10), un algorithme glouton est optimal.
2. Avec uniquement des pièces de (1, 3, 4, 10), un algorithme glouton est optimal.
3. Avec uniquement des pièces de (1, 3, 4, 5), un algorithme glouton est optimal.
4. Avec uniquement des pièces de (1, 3, 5, 10), un algorithme glouton est optimal.

Exercice 6.2 : La fonction `monnaie` du cours utilise une liste pour représenter les pièces rendues dans le problème du rendu de monnaie. Écrire une fonction qui utilise un dictionnaire dont les clés sont les valeurs des pièces rendues et les valeurs associées aux clés sont le nombre de pièces correspondantes. Tester la fonction pour rendre 48 euros avec le système (1, 2, 5, 10, 20, 50, 100).

Exercice 6.3 :** Rendu de monnaie

Reprendre la fonction `monnaie` donnée dans le cours et modifier la liste `solution` afin d'obtenir dans le résultat les valeurs des pièces rendues avec leur nombre. Le résultat est une liste de liste. Par exemple pour rendre 48 euros avec le système de pièces (1, 2, 5, 10, 20, 50, 100), on obtient le résultat `[[20, 2], [5, 1], [2, 1], [1, 1]]`.

Exercice 6.4 :** Rendu de monnaie

Reprendre le code de la fonction `monnaie` de l'exercice 2 utilisant un dictionnaire. Modifier le code afin que le résultat soit un dictionnaire dont chaque élément est un dictionnaire donnant la solution pour une valeur à rendre comprise entre 1 et `rmax` où `rmax` est un paramètre de la fonction qui remplace le paramètre `r`. Par exemple, pour une valeur de `rmax` égale à 10, on obtient :

```
{1: {1: 1}, 2: {2: 1}, 3: {2: 1, 1: 1}, 4: {2: 2}, 5: {5: 1}, 6: {5: 1, 1: 1},  
7: {5: 1, 2: 1}, 8: {5: 1, 2: 1, 1: 1}, 9: {5: 1, 2: 2}, 10: {10: 1}}
```

Exercice 6.5* : Rendu de monnaie

Le programme du cours est itératif. Écrire une version récursive.

Exercice 6.6 :** Couleur d'un point

On reprend le programme avec les principales fonctions du cours concernant la recherche du plus court chemin passant par des points donnés dans le plan.

1. Importer la fonction `randint` et la fonction `sqrt`. Créer une variable `nbpoints = 40` et une variable `dim = 20`, puis une liste de couleurs pour le graphique : `couleurs = ["r", "g", "b"]`. Modifier la fonction `points` du cours afin que les points dans la liste renvoyée par la fonction soient des listes de trois valeurs, les deux coordonnées et la couleur choisie au hasard.
2. Créer un nouveau point comme le point de départ du chemin dans le cours mais en lui ajoutant la couleur noire. Tracer tous les points dans un graphique.
3. Reprendre la fonction `distance` qui renvoie la distance entre deux points et procéder à la modification nécessaire. Reprendre les fonctions `distances` et `indice` telles quelles.
4. Écrire une fonction `proches_voisins` qui prend en paramètres le tableau des distances et un entier `k` et qui renvoie la liste des indices des `k` plus proches voisins. Il s'agit de s'inspirer grandement de la fonction `plus_court` du cours.

5. Terminer le programme afin que le nouveau point prenne la couleur dominante parmi ses quatre voisins les plus proches et afficher les points sur un graphique.

Exercice 6.7 : Problème du sac à dos**

Reprendre le code complet présenté dans le cours pour résoudre le problème du sac à dos. L'objectif est de transformer le programme itératif en un programme récursif. Pour cela on découpe la fonction `glouton` en deux parties. Le code de la première partie, qui permet de trier la liste des données suivant le critère choisi, est le suivant :

```
def solution(liste, poids_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    return glouton(copie, poids_max, [], 0, 0, 0)
```

Écrire la fonction `glouton` dont les paramètres gardent les mêmes noms que dans la version du cours et sont dans l'ordre `liste, poids_max, reponse, valeur, poids, i`.

Exercice 6.8* : Stations d'essence

Le programme du cours est itératif. Écrire une version récursive.

Exercice 6.9* : Programme télé

Une personne doit faire une critique des programmes diffusés à la télévision. Elle souhaite regarder le plus grand nombre de programmes possible dans une journée quel que soit le contenu. Chaque programme `pi`, (émission, film, série), est caractérisé par un intervalle `[di, fi]` où `di` est l'heure de début et `fi` l'heure de fin. La différence `fi-di` représente donc la durée du programme `pi`. Les intervalles de temps des différents programmes regardés doivent être disjoints.

Pour organiser le visionnage des programmes, elle utilise un algorithme glouton. Elle trie les programmes suivant un critère (durée, ou début, ou fin). Ensuite elle choisit le premier et supprime parmi les autres ceux qui ne sont pas compatibles avec ce choix. Elle recommence cette étape tant que la liste des programmes n'est pas vide.

Montrer que les algorithmes 1 et 2 qui suivent ne sont pas optimaux en donnant un contre-exemple. Prouver que l'algorithme 3 est optimal.

1. Algorithme 1 : elle trie les programmes par durées croissantes, choisit le plus court, puis le plus court parmi ceux qui restent et qui sont compatibles, et ainsi de suite.
2. Algorithme 2 : elle trie les programmes par heures de début croissantes, choisit le programme commençant le plus tôt, puis le plus tôt parmi ceux qui restent et qui sont compatibles, et ainsi de suite.
3. Algorithme 3 : elle trie les programmes par heures de fin croissantes, choisit le programme qui se termine le plus tôt, puis le programme qui se termine le plus tôt parmi ceux qui restent et qui sont compatibles, et ainsi de suite.

■ ■ Indications

Ex. 6.3

Pour un élément de la liste `solution` qui correspond à une pièce qui va être prise, utiliser une variable booléenne pour distinguer les cas où la pièce est déjà utilisée ou pas.

Ex. 6.6

Il faut avoir bien compris les fonctions définies dans la partie cours qui sont utilisées dans cet exercice.

Ex. 6.7

Attention, contrairement au problème du rendu de monnaie pour lequel on peut prendre plusieurs fois la même pièce, on ne peut prendre un objet qu'une seule fois. On crée une nouvelle liste pour chaque choix possible qui permet de mettre à jour la liste précédente.

Ex. 6.9

Reprendre la démonstration finale du cours.

■ ■ Corrigé du vrai/faux

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

Quelques explications

3. C'est plutôt principalement pour une question de rapidité.
4. La solution n'est pas toujours optimale.
6. Cela dépend du système de pièces utilisé.
7. Un algorithme glouton propose $4 + 1 + 1$ pour rendre la somme 6, la solution est $3 + 3$.

☐ Erreurs classiques et conseils.—

- Ne pas confondre un choix localement optimal et un choix globalement optimal.
- Un algorithme par force brute étudie tous les cas, donc trouve la solution optimale, cependant son coût en temps peut être rédhibitoire. Un algorithme glouton ne trouve pas la solution optimale dans tous les cas mais peut être exécuté en un temps raisonnable.
- Lorsque la solution donnée par un algorithme glouton n'est pas optimale, on parle d'*heuristique*. La solution non optimale trouvée peut quand même apporter satisfaction.

■ ■ Corrigé des exercices

Exercice 6.1

Pour 1 : $8 = 4 + 4$, alors qu'avec un algorithme glouton $8 = 5 + 1 + 1 + 1$. Pour 2 : $6 = 3 + 3$, alors qu'avec un algorithme glouton $6 = 4 + 1 + 1 + 1$. Pour 3 : $7 = 4 + 3$, alors qu'avec un algorithme glouton $7 = 5 + 1 + 1$. C'est l'affirmation 4 qui est exacte.

Exercice 6.2

La fonction `monnaie` du cours est modifiée.

```
def monnaie(p, r):
    n = len(p)
    i = n - 1
    solution = {}
    while r > 0:
        while p[i] > r:
            i = i - 1
        val = p[i]
        if val in solution:
            solution[val] = solution[val] + 1
        else:
            solution[val] = 1
        r = r - p[i]
    return solution
```

Test de la fonction :

```
>>> p = (1, 2, 5, 10, 20, 50, 100)
>>> monnaie(p, 48)
{20: 2, 5: 1, 2: 1, 1: 1}
```

Pour rendre 48 euros, on rend 2 billets de 20, une pièce de 5, une pièce de 2 et une pièce de 1.

Exercice 6.3

```
def monnaie(p, r):
    n = len(p)
    i = n - 1
    solution = []
    while r > 0:
        while p[i] > r:
            i = i - 1
```



```

    val = p[i]
    utilise = False
    for liste in solution:
        if liste[0] == val:
            utilise = True
            liste[1] = liste[1] + 1
            break
    if not utilise:
        solution.append([val, 1])
    r = r - p[i]
    return solution

```

Test avec un système de pièces :

```

>>> s = (1, 2, 5, 10, 20, 50, 100)
>>> monnaie(p, 48)
[[20, 2], [5, 1], [2, 1], [1, 1]]

```

On peut mesurer en comparant avec l'exercice 2, l'intérêt d'utiliser un dictionnaire plutôt qu'une liste pour simplifier le code.

Exercice 6.4

```

def monnaie(p, rmax):
    n = len(p)
    solutions = {}
    for r in range(1, rmax+1):
        cle = r
        i = n - 1
        solution = {}
        while r > 0:
            while p[i] > r:
                i = i - 1
            val = p[i]
            if val in solution:
                solution[val] = solution[val] + 1
            else:
                solution[val] = 1
            r = r - p[i]
        solutions[cle] = solution
    return solutions

p = (1, 2, 5, 10, 20, 50, 100)
print(monnaie(p, 10))

```

```
def monnaie_rec(p, r, i, sol=None):
    if sol == None:
        sol = len(p) * [0]
    if r == 0:
        return sol
    else:
        if p[i] > r:
            return monnaie_rec(p, r, i-1, sol)
        else:
            sol[i] = sol[i] + 1
            return monnaie_rec(p, r-p[i], i, sol)

p = (1, 2, 5, 10, 20, 50, 100, 200, 500)

print(monnaie_rec(p, 48, len(p)-1))
```

Exercise 6.6

1. Création des points.

```
from random import randint
from math import sqrt

nbpoints, dim = 40, 20
couleurs = ["r", "g", "b"]
def points(n, c):
    pts = []
    while len(pts) < n:
        x, y = randint(-c, c), randint(-c, c)
        cl = couleurs[randint(0, 2)]
        if (x, y, cl) not in pts:
            pts.append((x, y, cl))
    return pts
```

2. Tracer des points.

```
import matplotlib.pyplot as plt

nouveau = (randint(-dim, dim), randint(-dim, dim), "k")
pts = points(nbpoints, dim)
for u in pts:
    plt.plot(u[0], u[1], u[2] + "o")
```

```
plt.plot(nouveau[0], nouveau[1], nouveau[2] + "+")
plt.show()
```

3. Modification de la fonction distance.

```
def distance(p1, p2):
    x1, y1, c1 = p1
    x2, y2, c2 = p2
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def distances(pts, dep):
    n = len(pts)
    tab = [(n+1)*[0] for i in range(n+1)]
    for i in range(n):
        for j in range(i):
            tab[i][j] = distance(pts[i], pts[j])
            tab[j][i] = tab[i][j]
        tab[n][i] = distance(dep, pts[i])
        tab[i][n] = tab[n][i]
    return tab

def indice(position, dist, dispo):
    n = len(dist) - 1
    mini = 3 * dim
    for i in range(n):
        if dispo[i]:
            d = dist[position][i]
            if d < mini:
                mini = d
                ind = i
    return ind
```

4. Fonction proches_voisins.

```
def proches_voisins(dist, k):
    n = len(dist) - 1
    voisins = []
    dispo = n * [True]
    position = n
    while len(voisins) < k:
        position = indice(position, dist, dispo)
        voisins.append(position)
        dispo[position] = False
    return voisins
```


5. Dessin final.

```
tableau = distances(pts, nouveau)
k = 4
vs = [pts[i] for i in proches_voisins(tableau, k)]
# une liste de compteurs pour compter les couleurs
cpt = [0, 0, 0]
for v in vs:
    if v[2] == "r":
        cpt[0] = cpt[0] + 1
    elif v[2] == "g":
        cpt[1] = cpt[1] + 1
    else:
        cpt[2] = cpt[2] + 1
i = cpt.index(max(cpt)) # l'indice de la couleur la plus représentée
choix = couleurs[i] # la couleur la plus représentée

for u in pts:
    plt.plot(u[0], u[1], u[2] + "o")

plt.plot(nouveau[0], nouveau[1], choix + "+")
plt.show()
```

Exercice 6.7

```
def glouton(liste, poids_max, reponse, valeur, poids, i):
    if i < len(liste) and poids <= poids_max:
        nom, val, pds = liste[i]
        if poids + pds <= poids_max:
            reponse.append(nom)
            poids = poids + pds
            valeur = valeur + val
        return glouton(liste, poids_max, reponse, valeur, poids, i+1)
    else:
        return reponse, valeur
```

On obtient les mêmes solutions que dans le cours. Par exemple, avec `solution(objets, 15, poids)`, on obtient `(['objet 4', 'objet 2', 'objet 3', 'objet 5'], 75)`.

Exercice 6.8

```
def choix(distances, dmax, i, stations, d):
    n = len(distances)
    if i != n:
```

```

        if i < n and distances[i] <= d:
            choix(distances, dmax, i+1, stations, d-distances[i])
        else:
            stations.append(i-1)
            choix(distances, dmax, i, stations, dmax)

def stats(distances, dmax):
    stations = []
    choix(distances, dmax, 0, stations, dmax)
    return stations

```

Pour tester le programme, on reprend l'exemple du cours et on écrit `s = stats(tab, res)`.

Exercice 6.9

1. L'algorithme 1 n'est pas optimal : on choisit le programme le plus court, par exemple de 8h30 à 9h30, alors que 2 autres seraient possibles de 7h à 9h et de 9h15 à 11h.
2. L'algorithme 2 n'est pas optimal : on choisit le programme qui commence le plus tôt, par exemple de 6h à 8h30 alors que plusieurs seraient possibles : de 6h30 à 7h30, de 7h45 à 8h15, etc.
3. L'algorithme 3 est optimal : on peut le démontrer comme pour le problème des stations d'essence à la fin du cours. Soit f la date de fin la plus petite et une solution optimale $[f_1, f_2, \dots, f_k]$, avec $f_1 < f_2 < \dots < f_k$. On remplace f_1 par f ($f < f_1$) et alors $[f, f_2, \dots, f_k]$ est aussi optimale. On continue avec la date de fin la plus petite parmi les dates compatibles avec f qui remplace f_2 , et ainsi de suite.

Remarques

- Un algorithme glouton est optimal si l'on souhaite maximiser le nombre de programmes visionnés. On peut vérifier que ce n'est pas le cas si l'on souhaite maximiser la durée totale de visionnage.
- On peut écrire un programme semblable à celui qui est écrit pour le problème du sac à dos, chaque programme étant accompagné de son heure de début, son heure de fin et sa durée.

Manipulation d'une image

UN SCIENTIFIQUE



Le savant canadien **Willard Boyle** (1924-2011) effectue des études scientifiques couronnées par un doctorat obtenu à Montréal en 1950. Entré aux Laboratoires Bell, il conçoit en 1962 le premier laser à rubis à émission continue. Il participe à la recherche de sites d'atterrissage pour le programme Apollo. En 1969, avec l'Américain George Smith, il met au point le premier capteur photographique électronique. Cette découverte permettra aux deux savants d'obtenir le prix Nobel de physique en 2009.

■ Un peu d'histoire

Conçue et améliorée au cours du XIX^e siècle par Niepce, Daguerre et William Talbot, la photo argentique a connu ses heures de gloire jusque dans les années 1980.

En 1969, Willard Boyle et George Smith conçoivent le premier capteur photographique. Ce composant électronique est capable de transformer un rayonnement électromagnétique, par exemple la lumière visible, en un signal électrique. Grâce à une amplification du courant et une numérisation des données, ce procédé permet d'obtenir des photos numériques.

Compte tenu du nombre impressionnant de pixels que comporte une image de bonne qualité, des procédés de compression des images ont été mis au point pour permettre leur envoi ou leur stockage. Un algorithme de décodage est alors nécessaire ; le plus connu d'entre eux, connu sous le nom de JPEG, a été mis au point pour IBM par un groupe d'experts vers 1990.

OBJECTIFS

Les transformations d'images constituent un cadre dans lequel il est intéressant de manipuler des tableaux à deux dimensions et de pratiquer différentes méthodes précédemment rencontrées.

- Connaître quelques formats d'images :
 - ▶ distinguer une image matricielle d'une image vectorielle ;
 - ▶ utiliser simplement un module spécialisé pour récupérer une matrice de pixels ;
 - ▶ savoir créer une matrice sous la forme d'une liste de listes.
- Comprendre le principe d'algorithmes de transformations simples :
 - ▶ manipuler une matrice avec accès à différents éléments et modification ;
 - ▶ savoir écrire un algorithme d'une transformation simple.
- Pratiquer des modifications plus complexes :
 - ▶ programmer des modifications et transformations diverses ;
 - ▶ extraire des informations pour créer une nouvelle image ;
 - ▶ découvrir la notion de masque et la cryptographie.

■ Fichiers images

Si on regarde les propriétés d'un fichier avec l'extension jpg représentant une image, on obtient des informations sur l'emplacement, la taille, la date de création. Il y a aussi la possibilité d'avoir plus de détails pour un fichier d'une photo provenant d'un appareil comme un smartphone. Nous pouvons obtenir de nombreuses informations sur le fichier mais aussi sur l'image, la photo, l'appareil utilisé, le logiciel, la date de prise de vue, la position GPS, etc. Ces informations sont enregistrées dans le fichier avec l'image proprement dite.

■ Une image

On distingue parmi les images numériques, les images vectorielles et les images matricielles. Les images vectorielles sont créées par des équations mathématiques. Les images matricielles sont constituées d'un ensemble de points ou pixels. La suite concerne les images matricielles.

Une image est définie par son nombre de pixels. Par exemple une image contenant 1600 pixels en largeur et 1200 pixels en hauteur a une définition de 1600×1200 pixels soit 1920000 pixels (près de 2 mégapixels). La résolution est exprimée en points ou pixels par pouce (ppp) ou dot per inch en anglais (dpi). Un pouce vaut environ 2,54 cm. Ainsi, une résolution de 72 ppp en largeur et en hauteur signifie qu'un carré de côté 2,54 cm contient $72 \times 72 = 5184$ pixels.

Une image de longueur 1600 pixels et de résolution 300 ppp, qui est une résolution correcte pour une impression, a une longueur de 5,33 pouce soit 13,55 cm.

Dans le codage de couleur RGB (ou RVB en français pour rouge, vert, bleu), la couleur de chaque pixel est codée par trois octets. Chaque octet représente l'intensité d'une composante qui varie donc entre 0 et 255. Le triplet (255, 0, 0) code le rouge, le triplet (0, 0, 0) code le blanc, le triplet (255, 255, 255) code le noir, le triplet (120, 120, 120) code un niveau de gris, le triplet (255, 255, 0) code le jaune.

Le programme suivant aide à comprendre comment est composée une image. La fonction `pixel` permet de dessiner un pixel, en fait un petit carré. La fonction `dessine` dessine les pixels donnés dans une matrice.

```
from turtle import *

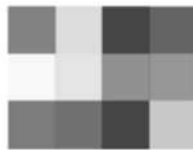
def pixel(coords, taille, couleur): # définition d'un pixel
    ht()
    up()
    goto(coords)
    down()
    color(couleur)
    begin_fill()
    for i in range(4): # un pixel est représenté par un carré
        forward(taille)
        right(90)
    end_fill()
```

```
def dessine(image, resolution, origine=(0,0)):
    speed('fastest')
    n = len(image)
    p = len(image[0])
    taille = 1 // resolution
    x_origine, y_origine = origine
    x0 = x_origine - p * taille // 2
    y0 = y_origine + n * taille // 2
    up(); goto((x0, y0)); down()
    for i in range(n):
        for j in range(p):
            coords = (x0 + j * taille, y0 - i * taille)
            pixel(coords, taille, image[i][j])

colormode(255)
image1 = [[(255, 0, 0), (0, 255, 0), (0, 0, 255), (100, 100, 100)],
          [(255, 255, 0), (0, 255, 255), (255, 0, 255), (150, 150, 150)],
          [(128, 128, 0), (0, 128, 128), (128, 0, 128), (200, 200, 200)]]

dessine(image1, 1/30, (0, 100)) # image1 est une matrice de pixels
```

Les « pixels » sont représentés ci-dessous en niveaux de gris.



La liste des pixels est longue à écrire. On peut créer une image (une matrice) aléatoire.

```
from random import randint
def alea(n, p): # largeur, hauteur
    img = [[(0, 0, 0) for j in range(n)] for i in range(p)]
    for i in range(p):
        for j in range(n):
            r, v, b = randint(0, 255), randint(0, 255), randint(0, 255)
            img[i][j] = (r, v, b)
    return img

image2 = alea(16, 12)
dessine(image2, 1/16, (-200, 0))
```

Le dessin prend du temps et le nombre de points est limité si on souhaite le voir s'afficher à l'écran. Ce n'est évidemment pas une bonne méthode pour construire des images.

■ Transformation d'une image

La modification d'une image consiste à modifier la matrice de pixels. En général cela s'effectue avec deux boucles imbriquées qui permettent d'avoir accès à chaque pixel. La complexité en temps est alors de l'ordre de $n \times p$ si n est le nombre de lignes et p le nombre de colonnes. On peut essayer autant que possible à limiter la complexité en espace.

Par exemple, on peut écrire une fonction *inverse* pour inverser les couleurs d'une image et obtenir un *négatif*. On commence par écrire une fonction *inversion* qui inverse la couleur d'un pixel. Puis on écrit la fonction *inverse* qui prend en paramètre une matrice de pixels représentant une image

```
def inversion(pixel):
    r, v, b = pixel
    return (255-r, 255-v, 255-b)

def inverse(image):
    n = len(image) # nombre de lignes
    p = len(image[0]) # nombre de colonnes
    inv = [[inversion(image[i][j]) for j in range(p)] for i in range(n)]
    return inv
```

On peut tester la fonction *inverse* avec en écrivant `image3 = inverse(image1)` où `image1` est définie plus haut, puis `dessine(image3, 1/30, (100, 0))`.

■ Images et matrices

Voici comment on utilise la bibliothèque PIL pour récupérer une matrice de pixels.

```
from PIL import Image

fichier = "image1.jpg" # fichier image enregistré dans le même dossier
img1 = Image.open(fichier)
print(img1.size, img1.mode, img1.format)

# copie d'une image pixel par pixel
largeur, hauteur = img1.size
img2 = Image.new("RGB", (largeur, hauteur)) # création d'une nouvelle image
for ha in range(hauteur):
    for la in range(largeur):
        pixel = img1.getpixel((la, ha)) # attention à l'ordre des indices
        r, g, b = pixel
        img2.putpixel((la, ha), (r, g, b))

img2.save("image2.jpg") # sauvegarde
img2.show() # affichage
```

Il est peut être nécessaire d'installer la bibliothèque Pillow (successeur du projet PIL, Python Imaging Library).

■ Fichiers PBM, PGM et PPM

Ce sont les formats de fichiers : portable bitmap, portable graymap et portable pixmap.

Écriture et lecture

Avec l'interpréteur Python, il est nécessaire de déterminer le répertoire de travail. Par défaut c'est le répertoire où est installé Python (par exemple "Python35"). Il est possible de changer ce répertoire de travail avec la fonction `chdir` (de l'anglais change directory) du module `os`, en donnant l'adresse complète :

```
>>>from os import chdir
>>>chdir('C:/Users/Toto/Informatique/Exercices')
```

C'est maintenant dans le répertoire `Exercices` que seront créés les fichiers. Attention, ce répertoire doit préalablement exister.

Par contre, à l'exécution d'un fichier Python, nommé par exemple `test.py`, le répertoire courant par défaut est celui où est enregistré ce fichier.

Écriture d'un fichier pbm

Il s'agit d'un fichier au format texte ASCII pour représenter une image en noir et blanc.

```
def ecrire_fichier_nb(mat, nom):
    hauteur = len(mat)
    largeur = len(mat[0])
    nom = nom + '.pbm'
    f = open(nom, 'w')
    f.write('P1\n' + str(largeur) + ' ' + str(hauteur) + '\n')
    for i in range(hauteur):
        for j in range(largeur):
            bit = str(mat[i][j])
            f.write(bit + ' ')
        f.write('\n')
    f.close()
```

L'écriture d'un fichier pgm (niveaux de gris) ou ppm (couleur) nécessite quelques adaptations. On peut aussi obtenir un fichier pbm, (pgm, ppm) à partir d'une image dans un format quelconque. On l'ouvre avec un logiciel comme GIMP et on exporte l'image en choisissant l'extension. Attention, pour chaque extension il existe un format texte ASCII et un format binaire. On trouve les informations nécessaires par exemple à l'adresse : https://fr.wikipedia.org/wiki/Portable_pixmap.

Lecture d'un fichier pbm

Il s'agit de lire un fichier texte et de récupérer les informations : le code P1 caractéristique d'un fichier pbm, la largeur, la hauteur et les valeurs des pixels 0 ou 1.

```
def lire_fichier_nb(nom):
    f = open(nom, 'r')
    magic = f.readline().rstrip()
    dim = f.readline()
    while dim[0] == "#":
        dim = f.readline()
    dim = dim.rstrip().split(" ")
    dim = [int(dim[0]), int(dim[1])]
    mat = [[0 for j in range(dim[0])] for i in range(dim[1])]
    tab = f.readlines()
    i, j = 0, 0
    for ligne in tab:
        if ligne[0] == "#":
            continue
        ligne = ligne.rstrip().split(" ")
        for n in ligne:
            mat[i][j] = int(n)
            j = (j+1) % dim[0]
            if j == 0:
                i = (i+1) % dim[1]
    f.close()
    return mat
```

■ Algorithmes

Pour tester les programmes on utilise une image au format pbm. Mais on peut les tester avec une image d'un format quelconque si on sait récupérer une matrice de pixels par exemple avec PIL.

Un exemple avec une inversion noir et blanc :

```
def inverse_nb(img):
    haut, larg = len(img), len(img[0])
    for i in range(haut):
        for j in range(larg):
            img[i][j] = 1 - img[i][j]
```



Le chat s'appelle Bruce. Son image est disponible sur demande (bruce.lechat@orange.fr).

On dispose d'une image `chat.pbm`. On écrit les instructions :

```
mat = lire_fichier_nb('chat.pbm')
inverse_nb(mat)
ecrire_fichier_nb(mat, 'inverse_chat')
```

Avec PIL et une image `jpg` par exemple, on peut procéder ainsi pour une inversion des couleurs :

```
from PIL import Image

def inverse_couleurs(img):
    largeur, hauteur = img.size
    inv = Image.new("RGB", (largeur, hauteur))
    for i in range(hauteur):
        for j in range(largeur):
            pixel = img.getpixel((j, i))
            r, v, b = pixel
            inv.putpixel((j, i), (255-r, 255-v, 255-b))
    return inv

fichier = "chat.jpg"
img1 = Image.open(fichier)
img2 = inverse_couleurs(img1)
img2.save("chat_inv.jpg")
```

Pour la suite, n est la hauteur de l'image considérée et p la largeur.

■ Symétrie

On envisage une symétrie axiale avec un axe vertical. Il convient donc d'échanger chaque pixel d'indice (i, j) avec le pixel d'indice $(i, p - 1 - j)$, i allant de 0 à $n - 1$ et j allant de 0 à $p/2$. Attention à la boucle interne, si on écrit `for j in range(larg)` les pixels sont échangés deux fois et on obtient l'image initiale. L'indice i est l'indice de la ligne.

```
def sym_vert(img):
    haut, larg = len(img), len(img[0])
    for i in range(haut):
        for j in range(larg//2):
            img[i][j], img[i][larg-1-j] = img[i][larg-1-j], img[i][j]
```

Pour une symétrie axiale avec un axe horizontal, il convient d'échanger chaque ligne d'indice i avec la ligne d'indice $n - 1 - i$ pour i allant de 0 à $n/2$. Ici aussi, attention à ne pas échanger deux fois les lignes avec une boucle `for i in range(haut)`.

```
def sym_hor(img):
    haut, larg = len(img), len(img[0])
    for i in range(haut//2):
        img[i], img[haut-1-i] = img[haut-1-i], img[i]
```



■ Rotation

Pour une image représentée par une matrice de pixel, une rotation présente de nombreuses difficultés. Celles-ci proviennent principalement du fait qu'un pixel a des coordonnées entières. La suite se limite donc à des images carrées ($n = p$) avec une rotation d'un quart de tour.

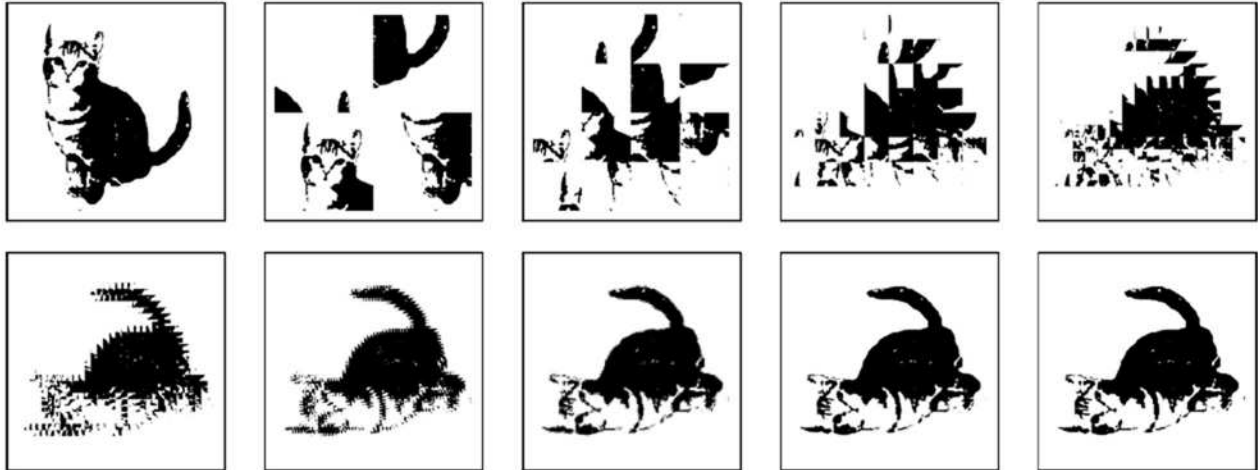
Une rotation d'un quart de tour consiste à permuter les pixels des quatre quadrants. On note p_1 le pixel d'indice (i, j) , p_2 le pixel d'indice $(n - 1 - j, i)$, p_3 le pixel d'indice $(n - 1 - i, n - 1 - j)$, p_4 le pixel d'indice $(j, n - 1 - i)$. On effectue pour i allant de 0 à $n/2$ et j allant de 0 à $n/2$, les permutations $(p_1, p_2, p_3, p_4) \mapsto (p_2, p_3, p_4, p_1)$.

Divers algorithmes ont été étudiés dans les années 1980 pour manipuler des images en particulier pour un affichage correct sur un écran, à l'époque en noir et blanc. Avec peu de mémoire disponible et une capacité de calcul bien moindre qu'aujourd'hui, différentes *primitives* (des procédures de base) ont été développées comme « déplacer un bloc de bits d'un endroit à un autre en mémoire ». Un algorithme classique comme la rotation d'un quart de tour utilisait ce déplacement. Une image carrée est découpée en quatre carrés de même taille. Chaque carré est déplacé, $(c_1, c_2, c_3, c_4) \mapsto (c_2, c_3, c_4, c_1)$ et on effectue la rotation d'un quart de tour de chaque carré par un appel récursif.

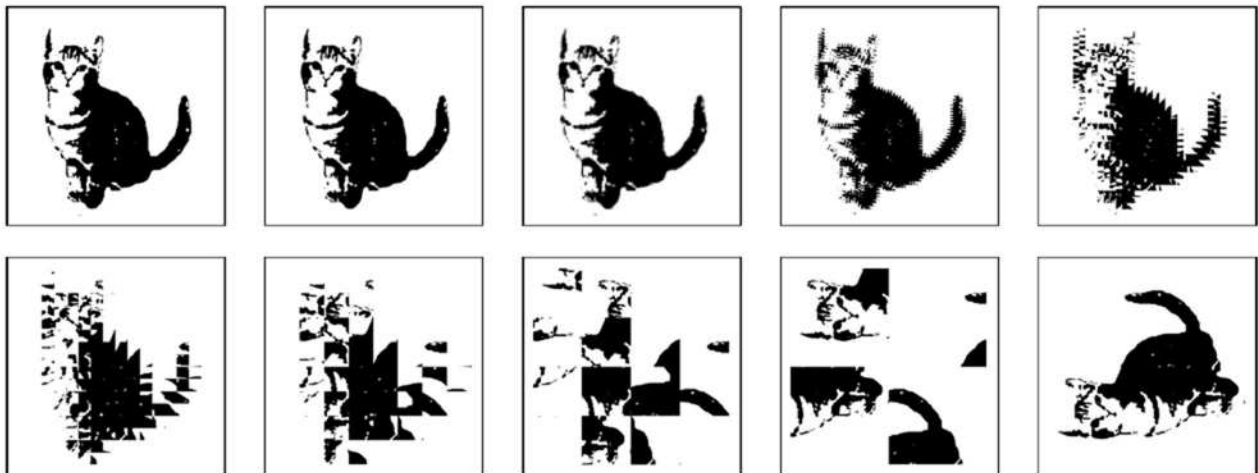
```
def rotation(img, x, y, n):
    if n > 1:
        n = n // 2
        for i in range(x, x+n):
            for j in range(y, y+n):
                temp = img[i][j]
                img[i][j] = img[i][n + j]
                img[i][n + j] = img[n + i][n + j]
                img[n + i][n + j] = img[n + i][j]
                img[n + i][j] = temp
        rotation(img, x, y, n)
        rotation(img, x, y+n, n)
        rotation(img, x+n, y, n)
        rotation(img, x+n, y+n, n)
```

On utilise la fonction `rotation` avec le fichier `chat.pbm`. La figure qui suit présente le résultat à chaque étape.

```
mat = lire_fichier_nb('chat.pbm')
rotation(mat, 0, 0, len(mat))
ecrire_fichier_nb(mat, 'rot_chat')
```



Si on permute les instructions en effectuant « découper en quatre, tourner, déplacer » à la place de « découper en quatre, déplacer, tourner » on obtient les résultats suivants :



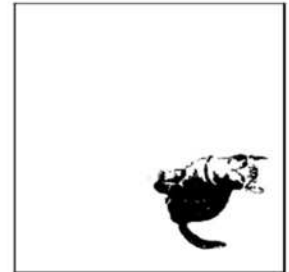
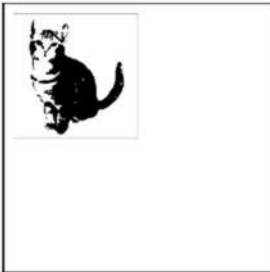
Une autre méthode permet d'effectuer cette rotation. C'est une méthode par cisaillement. L'image est placée dans un carré quatre fois plus grand en haut à gauche puis découpée en segments de pixels. À la première étape, on découpe des segments verticaux. Le premier segment ne bouge pas, le deuxième est descendu d'un pixel, le troisième de deux pixels et ainsi de suite jusqu'au dernier.

La deuxième étape consiste à découper des segments diagonaux en commençant au point extrême en haut à gauche. Les segments obtenus de haut en bas se placent respectivement de droite à gauche.

La troisième étape est similaire à la première. On découpe des segments verticaux. Le premier segment ne bouge pas, le deuxième est descendu d'un pixel, le troisième de deux pixels et ainsi de

suite jusqu'au dernier. On peut ajouter une quatrième étape pour ramener l'image dans le quadrant supérieur gauche.

```
def cisaille(m):
    haut, larg = dim(m)
    mat = [[0 for i in range(2*larg)] for j in range(2*haut)]
    for i in range(haut):
        for j in range(larg):
            mat[i+j][j] = m[i][j]
    # fin étape 1
    for i in range(haut):
        for j in range(i+1):
            mat[i][j+2*larg-1-i] = mat[i][j]
    for i in range(haut, 2*haut):
        for j in range(i-haut, larg):
            mat[i][j+larg-i+haut] = mat[i][j]
    fin étape 2
    mat2 = [[0 for i in range(2*larg)] for j in range(2*haut)]
    for i in range(haut, 2*haut):
        for j in range(larg, 2*larg):
            mat2[i][j] = mat[i+larg-j][j]
    # fin étape 3
    for i in range(haut):
        for j in range(larg):
            m[i][j] = mat2[i+haut][j+larg]
```



■ Réduction et agrandissement

On dispose d'une image de taille (n, p) et on souhaite la réduire. Nous supposons que la réduction consiste à diviser la longueur et la largeur par un nombre entier d . C'est le cas le plus simple et on envisage de garder une ligne sur d lignes et une colonnes sur d colonnes.

```
def reduction(img, d):
    n, p = len(img), len(img[0])
    mat1 = [[0 for j in range(p)] for i in range(n//d + 1)]
    for i in range(n):
        if i % d == 0:
            for j in range(p):
```

```

        mat1[i//d][j] = img[i][j]
mat2 = [[0 for j in range(p//d + 1)] for i in range(n//d + 1)]
for i in range(n//d + 1):
    for j in range(p):
        if j % d == 0:
            mat2[i][j//d] = mat1[i][j]
return mat2

```

La figure représente l'image intermédiaire avec $d = 3$.



Pour l'agrandissement par un facteur d entier, un procédé simple est de copier d fois chaque colonne puis d fois chaque ligne.

```

def agrandissement(img, d):
    n, p = len(img), len(img[0])
    mat1 = [[0 for j in range(d*p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            if img[i][j] == 1:
                for k in range(d):
                    mat1[i][d*j+k] = 1
    mat2 = [[0 for j in range(d*p)] for i in range(d*n)]
    for i in range(n):
        for j in range(d*p):
            if mat1[i][j] == 1:
                for k in range(d):
                    mat2[d*i+k][j] = 1
    return mat2

```

Tous les programmes précédents peuvent facilement être adaptés et utilisés sur des images en couleurs au format jpg ou png qui sont lues à l'aide de la bibliothèque PIL.

■ ■ Vrai/Faux

	Vrai	Faux
1. Si <code>m</code> est une liste de listes, on accède à l'élément d'indice <code>j</code> de la liste d'indice <code>i</code> par <code>m[i][j]</code> .	☐	☐
2. Après avoir écrit <code>m = [[1, 2], [3, 4]]</code> , puis <code>m[1] = m[0]</code> , puis <code>m[1][0] = 2</code> , <code>m</code> vaut <code>[[1, 2], [2, 2]]</code> .	☐	☐
3. Pour créer une liste <code>m</code> contenant trois listes de longueur trois, on écrit <code>m = [0 for i in range(3) for j in range(3)]</code> .	☐	☐
4. On écrit <code>m = [[0, 0, 0] for i in range(3)]</code> . Si on écrit l'instruction <code>m[0][0] = 1</code> , trois valeurs sont modifiées.	☐	☐
5. On écrit <code>m = 3 * [[0, 0, 0]]</code> . L'instruction <code>m[0][0] = 1</code> modifie trois valeurs.	☐	☐
6. Si on écrit l'instruction <code>m = [[0 for i in range(2)] for j in range(3)]</code> , <code>m</code> a pour valeur <code>[[0, 0, 0], [0, 0, 0]]</code> .	☐	☐
7. On écrit <code>m1 = [0, 1]</code> , puis <code>m2 = 2 * m1</code> et <code>m2.append(2)</code> . La valeur finale de <code>m1</code> est <code>[0, 1]</code> .	☐	☐
8. On écrit <code>m1 = [0, 1]</code> , puis <code>m2 = 2 * m1</code> et <code>m2[1] = 2</code> . La valeur finale de <code>m1</code> est <code>[0, 1]</code> .	☐	☐
9. Si on écrit <code>m1 = [0, 1]</code> , puis <code>m2 = 2 * [m1]</code> et enfin <code>m2[1].append(2)</code> , alors la valeur finale de <code>m1</code> est <code>[0, 1]</code> .	☐	☐
10. Si on écrit <code>m = 2 * []</code> puis <code>m.append(1)</code> , alors <code>m</code> a pour valeur <code>[1]</code> .	☐	☐

■ ■ Énoncé des exercices

Exercice 7.1 : Utilisation de la bibliothèque PIL

1. Écrire une fonction `gris1` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg ou png. La fonction crée une nouvelle image en niveaux de gris, la sauvegarde et l'affiche. S'inspirer du programme du cours pour copier une image. Pour calculer l'intensité de gris, on calcule la moyenne m des intensités de rouge, de vert et de bleu. Un pixel (r, v, b) devient un pixel (m, m, m) .
2. Écrire une fonction `gris2` semblable à la fonction `gris1`. La seule différence est la formule utilisée pour calculer le niveau de gris : remplacer la moyenne m par $0,2126 \times r + 0,7152 \times v + 0,0722 \times b$. Comparer les résultats obtenus avec les deux fonctions.

Exercice 7.2 : Image en noir et blanc avec PIL

Écrire une fonction `noir_blanc` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg ou png et un nombre entier naturel nommé `seuil`. La fonction crée une nouvelle image en noir et blanc, la sauvegarde et l'affiche. Pour cela reprendre la fonction `gris2` et si la valeur du niveaux de gris calculée est supérieure à la valeur de `seuil`, le pixel devient $(255, 255, 255)$, sinon $(0, 0, 0)$.

Tester avec un seuil égal à 128, puis avec différentes valeurs.

Exercice 7.3 : Permutation des couleurs avec PIL

Écrire une fonction `permute_couleurs` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg ou png. La fonction crée une nouvelle image en permutant les couleurs : un pixel (r, v, b) devient par exemple un pixel (v, b, r) .

Exercice 7.4* : Échanges de lignes

Écrire une fonction `echange_lignes` qui prend en paramètre une matrice représentant une image carrée de côtés n pixels, avec n pair, au format pbm. La fonction modifie la matrice en plusieurs étapes.

Première étape : la première ligne (indice 0) et la dernière (indice $n - 1$) sont fixes. Pour les autres lignes, dans chaque couple de lignes consécutives les deux lignes sont échangées, la ligne d'indice 1 avec celle d'indice 2, la ligne d'indice 3 avec celle d'indice 4, et ainsi de suite jusqu'à l'échange de la ligne d'indice $n - 3$ avec celle d'indice $n - 2$.

Deuxième étape : le même processus est reproduit sauf que les deux premières lignes et les deux dernières sont fixes.

On continue le processus d'échanges, en laissant fixes les trois premières et les trois dernières lignes, puis les quatre premières et les quatre dernières, etc. À la dernière étape, seule la ligne d'indice $n/2 - 1$ est échangée avec la ligne suivante.

Tester la fonction avec une matrice représentant une image. Exécuter à nouveau la fonction sur l'image obtenue et recommencer plusieurs fois en observant le résultat.

Exercice 7.5* : Transformation du photomaton

On considère une image carrée de côté n pixels. La transformation à appliquer pour créer une nouvelle image est décrite pour chaque pixel repéré par un couple d'indices (i, j) (les indices vont de 0 à $n - 1$) :

- si i et j sont pairs, le pixel est envoyé sur le pixel d'indice $(i/2, j/2)$;
- si i est impair et j pair, le pixel est envoyé sur le pixel d'indice $((n+i)/2, j/2)$;
- si i est pair j impair, le pixel est envoyé sur le pixel d'indice $(i/2, (n+j)/2)$;
- si i et j sont impairs, le pixel est envoyé sur le pixel d'indice $((n+i)/2, (n+j)/2)$.

Écrire une fonction `photomaton` qui prend en paramètre une matrice carrée et réalise cette transformation plusieurs fois (neuf fois par exemple).

Exercice 7.6* : Chiffrement d'une image bitmap.

Reprendre le programme de l'exercice 10 au chapitre 3 avec l'image `image1.pbm`.

1. Écrire une fonction `hasard` qui prend en paramètres les dimensions d'une image n et p , (n est la hauteur et p la largeur), et renvoie une matrice représentant une image aléatoire. Chaque pixel vaut 0 ou 1 de manière équiprobable.

Créer une image aléatoire avec les mêmes dimensions que l'image `image1.pbm`.

On nomme cette image `alea.pbm`. Visualiser cette image.

2. Écrire une fonction `chiffre` pour chiffrer l'image `image1.pbm`. La fonction prend en paramètres une image à chiffrer et une image clé sous la forme de matrices de mêmes dimensions. Elle crée une nouvelle image de même dimension. Pour déterminer chaque bit de l'image, on utilise l'opérateur *ou exclusif* entre les bits correspondant de l'image à chiffrer et de l'image clé. Le *ou exclusif* se note $\wedge$ en Python. Si $b1$ et $b2$ sont deux bits, $b1 \wedge b2$ vaut 0 si $b1$ et $b2$ sont égaux et vaut 1 sinon.

Exécuter la fonction avec l'image `image1.pbm` et l'image clé `alea.pbm`. Visualiser le résultat.

L'image clé s'appelle un *masque*. Comment déchiffrer l'image chiffrée?

3. Faire une copie de la fonction précédente, la renommer, et remplacer l'opérateur *ou exclusif* par l'opérateur *ou*. L'opérateur se note $|$, mais on peut utiliser `or`. Que peut-on dire de l'image obtenue si on exécute ce programme avec l'image chiffrée et l'image clé?

Exercice 7.7* : Reprendre le code permettant d'écrire un fichier `pbm`. L'objectif est de dessiner des anneaux concentriques limités par des cercles de rayons $R, R\sqrt{2}, R\sqrt{3}, \dots$. Chaque pixel d'une image carrée est codé de telle manière que le disque central soit blanc, le premier anneau noir, le deuxième anneau blanc, etc.

1. Le côté de l'image est $n = 257$ pixels. Les indices i et j (indice de la ligne et indice de la colonne) représentent les coordonnées d'un pixel. Écrire une fonction `matrice(n, r)` qui prend en argument la taille et le rayon r , calcul le quotient entier d^2/R^2 , où d est la distance entre le pixel de coordonnée (i, j) et le pixel central de l'image et affecte noir ou blanc au pixel. Les pixels sont codés en blanc si $d \leq R$, en noir si $R < d \leq R\sqrt{2}$, en blanc si $R\sqrt{2} < d \leq R\sqrt{3}$, en noir si $R\sqrt{3} < d \leq 2R$, ... Autrement dit, si $R\sqrt{n} < d \leq R\sqrt{n+1}$ avec n entier naturel, alors les pixels sont blancs si n est pair et noirs sinon.

2. Tester le programme et vérifier que l'image obtenue est composée de couronnes concentriques. Tester avec plusieurs valeurs de R , par exemple : $R = 20, R = 10, R = 5, R = \sqrt{5/3}, R = \sqrt{17/11}$; les couronnes concentriques ont disparu et laissent place à des figures géométriques.

Exercice 7.8 : Avec la bibliothèque PIL, on écrit une instruction `img = Image.open(fichier)`, où `fichier` est le nom d'un fichier image, pour récupérer les données de l'image. L'instruction `liste = list(img.getdata())` permet de stocker l'ensemble des pixels dans une liste. Il s'agit d'une simple liste qui contient à la suite la première ligne de l'image, la seconde, etc.

Écrire une fonction `matrice` qui prend en paramètre le nom d'un fichier et renvoie une matrice (liste de listes) représentant l'ensemble des pixels. La première sous-liste contient la première ligne de l'image, la deuxième sous-liste la deuxième ligne, et ainsi de suite.

■ ■ Indications

Ex. 7.3

Utiliser la fonction du cours pour copier une image avec PIL et traiter la permutation proposée par l'énoncé. Il est possible d'ajouter un deuxième paramètre, entier par exemple, qui permet de choisir une permutation parmi les cinq possibles et de compléter le code avec les quatre autres permutations possibles.

Ex. 7.4

La difficulté réside principalement dans la gestion des indices. L'échange de deux lignes est simple à écrire en Python.

Ex. 7.5

Écrire une fonction récursive qui prend en paramètres une matrice et un entier correspondant au nombre de répétitions. Pour une transformation, il suffit de traduire l'énoncé.

Ex. 7.7

Commencer par écrire une fonction qui calcule la distance entre deux points.

Ex. 7.8

C'est un simple calcul sur les indices.

■ ■ Corrigé du vrai/faux

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

Quelques explications

2. Avec `m[1] = m[0]`, `m[0]` et `m[1]` désignent le même objet.
4. Les trois sous-listes sont trois objets différents créés l'un après l'autre.
9. C'est le même objet qui est désigné par `m1`, `m2[0]` et `m2[1]`.

❑ Erreurs classiques et conseils.—

- La manipulation des listes et en particulier des listes de listes demande une grande attention.
- La notion d'objet mutable est importante. Le rôle des méthodes permettant la modification d'un élément ou l'ajout d'un élément doit être compris.
- Il faut bien maîtriser la notion d'affectation en Python. Les résultats obtenus dans diverses situations en dépendent.

■ ■ Corrigé des exercices

Exercice 7.1

1. Fonction gris1

```
from PIL import Image

def gris1(nom_image):
    img1 = Image.open(nom_image)
    largeur, hauteur = img1.size
    img1_gris1 = Image.new("RGB", (largeur, hauteur))
    for ha in range(hauteur):
        for la in range(largeur):
            pixel = img1.getpixel((la, ha))
            r, v, b = pixel
            gris = (r + v + b) // 3
            img1_gris1.putpixel((la, ha), (gris, gris, gris))
    img1_gris1.save("image1_gris1.jpg") # sauvegarde
    img1_gris1.show() # affichage
```

2. Fonction gris2

```
def gris2(nom_image):
    img1 = Image.open(nom_image)
    largeur, hauteur = img1.size
    img1_gris2 = Image.new("RGB", (largeur, hauteur))
    for ha in range(hauteur):
        for la in range(largeur):
            pixel = img1.getpixel((la, ha))
            r, v, b = pixel
            gris = int(0.2126 * r + 0.7152 * v + 0.0722 * b)
            img1_gris2.putpixel((la, ha), (gris, gris, gris))
    img1_gris2.save("image1_gris2.jpg")
    img1_gris2.show()
```

Avec la deuxième formule l'image obtenue est plus « lumineuse ».

Exercice 7.2

On modifie le code de la fonction gris2 :

```
def noir_blanc(nom_image, seuil):
    img1 = Image.open(nom_image)
```

```

largeur, hauteur = img1.size
img1_nb = Image.new("RGB", (largeur, hauteur))
for ha in range(hauteur):
    for la in range(largeur):
        pixel = img1.getpixel((la, ha))
        r, v, b = pixel
        gris = int(0.2126 * r + 0.7152 * v + 0.0722 * b)
        gris = 255 if (gris >= seuil) else 0
        img1_nb.putpixel((la, ha), (gris, gris, gris))
img1_nb.save("image1_nb.jpg")
img1_nb.show()

```

Exercice 7.3

Fonction `permute_couleurs`.

On choisit la permutation, ou on la propose en ajoutant un paramètre à la fonction.

```

def permute_couleurs(nom_image):
    img1 = Image.open(nom_image)
    largeur, hauteur = img1.size
    img1_perm = Image.new("RGB", (largeur, hauteur))
    for ha in range(hauteur):
        for la in range(largeur):
            pixel = img1.getpixel((la, ha))
            r, v, b = pixel
            img1_perm.putpixel((la, ha), (v, b, r))
    img1_perm.save("image1_perm.jpg")
    img1_perm.show()

```

Exercice 7.4

L'écriture de la fonction traduit le processus donné dans l'énoncé. On exécute plusieurs fois la fonction avec les matrices produites. On écrit avec la fonction du cours l'image obtenue après chaque exécution.

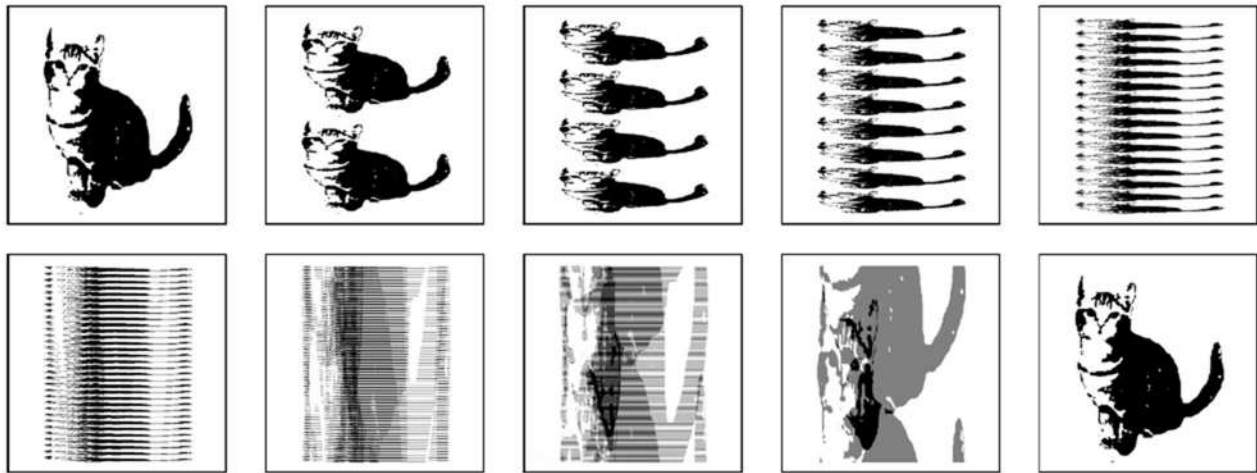
```

def echange_lignes(img): # matrice carrée
    n = len(img) # n pair
    for k in range(1, n//2):
        for i in range(k, n-1-k, 2):
            img[i], img[i+1] = img[i+1], img[i]

m = lire_fichier_nb('chat.pbm')
for i in range(1, 10):
    echange_lignes(m)
    ecrire_fichier_nb(m, "echange"+str(i))

```


Avec une image de 512 par 512 pixels, on retrouve la même image après neuf exécutions.



Exercice 7.5

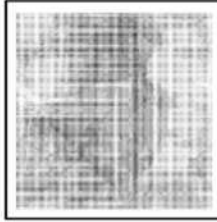
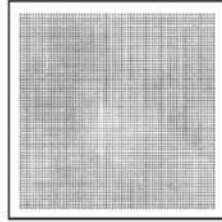
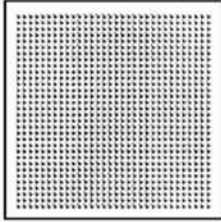
Pour écrire la fonction on suit le processus décrit dans l'énoncé. On écrit les fichiers images obtenus à chaque transformation.

```
def photomaton(img, k):
    n = len(img)
    mat1 = [[img[i][j] for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            if i % 2 == 0 and j % 2 == 0:
                img[i//2][j//2] = mat1[i][j]
            elif i % 2 == 1 and j % 2 == 0:
                img[(n+i)//2][j//2] = mat1[i][j]
            elif i % 2 == 0 and j % 2 == 1:
                img[i//2][(n+j)//2] = mat1[i][j]
            else:
                img[(n+i)//2][(n+j)//2] = mat1[i][j]
    ecrire_fichier_nb(img, "photomaton"+str(k))
    if k < 9:
        photomaton(img, k+1)

m = lire_fichier('chat.pbm')
photomaton(m, 1)
```

Avec une image de 512 par 512 pixels, on retrouve la même image après neuf exécutions.





Exercice 7.6

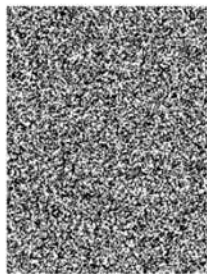
1. La fonction `hasard`.

On importe la fonction `randint` du module `random`. L'expression `randint(0, 1)` prend la valeur aléatoire 0 ou 1.

```
def hasard(n, p):
    from random import randint
    m = [[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            m[i][j] = randint(0, 1)
    return m

a = hasard(240, 180)
fichier(a, "alea") # fonction de l'exercice 10 chapitre 3
```

On obtient une image aléatoire comme celle-ci :



2. La fonction de chiffrement.

On définit une matrice `m` avec les bonnes dimensions remplies de 0. Ensuite on utilise l'opérateur *ou exclusif* entre chaque nombre de la matrice `img` et le nombre correspondant de la matrice `cle` pour obtenir la valeur de `m[i][j]`.

```
def chiffre(img, cle):
    n = len(img)
    p = len(img[0])
    m = [[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
```

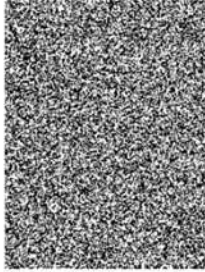
```

        m[i][j] = img[i][j] ^ cle[i][j]
    return m

c = chiffre(m1, a)
fichier(c, "image_chiffrée")

```

L'image chiffrée obtenue ressemble à une image aléatoire.



Il s'agit d'un chiffrement symétrique. Pour déchiffrer le résultat, on utilise la clé de chiffrement avec le même programme et on écrit simplement :

```

d = chiffre(c, a)
fichier(d, "image_déchiffrée")

```

On obtient une image identique à l'image initiale. On peut vérifier que le contenu des deux fichiers `image1.pbm` et `image_déchiffrée.pbm` est exactement le même.

3. On reprend la fonction `chiffre` qui sert aussi à déchiffrer et on remplace l'opérateur *ou exclusif* par l'opérateur *ou*.

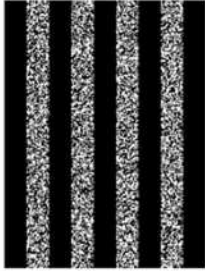
```

def devoile(img, cle):
    n = len(img)
    p = len(img[0])
    m = [[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            m[i][j] = img[i][j] | cle[i][j]
    return m

dv = devoile(c, a)
fichier(dv, "image_dévoilée")

```

On obtient une image qu'on dit « dévoilée ». Elle laisse apparaître l'image déchiffrée.



Pour travailler avec des images plus motivantes, on peut utiliser une image de son choix au format jpeg ou png et la convertir au format pbm. Après le chiffrement on reconvertit l'image chiffrée en jpeg ou png.

Pour des images en niveau de gris ou en couleur, les formats avec les mêmes principes que le pbm sont les formats pgm et ppm. Ces trois formats (P1, P2 et P3) utilisent le codage ASCII. Avec un codage brut sous forme d'octets on obtient les formats similaires (P4, P5 et P6), moins gourmands en taille de stockage.

Exercice 7.7

1. On commence par écrire une fonction qui calcule la distance entre deux points.

```
def distance(p1, p2): # renvoie le carré de la distance euclidienne
    x1, y1 = p1
    x2, y2 = p2
    return (x2-x1)**2 + (y2-y1)**2

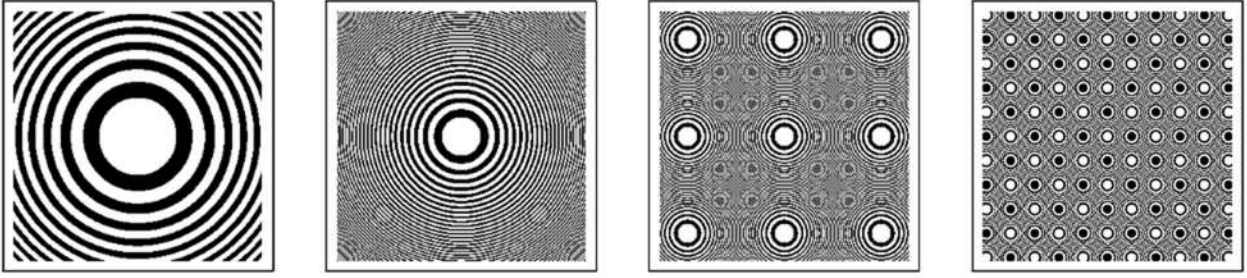
dim = 257

def matrice(n, r2): # on utilise le carré du rayon
    m = [[0 for j in range(n)] for i in range(n)]
    c = (n-1)//2
    centre = (c, c)
    for i in range(n):
        for j in range(n):
            temp = distance((i, j), centre) / r2
            e = int(temp)
            m[i][j] = e % 2
    return m

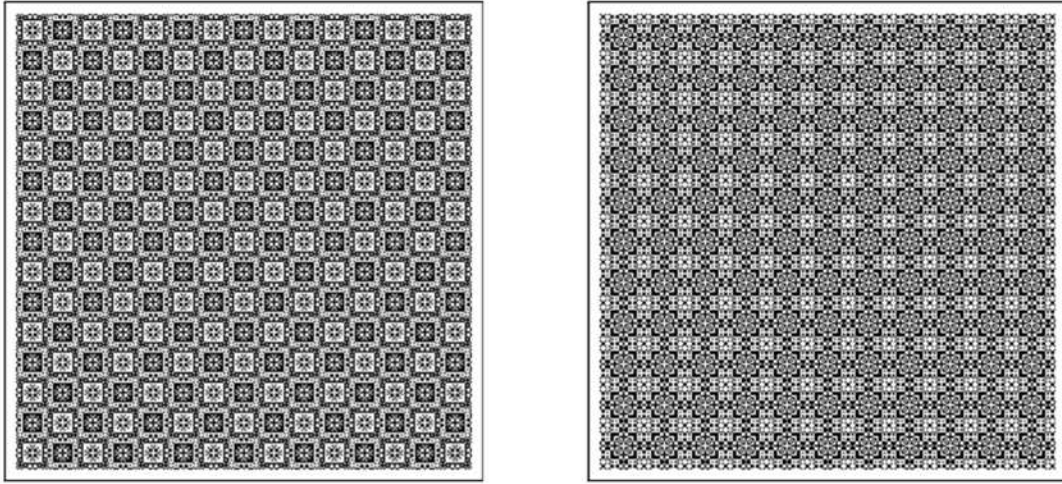
R = 20
m = matrice(dim, R**2)
ecrire_fichier_nb(m, "image1")

# tester avec R = 40 puis R = 20 puis R = 10 puis R = 5
# puis avec R = (17/11)**0.5 ou (5/3)**0.5, ...
```

2. On obtient les figures qui suivent pour $R = 40$, $R = 20$, $R = 10$ et $R = 5$.



On obtient les figures suivantes pour $R = \sqrt{17/11}$ et $R = \sqrt{23/14}$:



Exercice 7.8

Une fonction matrice :

```
from PIL import Image

def matrice(fichier):
    img = Image.open(fichier)
    largeur, hauteur = img.size
    liste = list(img.getdata())
    m = [[0 for j in range(largeur)] for i in range(hauteur)]
    for i in range(hauteur):
        for j in range(largeur):
            m[i][j] = liste[i*hauteur + j]
    return m
```

On peut tester le code en écrivant par exemple `mat = matrice('image1.jpg')` puis vérifier que la valeur de `len(mat)` est la hauteur de l'image et la valeur de `len(mat[0])` est la largeur de l'image. Diverses transformations présentées dans le cours peuvent être appliquées à cette matrice. Ensuite on crée une nouvelle image avec `img = Image.new("RGB", (largeur, hauteur))` et on insère les données avec `img.putdata(pixels)` après avoir retransformer la matrice `mat` en une simple liste `pixels`.

Algorithmes de tri

UNE SCIENTIFIQUE



Durant la Seconde Guerre mondiale, la *Moore School of Engineering* à Philadelphie embauche des femmes pour calculer les trajectoires balistiques. Parmi elles, **Betty Holberton** (1917-2001) est rapidement remarquée pour ses compétences et fait partie de l'équipe de conception de l'ENIAC, l'un des tout premiers ordinateurs. Là, elle développe le premier algorithme de tri et toute sa carrière est consacrée au développement de l'informatique en particulier des langages Fortran et Cobol.

■ Un peu d'histoire

Les premiers ordinateurs étaient avant tout de très puissants calculateurs d'où leur nom de *computer* donné en anglais, comme dans de nombreuses autres langues ; ce mot est construit sur le latin *computare* signifiant compter. Très rapidement on a pris conscience de leur grande utilité pour trier des fichiers. Alors qu'elle collaborait au fonctionnement de l'ENIAC, Betty Holberton est l'auteure du premier algorithme de tri connu en 1951. Cinq ans plus tard apparaît le tri à bulle ; qui s'avère rapidement insuffisant pour trier rapidement de gros fichiers. La complexité des algorithmes devient alors une préoccupation forte.

Au début des années 2000 apparaissent différents algorithmes adaptés à des problèmes spécifiques et permettant l'accessibilité à des informations dans des délais très brefs et ce, malgré l'immense masse des données traitées par des machines aux capacités toujours plus grandes.

OBJECTIFS

De nombreux algorithmes de tri ont été développés depuis les débuts de la programmation. Ils sont regroupés en différentes catégories.

- Connaître des algorithmes de tri quadratiques :
 - ▶ être capable de programmer un tri par sélection ou par insertion ;
 - ▶ savoir démontrer la validité ;
 - ▶ comprendre comment justifier le coût.
- Connaître des algorithmes de tri utilisant une stratégie « diviser pour régner » :
 - ▶ comprendre les stratégies de partition utilisées par le tri fusion et le tri rapide ;
 - ▶ apprécier l'intérêt de la récursivité ;
 - ▶ retenir leur coût suivant les cas.
- Connaître un algorithme de tri sans comparaison :
 - ▶ reconnaître un tri par comptage ;
 - ▶ être capable de justifier son coût.

Trier des données, c'est les ranger suivant un ordre défini au préalable. Les données peuvent être rangées suivant l'ordre croissant ou l'ordre décroissant.

C'est dans les années 1940 que le premier programme de tri a été développé par l'américaine Betty Holberton, l'une des six programmatrices de l'ENIAC (premier ordinateur électronique).

La plupart des algorithmes présentés dans la suite procèdent par des comparaisons successives entre deux éléments et éventuellement une permutation des deux éléments comparés. Le nombre de permutations est toujours inférieur au nombre de comparaisons et la complexité d'un algorithme est donc un ordre de grandeur du nombre de comparaisons effectuées.

Le nombre total r de rangements possibles de n données vaut $n \times (n-1) \times \dots \times 1$ noté $n!$. Après chaque comparaison, ce nombre de rangements possibles est divisé par 2. Donc après k comparaisons, il est divisé par 2^k . Le tri est terminé lorsqu'il ne reste plus qu'un seul rangement possible, c'est-à-dire dès que $2^k \geq r$ ou $k \geq \log_2(r)$. On montre alors que le minimum de comparaisons est de l'ordre de $n \log_2(n)$.

Application

Un tri est un préalable dans la recherche d'une médiane, ou plus généralement de quantiles dans une série de données. Avec une liste de longueur n triée dans l'ordre croissant, la médiane est l'élément d'indice $(n-1)//2$ si n est un nombre impair, et la moyenne des éléments d'indices $(n-2)//2$ et $n//2$ sinon.

En Python la méthode `sort` trie une liste en place (la liste est modifiée) et la fonction `sorted` renvoie une nouvelle liste triée. La syntaxe est `lst.sort()` ou `lst2 = sorted(lst1)`.

L'algorithme de tri utilisé est le **timsort**, du nom de son inventeur Tim Peters, en 2002. C'est un algorithme performant, dérivé de l'algorithme du *tri fusion*, qui utilise l'algorithme du *tri par insertion* sur des parties « réduites » ou « presque triées ». Ces deux algorithmes sont étudiés dans la suite.

■ Algorithmes de tri quadratiques

■ Tri par sélection

Le principe

On dispose de n données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi les données restantes et on la place en deuxième position, et ainsi de suite. Cet algorithme est souvent utilisé pour trier à la main des objets, comme des cartes ou des livres.

Si les données sont les éléments d'une liste `liste`, l'algorithme consiste donc à faire varier un indice i de 0 à $n-2$. Pour chaque valeur de i , on cherche dans la tranche `liste[i:n]` le plus petit élément et on l'échange avec `liste[i]`. On répète la recherche d'un minimum.

Écrire un algorithme du tri sélection consiste à insérer dans une boucle, où i varie de 0 à $n-2$, un algorithme de recherche du plus petit élément dans une liste, et pour chaque valeur de i à faire l'échange de `liste[i]` avec `liste[i.mini]`.

La donnée en entrée est une liste de n éléments. Il n'y a pas de résultat renvoyé en sortie, la liste est modifiée en place. On dit que *le tri sélection est un tri en place*.


```
def tri_selection(liste):
    for i in range(len(liste)-1):
        i_mini = i # indice du minimum
        mini = liste[i]
        for j in range(i+1, len(liste)):
            if liste[j] < mini:
                i_mini = j
                mini = liste[j]
        liste[i], liste[i_mini] = liste[i_mini], liste[i]
```

Exemple avec la liste [4, 4, 3, 2, 6, 5] et les éléments échangés.

Pour i égal à 0 :	[2, 4, 3, 4, 6, 5]	après échange de 2 et 4.
Pour i égal à 1 :	[2, 3, 4, 4, 6, 5]	après échange de 3 et 4.
Pour i égal à 2 :	[2, 3, 4, 4, 6, 5]	après aucun échange.
Pour i égal à 3 :	[2, 3, 4, 4, 6, 5]	après aucun échange.
Pour i égal à 4 :	[2, 3, 4, 4, 5, 6]	après échange de 5 et 6.

On remarque que le 4 qui était en début de liste se retrouve après le 4 qui était en deuxième position. On dit que *le tri sélection n'est pas stable*.

Pour utiliser cette fonction il suffit d'écrire l'instruction `tri_selection(liste)`.

Si nous ne voulons pas modifier la liste passée en paramètre il faut en faire une copie et ensuite appliquer l'algorithme de tri à cette nouvelle liste qui est renvoyée à la fin.

Validité de l'algorithme

Il est intéressant de noter qu'après k passages dans la boucle externe, les k premiers éléments de la nouvelle liste sont à leur place définitive.

■ Terminaison

La terminaison est simple à prouver. Nous avons deux boucles `for` imbriquées et le nombre de passages dans ces deux boucles est parfaitement déterminé et il est évidemment fini.

■ Correction

Nous prouvons la correction en utilisant un invariant de boucle : "pour chaque i , la liste est une permutation de la liste initiale, la liste `liste[0:i+1]` est triée et tous les éléments de la liste `liste[i+1:n]` sont supérieurs à tous les éléments de la liste `liste[0:i+1]`".

Pour chaque valeur de i , au plus une permutation de deux éléments distincts a lieu et elle a lieu seulement si `liste[i]` n'est pas le minimum de `liste[i:n]`. C'est pourquoi après chaque passage dans la boucle externe, la nouvelle liste est une permutation de la liste initiale.

Après le premier passage dans la boucle, pour i égal à 0, la liste `liste[0:1]` ne contient qu'un élément, le minimum de la liste, qui est inférieur à tous les éléments de la liste. La propriété est donc vraie pour i égal à 0.

Si après un passage pour i égal à un k quelconque, la liste `liste[0:k+1]` est triée et tous les éléments de `liste[k+1:n]` sont supérieurs à tous les éléments de `liste[0:k+1]`, alors au passage suivant le minimum de la liste `liste[k+1:n]` est placé en position d'indice $k+1$. Ce minimum est supérieur à tous les éléments de la liste `liste[0:k+1]` et inférieur à tous les éléments de la liste `liste[k+2:n]`. La propriété est donc vraie pour i égal à $k+1$.

La propriété est donc encore vraie après le dernier passage, pour i égal à $n - 2$. Donc la liste `liste[0:n-1]` est triée et l'élément d'indice $n-1$, le dernier de la liste, est supérieur à tous les éléments de la liste `liste[0:n-1]`. Donc la liste `liste[0:n]`, soit toute la liste, est triée.

Coût de l'algorithme

Quels que soient les éléments d'une liste de longueur n , pour chaque valeur de i , j prend les valeurs de $i+1$ à $n-1$, soit $n-i-1$ valeurs. Et pour chaque valeur de j , une unique comparaison est effectuée. Donc, pour chaque valeur de i , nous avons exactement $n-i-1$ comparaisons.

Au total, nous obtenons : $(n-1) + (n-2) + \dots + 2 + 1$ comparaisons, soit $n(n-1)/2$ comparaisons. Le coût est donc de l'ordre de n^2 quelle que soit la liste de longueur n , même si elle est déjà triée. Cela signifie que le tri par sélection n'est pas très efficace. Il est cependant simple à programmer et utile dans le cas de listes ne comptant pas plus de 10^4 éléments.

À retenir : dans tous les cas, l'algorithme de tri par sélection sur une liste de n éléments a un coût quadratique en fonction de n . Le nombre de comparaisons est de l'ordre de n^2 .

■ Tri par insertion

Le principe

On dispose de n données. À chaque étape, on suppose que les k premières données sont triées et on insère une donnée supplémentaire à la bonne place parmi ces k données.

Si les données sont les éléments d'une liste, l'algorithme consiste donc à faire varier un indice i de 0 à $n - 2$. Pour chaque valeur de i , on cherche dans la liste `liste[0:i+1]` à quelle place doit être inséré l'élément `liste[i+1]` qu'on appelle la clé. Pour cela on compare la clé successivement aux données précédentes, en commençant par la donnée d'indice i puis en remontant dans la liste jusqu'à trouver la bonne place, c'est-à-dire entre deux données successives, l'une étant plus petite et l'autre plus grande que la clé. Si la clé est plus petite que toutes les données précédentes, elle se place en premier. Pour ce faire, on décale d'une place vers la droite les données plus grandes que la clé après chaque comparaison.

La donnée en entrée est une liste de n éléments. Il n'y a pas de résultat renvoyé en sortie, la liste est modifiée en place. On dit que *le tri insertion est un tri en place*.

```
def tri_insertion(liste):
    for i in range(len(liste)-1):
        k = i + 1 # indice de la cle
        cle = liste[k]
        while k > 0 and cle < liste[k-1]:
            liste[k] = liste[k-1]
            k = k - 1
        liste[k] = cle
```

Exemple avec la liste `[4, 4, 3, 2, 6, 5]` et les clés successives.

Pour i égal à 0	avec la clé 4 :	<code>[4, 4, 3, 2, 6, 5]</code> .
Pour i égal à 1	avec la clé 3 :	<code>[3, 4, 4, 2, 6, 5]</code> .
Pour i égal à 2	avec la clé 2 :	<code>[2, 3, 4, 4, 6, 5]</code> .
Pour i égal à 3	avec la clé 6 :	<code>[2, 3, 4, 4, 6, 5]</code> .
Pour i égal à 4	avec la clé 5 :	<code>[2, 3, 4, 4, 5, 6]</code> .

Un examen approfondi de l'algorithme montre que le *tri insertion est stable*. Deux éléments de même valeur placés dans un certain ordre avant le tri restent dans le même ordre après le tri.

Pour utiliser cette fonction il suffit d'écrire l'instruction `tri_insertion(liste)`.

Si nous ne voulons pas modifier la liste passée en paramètre, il faut en faire une copie, trier cette nouvelle liste et ensuite la renvoyer.

On peut noter ici qu'après k passages dans la boucle, les k premiers éléments de la liste sont triés. Mais ils ne sont pas, à priori, à leur place définitive.

Validité de l'algorithme

■ Terminaison

La boucle externe est une boucle `for` donc le nombre de passages est déterminé et fini. La boucle interne est une boucle `while`. Les valeurs prises par le variant k constituent une suite d'entiers strictement décroissante incluse dans la suite des entiers de $i+1$ à 0. Il y a donc, pour chaque i , au plus $i+1$ passages dans la boucle `while`.

■ Correction

Pour prouver la correction nous utilisons un invariant de boucle : "pour chaque i , la liste est une permutation de la liste initiale et la liste `liste[0:i+2]` est triée".

Le principe de l'insertion assure que pour chaque i , la liste modifiée est une permutation de la liste initiale.

Après le premier passage dans la boucle, pour i égal à 0, l'élément `liste[0]` et la première clé, d'indice 1, sont rangés dans l'ordre. Donc la liste `liste[0:2]` est triée. La propriété est donc vraie pour i égal à 0.

Si après un passage pour i égal à un k quelconque, la liste `liste[0:k+2]` est triée, alors au passage suivant l'élément `liste[k+2]` est inséré à la bonne place parmi les éléments de la liste `liste[0:k+2]` ou reste à sa place. Donc la liste `liste[0:k+3]` est triée. La propriété est donc vraie pour i égal à $k+1$.

La propriété est donc encore vraie après le dernier passage, pour i égal à $n-2$. À ce moment la liste `liste[0:n]`, c'est-à-dire la liste `liste`, est triée.

Coût de l'algorithme

Nous avons deux boucles imbriquées. Pour une liste de longueur n , le nombre de comparaisons peut être différent suivant la liste.

Si la liste est déjà triée, pour chaque valeur de i , k prend la valeur de $i+1$ et il y a une seule comparaison, le test `cle < liste[k-1]`. La variable i prenant $n-1$ valeurs, cela nous fait un total de $n-1$ comparaisons. Le coût de l'algorithme est donc de l'ordre de n .

Si par contre les éléments de la liste sont rangés dans l'ordre décroissant, alors pour chaque valeur de i , k prend les valeurs de $i+1$ à 1 soit $i+1$ valeurs et donc $i+1$ comparaisons.

Au total, nous avons donc : $1 + 2 + \dots + (n-2) + (n-1)$ comparaisons. Un calcul mathématique nous donne $n(n-1)/2$ comparaisons. Le coût est donc de l'ordre de n^2 comparaisons.

On peut montrer qu'en moyenne, le coût est de l'ordre de n^2 comparaisons, comme pour le tri par sélection. Mais le tri par insertion est très intéressant si la liste est "presque triée".

À retenir : dans le pire des cas, et en moyenne, l'algorithme de tri par insertion sur une liste de n éléments a un coût quadratique en fonction de n . Le nombre de comparaisons est de l'ordre de n^2 . Mais dans le meilleur des cas, une liste déjà triée, le coût est linéaire. L'algorithme est en général efficace sur une liste « presque triée ».

Les deux algorithmes de tri étudiés (par sélection et par insertion) ne sont pas très efficaces en général. En effet, pour une liste de taille n , le coût en terme de nombre de comparaisons est de l'ordre de n^2 en moyenne. Mais le tri insertion est souvent utilisé pour des « petites » listes ou des listes « presque » triées car dans ces cas il est l'un des plus efficaces.

■ Stratégie « diviser pour régner »

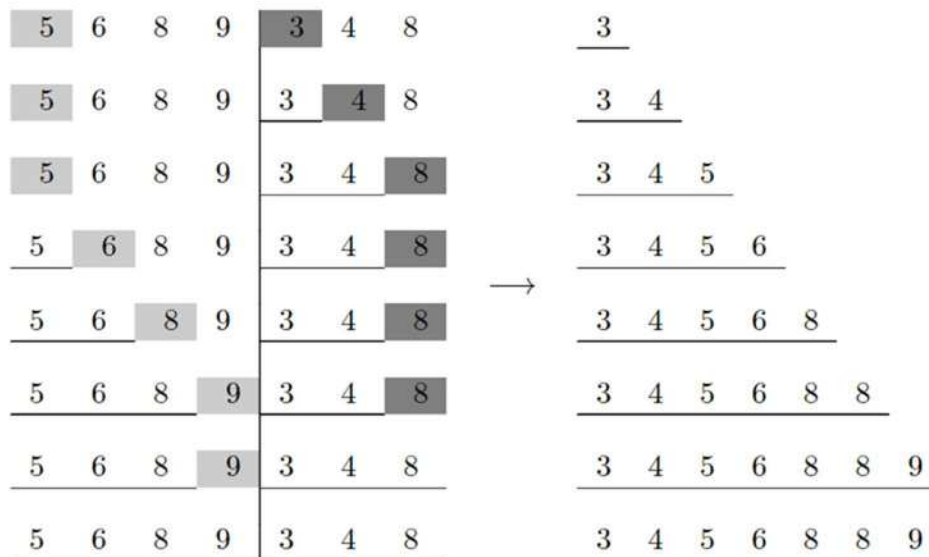
Principe d'une stratégie « diviser pour régner » : on « divise » en réduisant un problème en sous-problèmes du même type, puis on « règne » en résolvant ces sous-problèmes. Il reste ensuite à combiner les solutions des sous-problèmes pour obtenir une solution au problème initial. Un point très important est de ne résoudre qu'une seule fois chaque sous-problème.

■ Tri fusion

Le principe du tri fusion, en anglais *merge sort*, est simple. La liste à trier est partagée en deux parties de tailles égales à une unité près. Il s'agit d'un tri dichotomique. Un appel récursif est alors réalisé sur chacune des deux parties. Lorsque ces deux parties sont triées, elles sont fusionnées en une liste triée.

D'après Donald Knuth, le tri fusion est l'une des toutes premières méthodes de tri utilisées sur un ordinateur, et a été suggérée par John von Neumann dès 1945 (*The Art of Computer Programming* - vol 3 - Sorting and Searching).

Partager, sans aucune condition, une liste en deux parties est simple. Le programme repose donc sur l'écriture d'une fonction `fusion` qui prend deux listes triées en paramètres et renvoie une liste triée composée de la réunion des éléments de chaque liste. Pour effectuer le tri fusion, on fusionne deux par deux des parties contiguës de la liste qui ont été triées. Voici un exemple avec [5,6,8,9] et [3,4,8] à gauche, et le résultat à droite. Les éléments à comparer de chaque partie sont respectivement en gris clair et en gris foncé, et on choisit à chaque étape le plus petit. Les éléments choisis sont soulignés.



Une nouvelle liste est créée. La fonction `fusion` définie ci-dessous suit ce procédé.


```

def fusion(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
    for k in range(i, len(liste1)): # s'il reste des éléments dans liste1
        liste.append(liste1[i])
    for k in range(j, len(liste2)): # s'il reste des éléments dans liste2
        liste.append(liste2[j])
    return liste

```

Pour effectuer un tri, cette fonction `fusion` est appelée de manière récursive sur des parties de la liste à trier. La fonction de tri `tri_fusion` peut s'écrire ainsi :

```

def tri_fusion(liste):
    if len(liste) < 2:
        return liste
    else:
        milieu = len(liste) // 2
        liste1 = tri_fusion(liste[:milieu])
        liste2 = tri_fusion(liste[milieu:])
        return fusion(liste1, liste2)

```

La fonction renvoie une liste triée et la liste initiale n'est pas modifiée.

À chaque appel récursif sur une sous-liste, tous les éléments de la sous-liste sont mobilisés dans la construction de deux nouvelles sous-listes. Nous en déduisons que *le tri fusion n'est pas un tri en place*. Un examen des fonctions `fusion` et `tri_fusion` montre que *le tri fusion est un tri stable*.

Terminaison, correction

Les appels récursifs sont effectués sur des listes dont la taille est strictement décroissante. Si la taille de la liste initiale est n , cette taille est divisée par deux (à 1 près) à chaque appel récursif. Les tailles successives sont donc de l'ordre de $n/2$, $n/4$, $n/8$, etc. Après au plus k étapes, où $2^k \geq n$, les tailles ont atteint la valeur 1 et les appels sont arrêtés. Le nombre d'appels récursifs est fini.

Le nombre d'appels récursifs vaut $2 \times p$ où p est le nombre de fusions. Pour calculer approximativement le nombre de fusions, disons que nous avons 1 fusion de 2 listes de taille $n/2$, 2 fusions de listes de taille $n/4$, etc., soit $1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$ fusions, avec $2^k \geq n$. Le nombre de fusions est donc de l'ordre de n et le nombre d'appels récursifs est de l'ordre de $2n$ où n est la taille de la liste.

Pour la correction, nous pouvons remarquer qu'une liste de taille 1 est triée et que la fusion de deux listes triées produit une liste triée.

Coût

Nous commençons par exprimer le coût d'une fusion : pour deux listes dont les tailles respectives sont m et p , le coût d'une fusion est de l'ordre de $m + p$.

Ce résultat s'obtient après observation de la fonction `fusion`.

Le tri fusion d'une liste de taille $n = 2^k$ nécessite donc :

- ▶ $2^0 = 1$ fusion de listes de tailles $n/2^1$ dont le coût est n ;
- ▶ $2^1 = 2$ fusions de listes de tailles $n/2^2$ dont le coût est n ;
- ▶ $2^2 = 4$ fusions de listes de tailles $n/2^3$ dont le coût est n ;
- ...
- ▶ $2^{k-1} = n/2$ fusions de listes de tailles $n/2^k = 1$ dont le coût est n ;

Nous avons k étapes, chacune de coût n , où k est le nombre de chiffres de n en base 2. Le coût est donc de l'ordre de $k \times n$, soit environ $n \log_2(n)$.

De manière générale, le tri fusion d'une liste de taille n a une complexité de l'ordre de $n \log_2(n)$.

Si N est le nombre de chiffres dans l'écriture décimale de n , alors le coût est de l'ordre de $N \times n$. Le coût est « presque » linéaire. Par exemple, pour n suffisamment grand, si on passe d'une liste de taille n à une liste de taille $100n$, le coût est à peu près multiplié par 100.

Remarque : il est possible d'envisager une approche ascendante sans récursivité. En effet, il ne s'agit que d'une question de fusions qui sont effectuées principalement avec de petites listes (d'abord de taille 1, puis de taille 2, puis de taille 4, etc). C'est l'objet du programme qui suit.

Le code de ce programme est aussi prétexte à présenter une fonction `fusion` codée différemment, (mais cela n'a pas de conséquence sur le code de la fonction de tri qui suit, chacun peut coder la fonction `fusion` à sa manière). On utilise une liste auxiliaire afin de pouvoir fusionner les sous-listes directement dans la liste initiale qui est donc modifiée. La fonction de tri est alors écrite sans instruction `return`. Sans programmer un tri véritablement en place, on limite quand même la création de listes qui occupent la mémoire de la machine.

```
def fusion(liste, g, m, d):
    aux = (d - g + 1) * [None]
    for i in range(m-g+1):
        aux[i] = liste[g+i] # dans l'ordre
    for j in range(m-g+1, d-g+1):
        aux[j] = liste[d-g+m+1-j] # dans l'ordre inverse
    i = 0
    j = d - g
    for k in range(g, d+1):
        if aux[i] < aux[j]:
            liste[k] = aux[i]
            i = i + 1
        else:
            liste[k] = aux[j]
            j = j - 1
```



```
def tri_fusion_asc(liste): #approche ascendante
    n = len(liste)
    taille = 1
    while taille < n:
        for g in range(0, n-taille, 2*taille):
            fusion(liste, g, g+taille-1, min(g+2*taille-1, n-1))
        taille = 2 * taille
```

■ Tri rapide

Deux stratégies différentes peuvent être mises en œuvre.

Avec le tri fusion, les sous-problèmes sont résolus avec une fonction `tri_fusion` et ensuite les résultats sont combinés avec une fonction `fusion` :

```
liste1 = tri_fusion(liste[:milieu])
liste2 = tri_fusion(liste[milieu:])
fusion(liste1, liste2)
```

En 1960 le britannique Charles Antony Richard Hoare invente le tri rapide (*quicksort*) qu'il décrit dans un papier paru en 1961.

Un pré-traitement de la liste est effectué par une fonction `partition`, puis les sous-problèmes sont résolus de manière à ce que les résultats se combinent seuls, sans travail supplémentaire :

```
p = partition(liste, g, d)
tri_rapide(liste, g, p-1)
tri_rapide(liste, p+1, d)
```

Cette stratégie applique encore le principe *diviser pour régner*. La liste de nombres à trier est partagée en deux parties à l'aide d'une valeur choisie nommée la *limite* ou le *pivot*. Cette limite peut être n'importe quel élément de la liste. Une partie contient les valeurs plus petites que la limite et l'autre les valeurs plus grandes.

Voici la méthode décrite par Hoare appliquée à une liste de longueur n :

- ▶ on choisit un élément de la liste qui est la limite ;
- ▶ on parcourt les éléments de la liste à l'aide deux indices appelés respectivement *indice gauche* et *indice droit*, notés g et d ;
- ▶ l'indice gauche commence à 0 et on se déplace vers la droite en l'augmentant d'une unité tant que l'on rencontre des valeurs inférieures ou égales à la limite ;
- ▶ l'indice droit commence à $n-1$ et on se déplace vers la gauche en le diminuant d'une unité tant que l'on rencontre des valeurs supérieures ou égales à la limite ;
- ▶ les deux valeurs où s'arrêtent les indices g et d sont du mauvais côté de la limite donc elles sont échangées ;
- ▶ chaque indice est incrémenté ou décrémenté d'une unité dans la direction respective de déplacement et l'indice gauche recommence son déplacement ;
- ▶ ce processus est reproduit jusqu'à ce que les indices g et d se croisent ;
- ▶ on place la limite à la bonne place, par exemple en l'échangeant avec l'élément d'indice d si la limite qui a été choisie est le premier élément de la liste.

Finalement, la liste contient au début des éléments inférieurs ou égaux à la limite, puis la limite, puis des éléments supérieurs ou égaux à la limite. La limite est à la bonne place. On peut alors reproduire le processus sur la partie gauche et sur la partie droite de la liste. Le nombre total d'éléments à trier, séparés en deux parties, est diminué d'une unité.

Exemple d'une fonction `partition` qui applique l'algorithme décrit précédemment sur une partie d'une liste délimitée par les indices `i` et `j`, avec pour limite l'élément d'indice `i` :

```
def partition(liste, i, j):
    limite = liste[i]
    g = i + 1
    d = j
    while g <= d:
        while g <= j and liste[g] <= limite:
            g = g + 1
        while d > i and liste[d] >= limite:
            # d > i est inutile si on remplace par while liste[d] > limite
            d = d - 1
        if g < d:
            liste[g], liste[d] = liste[d], liste[g]
            g = g + 1
            d = d - 1
    liste[d], liste[i] = liste[i], liste[d]
    return d
```

Les deux parties sont déterminées par l'indice `d` renvoyé, celui de la limite.

Le tri d'une liste est exécuté à l'aide d'appels récursifs sur les parties successivement déterminées par la fonction `partition`. Il est implémenté dans la procédure suivante :

```
def tri_rapide(liste, g, d):
    if g < d:
        p = partition(liste, g, d)
        tri_rapide(liste, g, p - 1)
        tri_rapide(liste, p + 1, d)
```

Aucune nouvelle liste n'est créée. Les parties sont délimitées par les indices des extrémités. Le tri d'une liste est obtenu par l'appel initial `tri_rapide(liste, 0, len(liste)-1)`. C'est la liste passée en paramètre qui est modifiée. *Le tri rapide est un tri en place.*

On peut définir une fonction `tri` plus simple d'utilisation.

```
def tri(liste):
    tri_rapide(liste, 0, len(liste)-1)
```

Voici un exemple de l'exécution de la fonction partition où la limite est le nombre 6, la valeur en gris clair correspond à l'indice *g* et celle en gris foncé à l'indice *d* :

6	5	8	9	3	4
6	5	8	9	3	4
6	5	4	9	3	8
6	5	4	9	3	8
6	5	4	3	9	8
6	5	4	3	9	8
3	5	4	6	9	8

On peut remarquer en remplaçant le 3 par un 4, que *le tri rapide n'est pas stable*.

La condition `liste[d] >= limite` peut être remplacée par la condition `liste[d] > limite`. On obtient alors une partie gauche qui contient les éléments inférieurs ou égaux à la limite et une partie droite qui contient les éléments strictement supérieurs à la limite.

Il existe d'autres manières d'obtenir ce type de partition d'une liste afin d'effectuer le tri rapide. Certaines sont présentées en exercices.

Terminaison et correction

Pour une sous-liste de taille *k*, dans le pire des cas, les deux appels récursifs sont effectués sur une sous-liste vide et sur une sous-liste de taille *k* - 1. Lorsque la taille d'une sous-liste atteint la valeur 0 ou la valeur 1, ce qui se produit après au plus *n* partitions si la taille de la liste initiale est *n*, il n'y a plus d'appel récursif. Ceci assure la terminaison.

Une liste de taille 1 ou 0 est triée. Pour une liste de taille supérieure, la fonction `partition` positionne la limite à la bonne place, sa place définitive dans la liste triée. De plus les éléments de la sous-liste gauche sont tous inférieurs à la limite et les éléments de la sous-liste droite sont tous supérieurs à la limite.

Coût

Si *n* est la taille de la liste, on montre que le coût du tri rapide est en $n \log_2(n)$ dans le meilleur des cas et en moyenne. Dans le pire des cas il est en n^2 . Ce cas survient par exemple si on choisit pour limite le premier élément et que les éléments sont déjà rangés suivant l'ordre croissant. Après chaque appel de la fonction `partition`, on obtient une partie vide et une partie qui contient exactement un élément de moins que la partie initiale. Or, le coût de la fonction `partition` est de l'ordre de la taille de la sous-liste traitée, chaque élément étant examiné une seule fois.

■ Tri sans comparaison

Les tris présentés utilisent des comparaisons. D'autres algorithmes peuvent utiliser la structure des données. C'est le cas des tris par comptage, par base, par paquets. Le tri par comptage est présenté ici.

On dispose d'une liste d'entiers naturels qui sont tous inférieurs ou égaux à un entier naturel non nul *m* qui est de l'ordre de *n*, la taille de la liste.

On commence par écrire une fonction `comptage`, d'arguments une liste `entiers` et un entier `m`, renvoyant une liste de longueur `m + 1` telle pour tout `k` de 0 à `m`, l'élément d'indice `k` a pour valeur le nombre d'occurrences de l'entier `k` dans la liste `entiers`. Pour cela, on crée une liste composée de 0, de longueur `m+1`. Chaque élément de cette liste sert de compteur.

Une première version :

```
def comptage(entiers, m):
    compteurs = (m + 1) * [0]
    for k in range(m+1):
        cpt = 0
        for p in entiers:
            if p == k:
                cpt = cpt + 1
        compteurs[k] = cpt
    return compteurs
```

Cet algorithme contient deux boucles imbriquées et son coût est de l'ordre de $m \times n$, donc quadratique en `n` puisque la valeur de `m` est de l'ordre de la longueur `n` de la liste. Voici donc une deuxième version, plus efficace, dont le coût en fonction de la longueur `n` de la liste est linéaire :

```
def comptage(entiers, m):
    compteurs = (m + 1) * [0]
    for k in entiers:
        compteurs[k] = compteurs[k] + 1
    return compteurs
```

Les valeurs des éléments de la liste `entiers` sont représentés par les indices de la liste `compteurs`.

On en déduit une fonction `tri`, d'arguments une liste `entiers` et un entier `m`, renvoyant la liste triée dans l'ordre croissant.

```
def tri(entiers, m):
    cpts = comptage(entiers, m)
    triee = []
    for i in range(m+1):
        triee = triee + cpts[i] * [i]
    return triee
```

Le nombre d'opérations dans la fonction `comptage` est de l'ordre de `n`, la longueur de la liste à trier, (car `m` est de l'ordre de `n`), et dans la fonction `tri`, on construit une liste de même longueur. Le coût total est donc linéaire en la longueur de la liste.

Si l'entier `m` n'est pas donné, il suffit de déterminer le maximum de la liste. Le coût total ne change pas puisque la recherche d'un maximum a un coût linéaire en la longueur de la liste.

■ ■ Vrai/Faux

	Vrai	Faux
1. Le tri par sélection consiste à placer un par un les éléments à leur place définitive.	☐	☐
2. Après quelques étapes du tri par insertion, mais avant que le tri soit terminé, nous sommes sûrs qu'au moins un élément est rangé à sa place définitive.	☐	☐
3. Sur une liste déjà triée, le tri sélection et le tri insertion ont un coût linéaire.	☐	☐
4. La meilleure complexité qu'on peut obtenir dans le pire des cas avec un tri par comparaison est en $n \log_2(n)$ où n est la taille de la liste à trier.	☐	☐
5. Quels que soient les éléments d'une liste de taille n , un tri par sélection a un coût quadratique.	☐	☐
6. Le coût du tri fusion sur une liste de taille n est dans le pire des cas de l'ordre de n^2 .	☐	☐
7. Avec un tri rapide la liste à trier est partagée en deux sous-listes de taille égales, à une unité près.	☐	☐
8. Le coût d'un tri rapide est le même que celui d'un tri fusion.	☐	☐
9. Dans le cas d'une liste déjà triée, un tri insertion est plus rapide qu'un tri rapide.	☐	☐
10. Dans le cas d'une liste déjà triée, un tri sélection est plus rapide qu'un tri fusion.	☐	☐

■ ■ Énoncé des exercices

Exercice 8.1 : Écrire une fonction `permuter` qui prend en argument une liste de mots et modifie la liste en permutant le mot le plus court en nombre de lettres avec le premier mot de la liste. La fonction ne renvoie rien.

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

Exercice 8.2 : Temps d'exécution

Pour mesurer le temps d'exécution d'un programme, on importe la fonction `time` du module `time` avec l'instruction `from time import time`.

On souhaite comparer les temps d'exécution du tri insertion et du tri fusion sur deux types de listes : une liste de nombres au hasard et une liste de nombres déjà triée.

1. Construire une liste de 3000 entiers pris au hasard entre 1 et 10000, bornes comprises. Mesurer les temps d'exécution des programmes du tri insertion et du tri fusion pour trier cette liste. Penser à reconstruire la liste entre les deux tris !

Quel commentaire peut-on faire concernant les deux résultats ?

2. Construire la liste des 3000 entiers de 0 à 2999, bornes comprises. Mesurer le temps d'exécution du programme du tri insertion et du programme du tri fusion pour trier cette liste.

Quel commentaire peut-on faire concernant les deux résultats ?

Exercice 8.3* : Ordre lexicographique

L'objectif est d'écrire un programme qui trie une liste de mots (type `str`) et les range suivant l'ordre lexicographique (l'ordre des dictionnaires).

1. Écrire la définition de la variable `alphabet` :

`alphabet = "AaàBbCcDdEeèFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuùVvWwXxYyZz"`

2. Écrire une fonction `ordre_alphabetique` qui prend en arguments deux caractères alphabétiques `c1` et `c2` et renvoie `-1` si `c1` est avant `c2`, `1` si `c2` est avant `c1` et `0` si `c1 = c2`. On pourra utiliser la méthode `index` qui renvoie l'indice d'un élément dans une chaîne de caractères.

3. Écrire une fonction `ordre_lexicographique` qui prend en arguments deux mots `m1` et `m2` et renvoie `-1` si "`m1 < m2`" pour l'ordre lexicographique, `0` si "`m1 = m2`" et `1` si "`m1 > m2`". On utilisera la fonction `ordre_alphabetique`.

4. Écrire une fonction `tri_lexicographique` qui prend en argument une liste de mots et trie cette liste. On utilisera la fonction `ordre_lexicographique` avec l'algorithme du tri par insertion.

Exercice 8.4* : Trier des points

On dispose de points dans le plan muni d'un repère orthonormé d'origine $\mathcal{O}$. Ces points possèdent un couple de coordonnées $(x; y)$ représenté par la liste `[x, y]`. Nous allons 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 de ce point à $\mathcal{O}$.

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 la distance entre les points et $\mathcal{O}$. Utiliser un algorithme de tri du cours.

Exercice 8.5* : Programmation du tri par insertion de manière récursive.

1. Écrire une fonction récursive `insertion` qui prend trois paramètres, une liste, un élément de la liste `x` et son indice `n`. Si `n` est strictement positif, on suppose les éléments d'indice 0 à `n-1` triés et la fonction insère l'élément `x` à la bonne place.
2. Écrire une fonction récursive `tri_insertion` qui prend en paramètres une liste et la longueur de la liste et qui trie la liste.

Exercice 8.6* : Programmer un algorithme du tri insertion avec une insertion qui utilise une recherche dichotomique.

Exercice 8.7 : On considère la deuxième version de la fonction `fusion` donnée dans le cours avec la représentation de l'état à l'entrée dans la boucle pour la liste donnée en exemple. On complète avec la variable `k` qui prend successivement les valeurs allant de `g` à `d` incluses.

État à l'entrée dans la boucle :

	k							
liste ...	5	6	8	9	3	4	8	...
	g							
aux ...	5	6	8	9	8	4	3	...
	i							

Sur cet exemple et avec ce type de représentation, détailler pas à pas l'état de la liste à trier pendant l'exécution de la fonction `fusion`.

Exercice 8.8 : On trouve différentes manières de coder la fonction `partition` utilisée dans le tri rapide. Voici un exemple de code pour cette fonction :

```
def partition(liste, g, d):
    x = liste[d]
    i = g
    for j in range(g, d):
        if liste[j] <= x:
            liste[i], liste[j] = liste[j], liste[i]
            i = i + 1
    liste[i], liste[d] = liste[d], liste[i]
    return i
```

Expliciter tous les détails de l'exécution lors de l'appel `partition(liste, 0, len(liste)-1)` si la liste passée en paramètre est `[3, 8, 9, 1, 6, 4, 2, 5]`.

Exercice 8.9* : Tri rapide et partition d'après Hoare.

Dans la fonction `partition` du cours et dans celle de l'exercice précédent, il est procédé à un échange de valeurs. Ceci est particulièrement simple en Python avec une instruction comme

`liste[g], liste[d] = liste[d], liste[g]`. Dans la plupart des autres langages, cela nécessite l'utilisation d'une troisième variable temporaire.

Reprendre la fonction `partition` du cours, en utilisant pour limite l'élément de droite, et la modifier pour éviter de procéder à des échanges de valeurs.

■ ■ Indications

Ex. 8.1

C'est une recherche de minimum, sur le nombre de lettres d'un mot. Une variable est nécessaire pour stocker l'indice du minimum.

Ex. 8.3

Il faut tenir compte de la longueur des mots. Par exemple "bon" est avant "bonjour". La méthode `index` s'utilise simplement, par exemple : `"abcde".index("d")` vaut 3.

Ex. 8.5

Distinguer les cas extrêmes pour la condition d'arrêt.

Ex. 8.9

Faire varier séparément les indices en procédant à une affectation avant de passer à l'autre indice. Une place est libérée après chaque nouvelle affectation.

■ ■ Corrigé du vrai/faux

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

Quelques explications

2. Si le dernier élément à insérer est le plus petit, tous les autres éléments vont être décalés.
6. Le coût du tri fusion est dans tous les cas de l'ordre de $n \log_2 n$.
8. C'est faux dans le pire des cas.
9. Le coût d'un tri insertion est linéaire si la liste est déjà triée.

❑ Erreurs classiques et conseils.—

- Attention au risque de confusion entre l'algorithme du tri par insertion et celui du tri par sélection. Chacun doit être bien compris.
- La complexité dans tous les cas de chaque algorithme de tri présenté dans le cours doit être connue.
- Il faut retenir que la complexité dans le pire des cas d'un tri par comparaison sur une liste de taille n est au mieux en $n \log_2(n)$.

■ ■ Corrigé des exercices

Exercice 8.1

On cherche le mot le plus court, c'est une recherche de minimum. Il faut stocker son indice dans une variable pour pouvoir effectuer la permutation à la fin.

```
def permute(liste):
    n = len(liste)
    nblettres = len(liste[0])
    indice = 0
    for i in range(1, n):
        if len(liste[i]) < nblettres:
            nblettres = len(liste[i])
            indice = i
    liste[0], liste[indice] = liste[indice], liste[0]
```

Test :

```
>>> maliste = ['toto', 'bonjour', 'a', 'oui', 'non']
>>> permute(maliste)
>>> maliste
['a', 'bonjour', 'toto', 'oui', 'non']
```

Exercice 8.2

On importe la fonction `time` et on utilise les codes du cours des fonctions de tri.

1. On crée la liste aléatoire et on mesure le temps pour chaque tri.

```
from random import randint

liste = [randint(1, 10000) for i in range(3000)]
top = time()
tri_insertion(liste)
print(time() - top)

liste = [randint(1, 10000) for i in range(3000)]
top = time()
tri_fusion(liste)
print(time() - top)
```

Le résultat obtenu est environ 0.747 seconde pour le tri insertion et 0.026 seconde pour le tri fusion. Ces résultats sont bien sûr légèrement variables si on répète l'expérience ou si on change de machine.

2. On crée la liste triée et on mesure le temps passé pour chaque tri.

```
liste = [i for i in range(3000)]
top = time()
tri_insertion(liste)
print(time() - top)

top = time()
tri_fusion(liste)
print(time() - top)
```

Les résultats obtenus sont 0.001 seconde pour le tri insertion et 0.008 pour le tri fusion. Cette fois la différence de coût apparaît en faveur du tri insertion. En modifiant le code, avec une liste triée de 100 000 entiers, le temps affiché pour le tri par insertion est 0.031, ce qui s'explique par un coût linéaire.

Il est conseillé d'effectuer les tests plusieurs fois et de calculer une moyenne des résultats.

Exercice 8.3

- 1.

```
alphabet = 'AaàBbCcDdEeéèFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuùVvWwXxYyZz'
```

2. Fonction `ordre_alphabetique` :

```
def ordre_alphabetique(c1, c2):
    if alphabet.index(c1) < alphabet.index(c2):
        return -1
    elif alphabet.index(c2) < alphabet.index(c1):
        return 1
    else:
        return 0
```

3. Fonction `ordre_lexicographique` :

```
def ordre_lexicographique(m1, m2):
    n = min(len(m1), len(m2))
    for i in range(n):
        if ordre_alphabetique(m1[i], m2[i]) == -1:
            return -1
        elif ordre_alphabetique(m1[i], m2[i]) == 1:
            return 1
    if len(m1) < len(m2):
        return -1
```

```

elif len(m2) < len(m1):
    return 1
else:
    return 0

```

4. Fonction tri_lexicographique :

```

def tri_lexicographique(liste):
    for i in range(len(liste)-1):
        k = i + 1
        cle = liste[k]
        while k > 0 and ordre_lexicographique(cle, liste[k-1]) == -1:
            liste[k] = liste[k-1]
            k = k - 1
        liste[k] = cle

mots = ['bon', 'toi', 'Bonjour', 'salut', 'oui']
tri_lexicographique(mots)
print(mots)

```

Exercice 8.4

1.

```

def distance2(point):
    x, y = point
    return x * x + y * y

```

2.

```

def compare(p1, p2):
    d1 = distance2(p1)
    d2 = distance2(p2)
    if d1 < d2:
        return -1
    elif d2 < d1:
        return 1
    else:
        return 0

```

3. On utilise par exemple le tri par sélection.

```
def tri_points(liste):
    for i in range(len(liste)-1):
        i_mini = i
        mini = liste[i]
        for j in range(i+1, len(liste)):
            if compare(liste[j], mini) == -1:
                i_mini = j
                mini = liste[j]
        liste[i], liste[i_mini] = liste[i_mini], liste[i]
```

On peut tester par exemple avec la liste `[[3, 5], [2, 1], [0, 7], [-2, 0]]`.

Exercice 8.5

1. Fonction d'insertion :

```
def insertion(liste, n, x):
    if n == 0 or x >= liste[n-1]:
        liste[n] = x
    else:
        liste[n] = liste[n-1]
        insertion(liste, n-1, x)
```

2. Fonction de tri :

```
def tri_insertion_rec(liste, n):
    if n > 1:
        tri_insertion_rec(liste, n-1)
        insertion(liste, n-1, liste[n-1])
```

Pour tester avec une liste `lst`, on écrit l'instruction `tri_insertion_rec(lst, len(lst))`.

Exercice 8.6

```
def pos_insertion(liste, x, droite):
    gauche = 0
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if x < liste[milieu]:
            droite = milieu - 1
        else:
            gauche = milieu + 1
    return gauche
```



```
def tri_insertion(liste):
    for i in range(1, len(liste)):
        k = i # indice de la cle
        cle = liste[k]
        pos = pos_insertion(liste, cle, k)
        for j in range(k - pos):
            liste[k-j] = liste[k-j-1]
        if pos < k:
            liste[pos] = cle
```

Exercice 8.7

Entrée dans la boucle, k a la valeur de g :

				k								
liste	...			5	6	8	9		3	4	8	...
				g							d	
aux	...			5	6	8	9		8	4	3	...
				i							j	

Les valeurs successivement choisies apparaissent en gras dans la liste.
Fin du passage 1 :

				k								
liste	...			3	6	8	9		3	4	8	...
				g							d	
aux	...			5	6	8	9		8	4	3	...
				i						j		

Fin du passage 2 :

					k							
liste	...			3	4	8	9		3	4	8	...
				g							d	
aux	...			5	6	8	9		8	4	3	...
				i						j		

Fin du passage 3 :

						k						
liste	...			3	4	5	9		3	4	8	...
				g							d	
aux	...			5	6	8	9		8	4	3	...
				i						j		

Fin du passage 4 :

							k					
liste	...			3	4	5	6		3	4	8	...
				g							d	
aux	...			5	6	8	9		8	4	3	...
				i						j		

Fin du passage 5 :

liste	...	3	4	5	6	8	4	8	...
		g				k		d	
aux	...	5	6	8	9	8	4	3	...
					i	j			

Fin du passage 6 ($i = j$) :

liste	...	3	4	5	6	8	^k 8	8	...
		g						d	
aux	...	5	6	8	9	8	4	3	...
					j				

Fin du passage 7 :

liste	...	3	4	5	6	8	8	^k 9	...
		g						d	
aux	...	5	6	8	9	8	4	3	...
					j	i			

Exercice 8.8

La variable x prend la valeur 5 et la variable i prend la valeur 0.

En gris clair les éléments inférieurs à x placés à gauche, en gris foncé les éléments supérieurs à x placés à droite.

La variable j prend successivement les valeurs de 0 à 6.

La variable i augmente d'une unité lorsqu'un échange est effectué.

3	8	9	1	6	4	2	5	$3 \leq 5$	$i = 0$	$j = 0$
3	8	9	1	6	4	2	5	$8 > 5$	$i = 1$	$j = 1$
3	8	9	1	6	4	2	5	$9 > 5$	$i = 1$	$j = 2$
3	8	9	1	6	4	2	5	$1 \leq 5$	$i = 1$	$j = 3$
3	1	9	8	6	4	2	5	$6 > 5$	$i = 2$	$j = 4$
3	1	9	8	6	4	2	5	$4 \leq 5$	$i = 2$	$j = 5$
3	1	4	8	6	9	2	5	$2 \leq 5$	$i = 3$	$j = 6$
3	1	4	2	6	9	8	5		$i = 4$	
3	1	4	2	5	9	8	6			

Exercice 8.9

Une fonction `partition` sans échange de valeurs.

On fait varier d'abord l'indice gauche, puis on procède éventuellement à une affectation. La valeur de `liste[d]` est stockée dans la variable `limite`, donc la place est disponible. La place de `liste[g]` est alors libre. Ensuite on passe à l'indice droit et on procède éventuellement à une affectation sur la place d'indice `g` qui a été libérée. Et on continue ainsi de suite.

```
def partition(liste, g, d):
    limite = liste[d]
    while g < d:
        while g < d and liste[g] <= limite:
            g = g + 1
        if g < d:
            liste[d] = liste[g]
            d = d - 1
        while d > g and liste[d] >= limite:
            d = d - 1
        if d > g:
            liste[g] = liste[d]
            g = g + 1
    liste[d] = limite # ou liste[g] = limite (car g = d)
    return d
```


■ Programmes et algorithmes

Chapitre 9

Écriture d'un programme

UNE SCIENTIFIQUE



Augusta Ada Lovelace (1815-1852) était la fille du poète romantique anglais Lord Byron qui l'avait surnommée *la princesse des parallélogrammes*. Elle rencontre le mathématicien anglais Charles Babbage qui venait d'imaginer une machine capable d'effectuer des opérations mathématiques. Certes, cet ancêtre de l'ordinateur n'a jamais fonctionné, mais Ada Lovelace conçoit le premier programme permettant d'automatiser une suite d'opérations.

■ Un peu d'histoire

Vers 1830, Charles Babbage (1792-1871) avait conçu une machine capable de faire mécaniquement les opérations élémentaires grâce à des cartes perforées. C'est pour cette machine qu'Ada Lovelace conçoit, en étroite collaboration avec Babbage, le premier programme permettant de calculer les nombres de Bernoulli.

En 1936, Alan Turing (1912-1954) a eu l'intuition géniale de concevoir une machine abstraite qui formalise la notion de procédé calculable. Quelques années plus tard, apparaissent les premiers ordinateurs dont l'objet essentiel est de calculer, de trier, d'ordonner. Contrairement aux appareils construits jusque-là, les utilisations des ordinateurs se multiplient grâce à de nombreux programmes de plus en plus variés. Afin de faciliter leur conception, des langages sont créés, comme le Fortran ou le Cobol que l'on exécute, aux débuts de l'informatique, à l'aide de fiches perforées.

OBJECTIFS

Il existe différentes manières de programmer mais aussi des pratiques communes à respecter. Écrire un programme demande de suivre quelques règles.

- Assimiler les notions d'instruction, d'expression, d'affectation :
 - ▶ savoir reconnaître et utiliser un type d'instruction ;
 - ▶ comprendre ce qu'est une expression et une affectation en Python ;
 - ▶ aborder la notion d'effet de bord.
- Mesurer l'importance des spécifications et des annotations :
 - ▶ s'habituer à préciser le type des données en entrées et en sortie ;
 - ▶ percevoir l'utilité de commentaires pour mieux comprendre un bloc d'instructions ou un choix de programmation.
- Comprendre le principe d'utilisation d'une assertion :
 - ▶ concevoir des assertions pour vérifier certaines conditions ;
 - ▶ savoir décrypter et gérer un message d'erreur.

Après avoir écrit des programmes et réfléchi au rôle de parties de code, il est important d'étudier le contenu avec précision. Un programme est composé d'instructions et de commentaires.

■ Instructions

■ Expression et affectation

- Une *expression* est utilisée pour calculer une valeur ou pour appeler une procédure (une fonction qui renvoie la valeur `None`).

Une expression est composée de noms (ou identifiants), de nombres, de chaînes de caractères, d'éléments séparés par des signes de ponctuation, entourés de parenthèses, de crochets, d'accolades et d'opérateurs. Une expression a une valeur.

- Une *affectation* est utilisée pour lier un nom à une valeur ou pour modifier un élément d'un objet mutable comme une liste. L'instruction *affecter* est représentée par le signe `=`.

La syntaxe est : `nom = expression` ou `nom[i] = expression`.

On peut aussi écrire : `n1, n2, ..., nk = seq` si `seq` est une séquence à `k` éléments.

En Python, une expression et une affectation sont deux instructions particulières.

■ Instructions

Une *instruction* est un morceau de code minimal qui produit un effet. Une instruction est exécutée par une machine.

Une *instruction simple* peut s'écrire sur une seule ligne. On peut en écrire plusieurs séparées par des points-virgules.

Outre l'expression et l'affectation, quelques instructions simples sont :

- *affirmer* avec `assert`, qui est suivi d'une expression, (les assertions sont précisées plus loin dans ce chapitre) ;

- *renvoyer* avec `return`, qui est suivi d'une expression et s'emploie uniquement dans une fonction, (une expression peut être un `n-uplet` écrit sans parenthèse) ;

- *arrêter* avec `break`, mot isolé qui permet d'arrêter une boucle ;

- *importer* avec `import`, suivi d'un nom de module, (ou de fonction, de constante, appartenant à un module qui est précisé).

Une *instruction composée* est une instruction sur une ligne terminée par deux-points suivie d'une ou plusieurs instructions indentées : on dispose par exemple de `if` (qui peut être suivi de `elif`, de `else`) pour exécuter des instructions selon une condition, de `for` et de `while` pour exécuter des instructions de manière répétée, de `def` pour définir une fonction.

Exceptée les expressions, une instruction n'a pas de valeur. Certaines instructions nécessitent une expression, comme avec le signe `=`, les mots `assert`, `if`, `elif`, `while`, qui sont suivis d'une expression, donc d'une valeur. Ces choix de conception du langage Python ont des conséquences à connaître.

En premier lieu, ces choix permettent d'éviter quelques erreurs de programmation qui pourraient passer inaperçues en première lecture. Si par exemple on écrit `assert x = 1`, ou `while x = 0`, ou `if (assert x > 0)`, ou `return x = 0`, cela provoque une erreur de syntaxe et le programme

s'arrête. Ce sont des erreurs habituelles chez les novices et qui arrivent aussi simplement lorsqu'on veut écrire trop vite un programme.

Si par contre une instruction comme `x = 1` avait une valeur, et si on écrivait `while x = 1` par erreur à la place de `while x == 1`, on pourrait obtenir une boucle infinie (dans ce cas, on finirait par s'en rendre compte). Mais si on écrivait `if (x = 1)` au lieu de `if (x == 1)`, l'exécution du programme pourrait se poursuivre « normalement ». Et un bug peut-être difficile à déceler surviendrait plus tard.

En deuxième lieu, ces choix donnent de la liberté dans l'écriture et permettent des raccourcis. Les mots `if`, `elif`, `while`, `assert` doivent être suivis d'une expression. Cette expression n'a pas forcément une valeur booléenne, mais quelle que soit sa valeur, celle-ci est interprétée comme une valeur booléenne. Un exemple complètement artificiel comme `assert not print("!!!")` ne provoque pas d'erreur. En effet, `print("!!!")` est une expression de valeur `None` et l'expression `not None` a pour valeur `True`. Toute nombre nul et tout conteneur vide est interprété comme une valeur `False`. C'est donc le cas pour `0`, `0.0`, `"`, `[]`, `()`, `{}`, et aussi pour `None`. Toute autre valeur est interprétée comme une valeur `True`.

Cela permet d'écrire par exemple `if x % 2` plutôt que `if x % 2 == 1` pour tester si un nombre `x` est impair, ou `while x` plutôt que `while x != 0` pour poursuivre une boucle tant que `x` n'est pas nul. Mais attention, cela peut conduire à d'autres types de difficultés et donc de bugs potentiels.

■ Effet de bord

On dit qu'une fonction a un *effet de bord* si son exécution modifie quelque chose en dehors de ce qui est défini dans le corps de la fonction, par exemple un de ses paramètres ou une variable globale définie dans le programme.

Un effet de bord, en anglais *side effect*, est un effet secondaire qui peut être désiré ou pas. Une fonction a un objectif principal. Et la suite des instructions écrites pour atteindre cet objectif peut avoir des effets sur les objets manipulés. C'est en particulier le cas avec les fonctions qui manipulent des objets mutables comme les listes. C'est un effet durable qui apparaît avec une fonction autrement que par la valeur renvoyée.

Dans le code qui suit, la première fonction a un effet de bord, pas la seconde.

```
def ajoute(liste, x):
    liste.append(x)

def ajoute(liste, x):
    liste = liste + [x]
    return liste
```

Avec la première fonction, la liste passée en paramètre est modifiée par la méthode `append`. Avec la seconde fonction, l'affectation crée une nouvelle liste (qui est une variable locale).

On peut utiliser l'une ou l'autre de ces fonctions mais en connaissance de cause pour éviter un éventuel bug.

Remarque : une fonction sans effet de bord peut être plus simple à tester.

Dans l'exemple qui suit, on définit deux fonctions `multiplie` qui prennent en paramètres une liste et un nombre.

```
def multiplie(liste, x):
    res = liste
    for i in range(len(res)):
        res[i] = x * res[i]
    return res

def multiplie(liste, x):
    res = liste
    return [x * res[i] for i in range(len(res))]
```

On définit une liste `liste1` et on écrit `liste2 = multiplie(liste1, 3)`. On obtient le même résultat pour `liste2` avec les deux fonctions mais `liste1` est modifiée avec la première fonction.

Remarque : un effet de bord peut être désiré, par exemple quand on utilise une liste ou un dictionnaire pour mémoriser des données.

■ Spécification, annotation

■ Spécification d'une fonction

Une spécification permet d'informer les utilisateurs de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs affichés en cas de mauvaise utilisation. Elle est résumée dans la *docstring*, inscrite au début du corps de la fonction entre des triples guillemets.

```
def nom_de_la_fonction(arguments):
    """ informations sur la fonction, non obligatoires
        mais recommandées """
    corps de la fonction
```

La spécification est destinée à l'utilisateur. Elle n'est pas utilisée par l'interpréteur Python.

Par exemple, si nous souhaitons obtenir des informations sur la fonction nommée `divmod`, nous utilisons la fonction `help` dans l'interpréteur :

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

La fonction `help` affiche la *docstring* inscrite dans le code de la fonction `divmod`. Les opérateurs utilisés sont classiques, ils sont en général utilisés avec des nombres entiers mais peuvent l'être aussi

avec des flottants. Ceci nous permet de savoir le type des variables passées en paramètres. Nous apprenons donc que la fonction renvoie un couple. Les deux éléments de ce couple sont $x//y$ et $x\%y$. Si nous les notons `div` et `mod`, il est précisé que la propriété $\text{div} \cdot y + \text{mod} == x$ est invariante. Les deux valeurs renvoyées sont donc sans aucun doute le quotient et le reste de la division euclidienne des deux nombres x et y . Cette fonction est aussi utilisable avec des flottants.

On peut effectuer un test avec un nombre décimal :

```
>>> x = 7.2
>>> y = 3
>>> div, mod = divmod(x, y)
>>> div
2.0
>>> mod
1.2000000000000002
>>> div*y + mod == x
True
```

Il ne faut pas être surpris de la valeur obtenue pour `mod`. Les calculs avec des flottants sont des calculs approchés. Ceci est précisé au chapitre 12 sur la représentation des nombres en machine.

On peut obtenir des informations sur une fonction d'un module, par exemple le module `math`.

```
>>> import math
>>> help(math.hypot)
Help on built-in function hypot in module math:

hypot(x, y, /)
    Return the Euclidean distance,  $\sqrt{x^2 + y^2}$ .
```

La spécification d'une fonction est écrite, comme l'est un commentaire dans un programme, à l'attention des utilisateurs qui ont besoin de savoir comment l'utiliser. L'objectif est de les éclairer, de les aider à saisir rapidement le rôle d'une ou plusieurs instructions. Un choix pertinent dans les noms de variables et de fonctions participent aussi à la compréhension d'un code.

Voici comment définir une fonction avec sa spécification et quelques annotations :

```
def permute(liste):
    """ liste est de type list
    la fonction permute le premier et le dernier élément
    et renvoie une nouvelle liste
    permute([1, 2, 3, 4]) renvoie [4, 2, 3, 1] """
    n = len(liste)
    copie = liste[:] # une copie superficielle de la liste
    copie[0], copie[n-1] = copie[n-1], copie[0] # permutation des valeurs
    return copie
```

Le texte entre les triples guillemets fournit ici une description de l'entrée, du traitement et du résultat. Ceci constitue, avec le nom de la fonction, sa *signature*. La fonction est enregistrée dans un fichier et si elle a été chargée dans l'interpréteur Python, le texte s'affiche lorsqu'on écrit `help(permute)`.

Les commentaires précédés par le signe `#` n'ont aucune importance sur l'exécution de la fonction. Avec la spécification, ils ne sont pas lus par l'interpréteur Python.

```
>>> permute([1, 2, 3, 4])
[4, 2, 3, 1]
>>> help(permute)
Help on function permute in module __main__:

permute(liste)
    liste est de type list
    la fonction permute le premier et le dernier élément
    et renvoie une nouvelle liste
    permute([1, 2, 3, 4]) renvoie [4, 2, 3, 1]
```

Le message d'information écrit dans le corps de la fonction est bien affiché à l'appel de la fonction `help`. Ce message s'affiche aussi à l'intérieur d'une bulle dans l'éditeur ou l'interpréteur si nous écrivons un appel de la fonction `permute`. Cette bulle apparaît dès que nous ouvrons la parenthèse au moment de l'écriture `permute(`.

Avec cette fonction `permute`, une erreur sur le paramètre provoque l'arrêt de l'exécution de la fonction. Cela se produit si la fonction est appelée avec un nombre ou une chaîne de caractères en paramètre. Mais une fonction peut s'exécuter sans problème même si les paramètres ne sont pas ceux attendus. Les problèmes peuvent survenir plus loin dans le programme et la correction de l'erreur est alors plus difficile.

Prenons un exemple très simple :

```
def calcul(x, y):
    return 2 * (x + y)
```

Cette fonction est programmée pour effectuer un calcul avec deux nombres.

```
>>> calcul(1, 3)
8
```

Si ce résultat doit être utilisé ensuite dans le programme pour un autre calcul, il est capital que ce soit bien un nombre. Mais ici, si les deux paramètres sont de type `str`, la fonction s'exécute sans aucun souci et produit un résultat qui n'est pas du tout du type attendu.


```
>>> calcul('1', '3')
'1313'
```

Il est donc important que les conditions restrictives sur les paramètres soient bien précisées dans la docstring pour que la fonction soit utilisée correctement.

```
def calcul(x, y):
    """ x et y sont du type int ou float """
    return 2 * (x + y)
```

Les exemples sont simples mais il faut bien comprendre qu'une spécification est une sorte de contrat entre l'auteur d'un code et l'utilisateur. L'auteur garantit un résultat sous réserve d'une utilisation correcte qui est précisée.

■ Annotations, commentaires

Un programme doit pouvoir être lu et relu facilement par l'auteur mais aussi par quelqu'un qui découvre le programme. Il est important pour cela d'annoter certaines lignes de code ou des blocs d'instructions afin de préciser leur rôle.

On utilise pour cela un commentaire qui est une ligne de texte précédée du signe `#`. Comme c'est le cas pour une spécification, un commentaire n'est pas utilisé par l'interpréteur Python.

Un choix de structure ou de méthode par exemple peut aussi être précisé et expliqué à l'aide de commentaires. Pourquoi choisir un dictionnaire plutôt qu'une liste, pourquoi une fonction récursive plutôt qu'itérative, etc.

```
def mafonction(...):
    ...
    # utilisation d'un dictionnaire pour stocker les données
    dico = {}
    # on teste si la clé appartient au dictionnaire
    if cle in dico:
        ...
    # si la clé n'appartient pas au dictionnaire, on la crée
    else:
        dico[cle] = val
    ...
```

On peut découper un programme en plusieurs parties en précisant l'utilité de chaque partie.

```
"""description du contenu,
objectifs du programme,
choix de conception"""
```



```
#importation de modules et de fonctions
...

#définitions de constantes, de variables
...

#définitions de fonctions annexes
...

#définition de la fonction principale
...

#exemples d'utilisation et tests
...
```

■ Assertion

Une spécification permet d'éclairer sur les données en entrées, le type des valeurs autorisées, la plage de valeurs acceptées. On peut ajouter des instructions qui vont arrêter le programme en cas de mauvaises utilisation.

Une assertion est l'affirmation qu'une propriété est vraie. Elle est composée du mot **assert** suivi d'une expression dont la valeur est interprétée comme une valeur booléenne. Si l'expression a la valeur **True** il ne se passe rien. Sinon le programme est interrompu et un message d'erreur s'affiche **AssertionError**.

Voici un exemple simple :

```
def inverse(x):
    """x est un nombre non nul de type int ou float
    renvoie l'inverse de x
    """
    assert x != 0
    return 1 / x
```

Utilisation de la fonction :

```
>>> inverse(-3)
-0.3333333333333333
>>> inverse(0)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    inverse(0)
  File "C:/Documents/progs/test.py", line 5, in inverse
    assert x != 0
AssertionError
```

Dans le deuxième exemple ci-dessous, avec la recherche d'un maximum dans une liste de nombres, la liste doit être non vide.

```
def cherche_max(liste):  
    """liste est une liste de nombres non vide  
    renvoie le maximum de la liste  
    """  
    assert liste != []  
    ...
```

Un autre exemple avec une recherche dichotomique pour résoudre une équation $f(x) = 0$ sur un intervalle $[a; b]$:

```
def cherche_dicho(f, a, b, n):  
    """f est une fonction continue qui s'annule entre a et b  
    renvoie la solution à  $10^{-n}$  près  
    """  
    assert f(a) * f(b) <= 0  
    ...
```

Un dernier exemple avec une fonction qui prend en paramètres un dictionnaire et une clé :

```
def fonction(dico, k):  
    """dico est un dictionnaire, k est une clé de ce dictionnaire  
    """  
    assert k in dico  
    ...
```

On contrôle si la valeur passée en paramètre pour exécuter la fonction est bien une clé du dictionnaire. Si ce n'est pas le cas, le programme s'interrompt.

De manière générale, on utilise `assert` pour contrôler une valeur.

■ ■ Vrai/Faux

	Vrai	Faux
1. Les mots <code>if</code> , <code>elif</code> et <code>else</code> sont toujours suivis d'une expression.	☐	☐
2. Les utilisations des mots <code>return</code> et <code>assert</code> comportent une différence de syntaxe, <code>return</code> ne s'écrit que dans une fonction.	☐	☐
3. Si on exécute une fonction sans suivre sa spécification, une erreur se produit.	☐	☐
4. C'est grâce à une spécification que l'interpréteur peut vérifier les paramètres d'une fonction.	☐	☐
5. Trop de commentaires ralentissent l'exécution d'un programme.	☐	☐
6. Un commentaire n'est pas lu par un interpréteur Python.	☐	☐
7. Le mot <code>assert</code> est toujours suivi d'une expression.	☐	☐
8. L'exécution de l'instruction <code>if x > 0 and if x < 5</code> , si <code>x</code> est un entier, ne provoque pas d'erreur.	☐	☐
9. L'instruction <code>assert x = 0</code> permet de contrôler si <code>x</code> vaut 0.	☐	☐
10. Une spécification sert à l'interpréteur pour vérifier le résultat renvoyé par une fonction.	☐	☐

■ ■ Énoncé des exercices

Exercice 9.1 : La fonction définie ci-dessous a-t-elle un effet de bord ? Autrement dit, est-ce que la valeur de la variable `m` est modifiée après l'exécution du code ?

```
m = [[0, 1], [2, 3]]

def f(liste):
    liste = 2 * liste + m
    n = len(liste)
    liste[n-1][0] = 4
    return liste

f([0])
```

Exercice 9.2 : On définit la suite de Collatz ou suite de Syracuse de la manière suivante. Le premier terme est un entier naturel non nul n . Si un terme est pair, le suivant vaut sa moitié, si un terme est impair, le suivant vaut le triple plus un. Mathématiquement, on écrit $u_0 = n$ et $u_{k+1} = u_k/2$ si u_k est pair et $u_{k+1} = 3u_k + 1$ si u_k est impair.

La fonction qui suit prend en paramètre un entier naturel `n` et renvoie la liste des termes de la suite.

```
def syracuse(n):
    liste = [n]
    while n != 1:
        if n % 2:
            n = 3 * n + 1
        else:
            n = n // 2
        liste.append(n)
    return liste
```

Ajouter une première assertion permettant de vérifier que `n` est strictement positif.

Il peut y avoir un problème si la fonction est appelée avec un flottant dont la valeur n'est pas entière. Proposer deux manières d'éviter ce problème, soit par une assertion, soit en modifiant une ligne (unique) du programme.

Exercice 9.3 : La fonction écrite à la suite calcule le n -ième nombre de Fibonacci. Ces nombres notés u_n sont définis par $u_0 = 0$, $u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$ pour tout entier $n > 1$.

Cette fonction a-t-elle un effet de bord ?

```
dico = {0: 0, 1: 1}

def fibo(n):
    for i in range(2, n+1):
        dico[i] = dico[i-2] + dico[i-1]
    return dico[n]
```

Exercice 9.4 : Les deux codes qui suivent sont censés calculer le n-ième nombre de Fibonacci (ces nombres sont définis dans l'exercice précédent).

Commenter ces codes. Écrire un code sans effet de bord qui fournit un résultat correct.

```
liste1 = [0, 1]
def fibo1(n):
    for i in range(2, n+1):
        liste1.append(liste1[i-2] + liste1[i-1])
    return liste1[n]

print(fibo1(3), fibo1(5))

liste2 = [0, 1]
def fibo2(n):
    for i in range(2, n+1):
        liste2.append(liste2[i-2] + liste2[i-1])
    return liste2[len(liste2)-1]

print(fibo2(3), fibo2(5))
```

Exercice 9.5 : Une fonction qui calcule le quotient et le reste de la division de deux entiers naturels est donnée ci-dessous. Sans modifier les instructions de cette fonction écrire sa spécification. Compléter avec une assertion permettant de vérifier l'entrée et une assertion permettant de vérifier une propriété invariante de la boucle.

```
def divise(m, n):
    q, r = 0, m
    while r >= n:
        r = r - n
        q = q + 1
    return q, r
```

Exercice 9.6 : Écrire une fonction `ajoute` qui prend en paramètres deux listes de nombres de même longueur et renvoie une nouvelle liste construite en effectuant la somme terme à terme des éléments des deux listes. Utiliser l'instruction `assert` pour vérifier les longueurs des listes.

Exercice 9.7 : La fonction `moyennes` prend en paramètre une liste de nombres qui est non vide. Utiliser les deux commentaires pour corriger le code de cette fonction.

```
def moyennes(liste):
    n = len(liste)
    # initialisation d'une liste lisse avec la 1ère valeur de liste
    lisse = liste[0]
    for i in range(n-1):
        # moyenne de deux éléments consécutifs
        m = (liste[i] + liste[i+1] / 2)
        lisse.append(m)
    return lisse
```

Exercice 9.8 : La fonction `lissage` prend en paramètre une liste de nombres qui est non vide. Utiliser le commentaire pour corriger le code de cette fonction.

```
def lissage(liste):
    n = len(liste)
    lisse = [liste[0]]
    for i in range(1, n-1):
        # ajoute à la liste la moyenne de trois éléments consécutifs
        lisse.append(sum(liste[i-1:i+1])/3)
    lisse.append(liste[n-1])
    return lisse
```

Exercice 9.9 : La fonction qui suit prend en paramètre un entier `n` et doit renvoyer `True` si `n` est un nombre pair et `False` sinon.

```
def pair(n):
    """ n est un entier
    renvoie True si n est pair, sinon renvoie False"""
    if n % 2:
        return True
    else:
        return False
```

En utilisant cette fonction, on obtient exactement le résultat contraire à celui qui est prévu. Expliquer pourquoi et écrire une fonction qui satisfait à la spécification.

Exercice 9.10 : Le programme qui suit contient deux fonctions `double1` et `double2`. Ces fonctions prennent une liste en paramètre et leur rôle doit être de concaténer la liste avec elle-même. Si une liste vaut `[1, 2]`, après l'exécution d'une fonction la liste doit avoir pour valeur `[1, 2, 1, 2]`.


```
def double1(liste):
    for i in range(len(liste)):
        liste.append(liste[i])

def double2(liste):
    i = 0
    while i < len(liste):
        liste.append(liste[i])
        i = i + 1
```

1. L'une des deux fonctions ne remplit pas son rôle et ne termine pas. Expliquer la non-terminaison.
2. Modifier le code de la fonction qui ne termine pas en gardant le même style de boucle afin qu'elle remplisse son rôle correctement.

Exercice 9.11 : La fonction `f` prend en paramètre un nombre (de type `int` ou `float`).

```
def f(x):
    return 2 * x, 3 * x, 5 * x
```

Quel est le type du résultat renvoyé par la fonction `f` ? Après l'instruction `a, b, c = f(2)`, quelle est la valeur de `c` ?

■ ■ Indications

-
- Ex. 9.1** _____
Le comportement serait identique avec `liste = m` à la place de `liste = 2 * liste + m`.
-
- Ex. 9.4** _____
Les deux listes `liste1` et `liste2` sont des variables globales donc modifiables par une fonction.
-
- Ex. 9.5** _____
Pour trouver une propriété invariante, examiner la définition de la division euclidienne.
-
- Ex. 9.8** _____
Il est habituel en Python, lorsque deux bornes sont précisées, que la borne gauche est incluse et la borne droite est exclue.

■ ■ Corrigé du vrai/faux

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

Quelques explications

1. C'est faux pour le mot `else`.
8. On ne peut pas avoir un `and` logique entre deux instructions (`if`) qui ne sont pas des expressions.
9. Le mot `assert` est suivi d'une expression. Donc il faut écrire `assert x == 0`.

❑ Erreurs classiques et conseils.—

- Les notions d'instruction et d'expression doivent être parfaitement maîtrisées.
- Pour reconnaître une expression, il faut se poser la question de la valeur en se souvenant que même `None` est une valeur.
- Les spécifications et commentaires divers ne sont pas lus par l'interpréteur. Ils n'ont aucune influence sur l'exécution d'un programme mais sont importants pour l'utilisateur.

■ ■ Corrigé des exercices

Exercice 9.1

Avec l'appel `f([0])`, la variable locale `liste` prend la valeur `[0]`.

Avec l'affectation `liste = 2 * liste + m`, une nouvelle liste est créée. La liste `m` n'est pas copiée, et le comportement qui suit est identique à celui que l'on obtiendrait avec une affectation `liste = m`. Donc la modification du dernier élément de `liste` modifie le dernier élément de `m`.

Exercice 9.2

Un nombre `n` a une valeur entière si `n % 2` vaut 0 ou 1. C'est l'objet de la deuxième assertion.

```
def syracuse2(n):
    assert n > 0
    assert n % 2 == 0 or n % 2 == 1
    liste = [n]
    while n != 1:
        if n % 2:
            n = 3 * n + 1
        else:
            n = n // 2
        liste.append(n)
    return liste
```

On peut gérer le problème de valeur entière en modifiant l'instruction `if n % 2`. En effet ce test est passé pour tout flottant qui n'est pas pair. Il vaut mieux écrire à la place `if n % 2 == 1`.

Exercice 9.3

La variable `dico` peut être modifiée à chaque appel de la fonction `fibonacci`. Cependant grâce aux propriétés d'un dictionnaire, cette variable ne contient que les nombres de Fibonacci calculés par l'exécution de la fonction et ne contient chaque nombre calculé qu'une seule fois.

Exercice 9.4

Les variables `liste1` et `liste2` sont modifiées à chaque appel des fonctions. Dans les deux cas, la première utilisation de la fonction produit un résultat exact. Mais avec deux appels consécutifs comme `fibonacci(3)` puis `fibonacci(5)`, ou comme `fibonacci2(3)` puis `fibonacci2(5)`, le deuxième résultat est incorrect.

Pour obtenir un code sans effet de bord, la liste utilisée doit être définie dans le code de la fonction. Voici un exemple de code :

```
def fibonacci(n):
    liste = [0, 1]
    for i in range(2, n+1):
        liste.append(liste[i-2] + liste[i-1])
    return liste[n]
```

Exercice 9.5

On peut constater, avec quelques tests éventuellement, que le paramètre `n` doit être non nul. On place donc l'assertion `assert n != 0` au début du code.

La division euclidienne de m par n est définie par deux conditions : $m = nq + r$ et $0 \leq r < n$. Une propriété invariante de la boucle est « $m = nq + r$ ».

On peut donc placer une assertion `assert m == n * q + r` dans la boucle, après le calcul des nouvelles valeurs des deux variables `r` et `q`.

```
def divise(m, n):
    """m et n sont deux entiers naturels, n est non nul,
    renvoie le quotient et le reste de la division de m par n
    propriété invariante: m == n * q + r"""
    assert n != 0
    q, r = 0, m
    while r >= n:
        r = r - n
        q = q + 1
        assert m == n * q + r
    return q, r
```

Exercice 9.6

Il suffit d'obtenir les longueurs des deux listes avec la fonction `len` et de les comparer.

```
def ajoute(liste1, liste2):
    n1 = len(liste1)
    n2 = len(liste2)
    assert n1 == n2
    return [liste1[i] + liste2[i] for i in range(n1)]
```

Exercice 9.7

Le premier commentaire précise que `lisse` est une liste. Or, la valeur affectée à `lisse` dans le code est un nombre. Donc l'initialisation doit être remplacée par : `lisse = [liste[0]]`.

Le deuxième commentaire précise que `m` est la moyenne de deux termes consécutifs. Il y a donc une erreur dans le parenthésage. Écrire `m = (liste[i] + liste[i+1]) / 2`.

Exercice 9.8

L'expression `liste[i-1:i+1]` doit être une liste contenant trois éléments consécutifs. Or cette liste ne contient que deux éléments : `liste[i-1]` et `liste[i]`.

Il faut donc remplacer cette expression par `liste[i-1:i+2]`.

Exercice 9.9

Le test `if x % 2` n'est pas vérifié si `x` est pair puisque l'expression `x % 2` a pour valeur 0 dans ce cas, (c'est le reste de la division), et le nombre 0 est interprété comme `False`.

Pour obtenir un code correct, il suffit de permuter les valeurs renvoyées `True` et `False`. Une autre manière de corriger le code est de modifier le test en écrivant à la place : `if x % 2 == 0`.

Exercice 9.10

On peut tester les fonctions sur machine.

1. Avec les deux fonctions, les listes sont modifiées (en place). La deuxième fonction pose problème car la longueur de la liste est réévaluée à chaque passage dans la boucle. La première fonction ne pose aucun problème.

```
>>> liste = [1, 2]
>>> double1(liste)
>>> liste
[1, 2, 1, 2]
```

Avec la fonction `double2`, le programme ne termine pas. On peut arrêter l'exécution en cours en appuyant simultanément sur les touches `ctrl` et `C`.

2. Pour que la fonction termine, il faut éviter de réévaluer la longueur de la liste à chaque passage dans la boucle. Pour cela, on l'évalue uniquement avant l'entrée dans la boucle.

```
def double3(liste):
    i = 0
    n = len(liste)
    while i < n:
        liste.append(liste[i])
        i = i + 1
```

Test de la nouvelle fonction :

```
>>> liste = [1, 2]
>>> double3(liste)
>>> liste
[1, 2, 1, 2]
```

Exercice 9.11

Le résultat renvoyé par la fonction `f` est de type tuple.

En effet, ce sont les virgules qui définissent un tuple. Les parenthèses peuvent être utilisées pour des questions de priorité ou de lisibilité.

Le résultat `f(2)` est le tuple `(4, 6, 10)`, donc `c` prend la valeur 10.

Analyse d'un programme

UN SCIENTIFIQUE



Après des études de chimie, puis de médecine, **John Backus** (1924-2007) devient technicien radio. Ce n'est que tardivement qu'il s'intéresse aux mathématiques puis entre à IBM en 1950. En novembre 1954, il présente le langage Fortran qu'il a élaboré pour simplifier la tâche des programmeurs. Par la suite, il participe à l'élaboration du langage de programmation ALGOL puis poursuit le développement de langages formels. Ses travaux sont récompensés par le prix Turing obtenu en 1977.

■ Un peu d'histoire

Lors des tout débuts de l'informatique, dans l'immédiat après-guerre, on travaillait directement en langage machine ce qui demandait une connaissance technique très spécifique. Le besoin s'est fait sentir de faciliter la tâche du programmeur en créant un langage intermédiaire.

Arrivé à IBM en 1950, John Backus est confronté à la difficulté de noter les nombres. Il élabore la syntaxe et le fonctionnement d'un langage intermédiaire entre le programmeur et la machine et conçoit le Fortran.

De très nombreux langages suivirent très rapidement comme le COBOL en 1959. Dans les années 1970 apparaît la programmation structurée et les langages de programmation s'adaptent aux différents besoins qui apparaissent. Ils acquièrent alors de nouvelles potentialités grâce à la vitesse toujours plus grande des processeurs. Le développement d'Internet nécessite la création de nouveaux langages spécifiques.

OBJECTIFS

L'analyse d'un algorithme permet de s'assurer de sa terminaison et de sa correction. Des jeux de tests permettent de contrôler l'exécution dans des cas particuliers. La complexité en temps et en mémoire de l'algorithme doit être évaluée.

- Savoir prouver la terminaison et la correction d'un algorithme simple :
 - ▶ distinguer clairement les notions de variant et d'invariant ;
 - ▶ utiliser un variant de boucle pour prouver la terminaison ;
 - ▶ utiliser un invariant de boucle pour prouver la correction.
- Écrire et utiliser un jeu de tests :
 - ▶ être capable de repérer des cas simples à tester ;
 - ▶ distinguer et traduire les cas limites ;
 - ▶ concevoir des partitions d'ensemble de données pour élaborer un jeu de tests.
- Acquérir des compétences sur la notion de complexité :
 - ▶ connaître quelques ordres de grandeur de complexité d'algorithmes classiques ;
 - ▶ distinguer les notions de complexité en temps et de complexité en espace ;
 - ▶ savoir étudier la complexité en temps ou en espace dans des cas simples.

L'écriture d'un programme est étudiée au chapitre précédent. Le contenu doit être analysé afin de vérifier que le résultat obtenu lors de son exécution est celui attendu.

Un programme peut être exécuté seul ou utilisé par un autre programme et dans ce cas il est une donnée d'entrée de cet autre programme. Il doit produire en sortie le résultat attendu d'après la spécification quelles que soient les données en entrées pour ne pas provoquer des erreurs en chaînes qui pourraient être catastrophiques.

Les problèmes de sécurité, (données personnelles, systèmes sensibles comme un hôpital), et de sûreté (éviter des pannes aux conséquences parfois dramatiques), sont devenus des questions très importantes dans notre société de plus en plus connectée. Il est primordial que les programmes fonctionnent correctement.

■ Validité d'un algorithme

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme produit un résultat après un nombre fini d'étapes et que ce résultat est correct dans le sens où il est conforme à la spécification précisée. Nous disons alors que l'algorithme est valide.

Un algorithme itératif est construit avec des boucles. Le nombre de passages dans une boucle doit être fini. Si l'algorithme est récursif, le nombre d'appels récursifs doit être fini.

Deux conditions sont donc à vérifier :

- l'algorithme donne une réponse, c'est l'étude de la *terminaison* ;
- la réponse donnée est celle attendue, c'est l'étude de la *correction*.

Si les deux conditions sont satisfaites, nous disons que l'algorithme est *valide*.

Si lorsqu'il termine, l'algorithme donne la réponse attendue on parle de *correction partielle*. Si la terminaison est assurée dans tous les cas et que la réponse est correcte, on parle de *correction totale*.

Pour prouver la terminaison d'un algorithme itératif, nous disposons de la notion de variant de boucle. Pour prouver qu'un algorithme itératif est correct, nous disposons de la notion d'invariant de boucle.

■ Terminaison

Méthode

Dans le cas de boucles non conditionnelles, le nombre d'étapes est déterminé. Nous parlons donc de boucles conditionnelles. Pour prouver la terminaison, nous exhibons une expression, qui peut être une simple variable, dont les valeurs prises au cours des itérations constituent une suite qui converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt de la boucle. Cette expression se nomme *un variant de boucle*.

Considérons par exemple le code suivant où la valeur de la variable `a` est un nombre quelconque :

```
x = 0
while x ** 2 < a:
    x = x + 1
```


Si la valeur de a est négative ou nulle, il n'y a aucun passage dans la boucle. Sinon, la suite des valeurs de la variable x est $0, 1, 2, \dots, n$, où n est la première valeur dont le carré est supérieur ou égal à la valeur de a . Le nombre de passages dans la boucle est donc fini.

Voici un deuxième exemple où les variables a et b sont des entiers naturels.

```
m = 0
p = 0
while m < a:
    m = m + 1
    p = p + b
```

Dans cet exemple, nous choisissons comme variant la variable m . Cette variable prend pour valeurs successives $0, 1, \dots, a$ et il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

■ Correction

Définition

Un *invariant d'une boucle* est une propriété qui est vérifiée avant l'entrée dans une boucle et après chaque passage dans cette boucle.

Pour démontrer qu'une propriété est un invariant d'une boucle, on commence donc par vérifier que la propriété est vraie avant l'entrée dans la boucle. On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. On peut alors conclure, si la terminaison est assurée, que la propriété est vraie à la sortie de la boucle.

Ceci traduit le fait que la boucle réalise bien la tâche souhaitée.

Exemple

Reprenons l'algorithme de calcul avec une boucle conditionnelle présenté plus haut :

```
m = 0
p = 0
while m < a:
    m = m + 1
    p = p + b
```

Notons m et p les valeurs des variables m et p .

Nous allons montrer que la propriété $\ll p = m \times b \gg$ est un invariant de la boucle.

Avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m \times b$ est vraie.

Supposons que $p = m \times b$ avant un passage dans la boucle. Les nouvelles valeurs de m et p après le passage, notées m' et p' , vérifient : $m' = m + 1$ et $p' = p + b$. Alors $p' = m \times b + b = (m + 1) \times b = m' \times b$. La propriété est donc vraie après ce passage dans la boucle.

La terminaison a été prouvée précédemment et nous avons en sortie de boucle $p = m \times b$. Or, à la sortie de la boucle, la variable m a pour valeur celle de a . Nous avons finalement obtenu à la fin du programme le produit $p = a \times b$.

■ Tests d'un programme

■ Bugs

Un circuit électronique, et plus généralement une machine, fait ce qu'on lui demande de faire. Si une erreur est commise lors de l'exécution d'un programme, alors cette erreur est d'une certaine manière écrite dans le programme. De plus, ce qui peut sembler être une « petite erreur » peut avoir des conséquences très importantes.

Considérons le programme qui suit.

```
>>> def f(x, y):
    return (x * y) ** 0.5

>>> def g(x, y):
    return (x ** 0.5) * (y ** 0.5)

>>> g(1e152, 1e152) / f(1e152, 1e152)
1.0
>>> g(1e155, 1e155) / f(1e155, 1e155)
0.0
```

En mathématiques, si x et y sont positifs : $\sqrt{xy} = \sqrt{x}\sqrt{y}$.

Le premier résultat est donc celui attendu, pas le second. Ce simple exemple montre la nécessité de tester un programme. Il prouve que l'un des deux programmes n'est pas correct (mais ne prouve pas que l'autre est correct).

Le même type de bug peut être présent avec un algorithme bien connu, celui d'une recherche dichotomique, lorsqu'on écrit `m = (g + d) // 2` pour obtenir l'indice du milieu des indices gauche et droite. Même si les valeurs de `g` et `d` appartiennent à la plage de valeurs autorisées par le langage et la machine, la valeur de `g + d` peut dépasser la valeur maximale autorisée. Ce qui entraîne une erreur de dépassement ou d'overflow. Ce type de bug peut se trouver aussi dans des algorithmes qui suivent une stratégie diviser pour régner, basé sur la dichotomie.

En Python, si on utilise des flottants et une instruction comme `int((a+b)/2)`, on risque un dépassement. Par exemple `(10.0**308 + 50.0**308)/2` produit une erreur.

La solution pour éviter ce type de bug est d'écrire `m = g + (d - g) // 2`.

■ Jeu de tests

Les programmes que nous écrivons ne fonctionnent pas toujours au premier essai comme nous le prévoyons.

- ▶ Des tests permettent d'observer le comportement d'un programme. Est-ce qu'il produit le résultat attendu dans un cas précis ?
- ▶ Le débogage consiste à corriger un programme lorsque nous savons qu'il ne fonctionne pas correctement.

Afin de rendre les tests et le débogage plus simples, il est important d'y penser lors de l'écriture d'un programme. Une bonne attitude est de décomposer le programme en éléments qui peuvent

chacun être testés et débogués indépendamment des autres. L'utilisation de fonctions est une bonne habitude. La documentation des programmes, l'écriture de spécifications et l'utilisation d'assertions participent à l'élaboration de tests pertinents.

Des tests peuvent permettre de faire apparaître différents types d'erreurs.

- ▶ une erreur survient lors de certains test et pas avec d'autres, c'est un bug intermittent ;
- ▶ une erreur survient pour chaque test, c'est un bug permanent.
- ▶ le programme s'arrête de manière prématurée ou ne s'arrête pas, le bug est visible ;

Les message d'erreurs affichés lors de l'exécution d'un programme permettent d'effectuer des corrections. Les deux dernières lignes du message donnent l'instruction fautive et le type d'erreur.

Quand le programme s'exécute entièrement, il est possible de vérifier que le résultat obtenu est celui qui est attendu en ajoutant au programme des assertions. Lorsqu'une assertion n'est pas vérifiée, par exemple `assert f(3) == 0`, le message `AssertionError` est affiché.

Notons que la fonction `affiche` écrite ci-dessous s'exécute sans signaler de problème mais ne termine pas. C'est l'étude de la terminaison qui permet de le savoir. L'étude de la complexité présentée dans la section suivante permet elle d'évaluer le temps d'exécution.

```
def affiche(i):  
    while True:  
        print(i)  
        i = i + 1
```

Lorsque des tests, aussi nombreux soient-ils, ont été effectués avec succès, il reste encore la possibilité d'avoir un bug caché avec un programme qui produit un résultat faux dans des cas qui n'ont pas été testés.

■ Construire un jeu de tests

Construire un jeu de tests consiste à définir un ensemble de données qui vont être utilisées pour vérifier que le programme produit bien les résultats attendus avec ces données. Ces vérifications peuvent s'effectuer de plusieurs manières. On peut utiliser la fonction `print` pour afficher quelques réponses et vérifier visuellement qu'elles sont correctes. Pour un nombre conséquent de tests, il est plus pratique d'utiliser des assertions. Un message d'erreur est affiché seulement pour les cas qui posent problème.

Dans la construction d'un jeu de tests, on distingue plusieurs types de tests.

- ▶ Tester quelques cas simples typiques (pour une utilisation basique du programme).
- ▶ Tester des valeurs extrêmes, des cas limites, des cas interdits.
- ▶ Tester un nombre important de données (choisies de manière aléatoire par exemple).
- ▶ Tester des cas qui pourraient nécessiter un temps d'exécution important afin de pouvoir évaluer l'efficacité du programme.

Pour effectuer ces tests, on partitionne le domaine d'entrée. On utilise ensuite une donnée, représentante de chaque partie, pour effectuer un test en vérifiant la sortie.

Par exemple avec une fonction qui renvoie le pgcd de deux entiers naturels m et n , l'ensemble des données acceptables en entrées est l'ensemble des couples d'entiers naturels. On peut distinguer les cas où $m > n$, $m < n$, $m = n$, mais aussi les cas où m est divisible par n ou n divisible par m , les cas où m et n sont premiers entre eux. Il y a aussi les cas limites, soit l'un des deux entiers est nul ou les deux entiers sont nuls. Les tests peuvent alors être effectués avec par exemple les couples (16, 12), (9, 15), (8, 8), (12, 6), (10, 30), (15, 14), (4, 0), (0, 4), (0, 0).

On écrit donc des instructions comme `assert pgcd(16, 12) == 4`.

Avec une fonction qui prend une liste en entrée, on teste naturellement le cas d'une liste vide et le cas d'une liste à un élément. En général, lorsqu'on commence à écrire des tests, des cas auxquels on ne pense pas immédiatement viennent peu à peu à l'esprit.

■ Exemples

Considérons la fonction `permute` définie ci-dessous avec sa spécification et des commentaires :

```
def permute(liste):
    """ liste est de type list
    la fonction permute le premier et le dernier élément
    et renvoie une nouvelle liste
    permute([1, 2, 3, 4]) renvoie [4, 2, 3, 1] """
    copie = liste[:] # une copie superficielle de la liste
    n = len(copie)
    copie[0], copie[n-1] = copie[n-1], copie[0] # permutation des valeurs
    return copie
```

Pour écrire un jeu tests, nous considérons quelques cas typiques ; une simple liste de nombres, une liste dont les éléments sont des listes, une liste qui contient deux éléments ou un unique élément, et une liste vide.

```
assert permute([1, 2, 3, 4]) == [4, 2, 3, 1]
assert permute([[1, 2], [3, 4], [5, 6]]) == [[5, 6], [3, 4], [1, 2]]
assert permute([1, 2]) == [2, 1]
assert permute([1]) == [1]
assert permute([]) == []
```

En exécutant le programme, nous remarquons que la fonction effectue bien ce qui est prévu même pour une liste ne contenant qu'un seul élément. Par contre, le cas d'une liste vide n'est pas traité et une erreur interrompt le programme.

Nous complétons donc le code de la fonction en ajoutant `if liste == []: return []` au début du code de la fonction. Avec ce code, tous les tests sont passés sans problème.

Les tests ont donc permis d'améliorer le comportement de la fonction en ajoutant la gestion d'un cas d'utilisation qui n'avait pas été prévu au départ.

Il peut être intéressant d'écrire une fonction `test` contenant une batterie de tests.

Pour un exemple nous considérons la division euclidienne.


```
def division(a, b):
    """a est un entier naturel
    b est un entier naturel non nul"""
    r = a
    q = 0
    while r >= b:
        r = r - b
        q = q + 1
    return q, r
```

Tous les types de tests envisageables ne sont pas explorés ici. La fonction écrite ci-dessous permet simplement de vérifier que le résultat renvoyé par la fonction `division` est correct dans une série de cas.

```
def test_division():
    """ la fonction division renvoie le quotient q et le reste r
    dans la division euclidienne de a par b
    un invariant est : a == b * q + r """
    for a in range(13):
        for b in range(1, 13):
            q, r = division(a, b)
            assert a == b * q + r and 0 <= r < b
```

Lors de l'exécution de la fonction `test_division`, aucun problème n'est signalé. Si par erreur il était écrit `while r > b` à la place de `while r >= b` dans la fonction `division`, la fonction de test révélerait cette erreur avec une `AssertionError`.

De manière générale, si un test n'est pas réussi, la fonction ne précise pas quelle erreur a été commise mais elle informe que le test a échoué avec une `AssertionError`.

Les tests présentés sont très simples. Le plus important est d'avoir toujours en tête que même avec un grand nombre de tests variés, l'objectif n'est pas de prouver qu'il n'y a aucune erreur mais d'en trouver.

Considérons une fonction avec pour arguments deux nombres. Il n'est pas question de tester tous les couples de nombres. Il s'agit de déterminer un ensemble de couples qui ont une forte probabilité de provoquer une erreur. Pour cela, nous construisons une partition de l'ensemble de tous les couples suivant leurs caractéristiques : deux nombres entiers, deux nombres flottants, un entier et un flottant. Chacun de ses trois groupes peut encore être divisé : deux nombres positifs, deux négatifs, deux nombres de signes opposés. On peut aussi envisager les cas de deux nombres nuls ou d'un nombre nul et l'autre non nul. L'idée est qu'il suffit de tester la fonction avec un représentant de chacun des groupes constitués en admettant que la fonction aura le même comportement avec deux couples d'un même groupe.

La construction des tests est liée à la spécification, aux conditions sur les arguments et les résultats. Mais l'étude du code permet aussi de compléter ces tests. Si, dans le cas évoqué ci-

dessus, le code différencie les cas où les nombres entiers sont pairs ou impairs, les tests devront en tenir compte. Il faudra encore partager les groupes en fonction de la parité des nombres entiers.

Dans l'exemple qui suit, une fonction prend en argument deux listes de même longueur. La deuxième liste est modifiée et contient à la fin les mêmes éléments que la première liste aux mêmes places s'ils sont positifs et des zéros sinon. On dispose de deux versions :

```
def modif1(liste1, liste2):
    for i in range(len(liste1)):
        if liste1[i] >= 0:
            liste2[i] = liste1[i]
        else:
            liste2[i] = 0

def modif2(liste1, liste2):
    for i in range(len(liste2)):
        liste2[i] = 0
    for i in range(len(liste2)):
        if liste1[i] >= 0:
            liste2[i] = liste1[i]
```

On teste les deux fonctions sur un exemple simple :

```
liste1 = [2, -3, 5, -1]
liste2 = [1, 2, 3, 4]
modif1(liste1, liste2)
assert liste2 == [2, 0, 5, 0]
modif2(liste1, liste2)
assert liste2 == [2, 0, 5, 0]
```

L'observation du code nous apprend que la deuxième liste est modifiée en utilisant la notation indicielle. Il est donc capital de tester le cas où les deux paramètres se réfèrent à la même liste.

```
modif1(liste1, liste1)
assert liste1 == [2, 0, 5, 0]
modif2(liste1, liste1)
assert liste1 == [2, 0, 5, 0]
```

La dernière assertion n'est pas vérifiée (la valeur de `liste1` est `[0, 0, 0, 0]`). On en déduit que la fonction `modif2` n'est pas correcte.

Un algorithme classique, la recherche dichotomique dans une liste triée, pose souvent quelques difficultés de programmation. Pour écrire un jeu de tests, il faut envisager différents types de cas :

- un élément qui appartient à la liste : 2 et `[1, 2, 3, 4]` ;

► un élément qui n'appartient pas à la liste : 0 et [1, 2, 3, 4], 5 et [1, 2, 3, 4], 2.5 et [1, 2, 3, 4].

On regarde ensuite des cas extrêmes, par exemple en lien avec la taille de la liste :

► 5 et [], 5 et [5], 0 et [5].

On teste des cas en fonction de la place de l'élément cherché, en premier, en dernier, au milieu :

► 1 et [1, 2, 3, 4], 4 et [1, 2, 3, 4], 2 et [1, 2, 3].

On peut ensuite effectuer des tests avec de très grandes listes.

Le calcul du milieu ne doit pas être négligé. Le résultat doit être correct même avec de très grands nombres. On peut envisager de séparer le calcul du milieu en créant une nouvelle fonction, ce qui produit un code encore plus lisible et plus facile à tester.

■ Complexité d'un algorithme

Lorsqu'un algorithme est correct, il doit encore, avant d'être programmé et exécuté, satisfaire à deux impératifs en terme de consommation de ressources :

- utiliser un espace en mémoire acceptable, on parle de complexité en espace ;
- produire la réponse attendue en un temps acceptable, on parle de complexité temporelle.

Nous parlons de la complexité d'un algorithme ou de son coût.

La complexité (ou le coût) en espace correspond aux tailles des variables utilisées. Elle intervient par exemple dans la manipulation d'images au chapitre 7 où on recherche pour les algorithmes une complexité constante, c'est-à-dire une écriture d'éléments en mémoire qui utilise un espace constant et réduit autant que possible. Diverses parties d'une machine sont utilisées pour la mémoire et plus les données à conserver ont une taille importante, plus elles sont stockées sur des parties éloignées du processeur avec un temps d'écriture et de lecture qui s'allonge.

Il faut être vigilant avec des algorithmes de tri pour lesquels de nouvelles listes sont construites à chaque appel récursif. Chaque fois que de nouvelles listes sont créées ou que des permutations d'éléments utilisent de l'espace mémoire, on perd en temps d'exécution une partie du bénéfice obtenu par l'algorithme sur le nombre d'opérations à effectuer.

Étudier la complexité temporelle consiste à évaluer le temps d'exécution d'un algorithme en fonction de la taille des données en entrée. Diverses études ont déjà été menées dans les chapitres précédents.

Pour conduire ces études, on évalue un ordre de grandeur du nombre d'opérations élémentaires contenues dans l'algorithme (affectations, opérations arithmétiques, opérations logiques, comparaisons) en supposant qu'elles ont le même temps d'exécution constant. C'est largement suffisant en première analyse. Pour un même problème, ce nombre d'opérations peut être différent suivant les cas. Par exemple trier une liste déjà triée est un cas à priori favorable. Il est important d'étudier le cas le moins favorable, le pire qui puisse se produire, celui qui coûte le plus cher.

Si une machine effectue 10^8 opérations en une seconde, la question n'est pas de savoir si un algorithme effectue $10^8 + 1$ ou $10^8 - 1$ opérations. La question est plutôt d'évaluer un ordre de grandeur du taux d'accroissement du nombre d'opérations en fonction de la taille n des données par exemple pour $n > 10^8$. Si n est multiplié par 2, est-ce que le nombre d'opérations est environ 2 fois plus grand, 4 fois plus grand, 8 fois plus grand ou pratiquement identique ?

Avec 10^8 opérations par seconde, si un programme comporte environ 10^{11} opérations et qu'il doit être exécuté mille fois, il faudra attendre la réponse pendant plus de dix jours. Il est donc important de pouvoir choisir l'algorithme le plus « performant ».

■ Temps d'exécution

Le temps d'exécution d'un programme dépend de la machine, du langage de programmation, de l'algorithme. La part de l'algorithme est obtenue par une évaluation de sa complexité temporelle.

Nous supposons que les temps d'exécution d'une affectation, d'une opération arithmétique simple, d'une comparaison sont pratiquement identiques et constituent une unité de base.

Nous posons les règles suivantes :

- le temps d'exécution d'une suite d'instructions est la somme des temps d'exécution de chaque instruction ;

- le temps d'exécution d'une instruction conditionnelle

si test :

instructions1

sinon :

instructions2

est inférieur ou égal au maximum des temps d'exécution de instructions1 et instructions2, plus le temps d'exécution du test ;

- le temps d'exécution d'une boucle

pour i variant de 1 à p :

instructions

est p fois le temps d'exécution de instructions, si ce temps est constant, plus p (le nombre d'affectations pour la variable i). Si le temps d'exécution de instructions dépend de la valeur de p, l'étude se mène au cas par cas.

Pour une boucle tant que, l'étude se mène aussi au cas par cas.

L'évaluation du temps d'exécution d'un algorithme se réduit ainsi à une évaluation en fonction d'un nombre n , (entier représentant la taille des données en entrée), du nombre total d'opérations élémentaires noté u_n . Le niveau de complexité correspond au type de croissance de la suite (u_n) .

Suivant les valeurs de l'entrée, u_n peut prendre des valeurs très différentes. Si, par exemple, nous parcourons une liste à l'aide d'une boucle, à la recherche d'un élément, celui-ci peut se trouver en premier et nous sortons de la boucle, c'est le cas le plus favorable. Il peut se trouver à la fin de la liste, c'est le pire des cas.

■ Niveaux de complexité

Notation

Considérons deux suites (u_n) et (v_n) .

Si pour tout n assez grand, u_n est majorée par v_n à une constante près, on écrit $u_n \in \mathcal{O}(v_n)$. On dit que u_n est de l'ordre de grandeur de v_n .

Mathématiquement, on écrit : $u_n \in \mathcal{O}(v_n)$ si $\exists N \in \mathbb{N}, \exists k \in \mathbb{R}, \forall n > N, u_n \leq kv_n$.

En pratique, on utilise souvent la propriété suivante : si $\lim_{n \rightarrow +\infty} \frac{u_n}{v_n} = \ell$ alors $u_n \in \mathcal{O}(v_n)$.

Autrement dit, on vérifie si $\frac{u_n}{v_n}$ converge vers une constante.

- **Complexité constante** : $u_n \in \mathcal{O}(1)$. Le temps d'exécution est borné (indépendant de n). C'est le cas, par exemple, pour obtenir le premier élément d'une liste.

- **Complexité logarithmique** : $u_n \in \mathcal{O}(\log_2 n)$. On double le temps d'exécution, en élevant au carré la taille des données. C'est le cas avec la recherche dichotomique dans une liste triée.

Les algorithmes de complexité logarithmiques sont très performants. Les ordres de grandeur $\log_2 n$, $\log_{10} n$ et $\ln n$ sont comparables.

- **Complexité linéaire** : $u_n \in \mathcal{O}(n)$. Cet ordre de grandeur peut s'obtenir avec une boucle non conditionnelle. Par exemple, le calcul de la somme ou de la moyenne de n termes, la recherche séquentielle dans une liste non triée de longueur n , ont une complexité en $\mathcal{O}(n)$.

Soit n la taille d'une donnée. On obtient une complexité linéaire en n si le nombre d'opérations à effectuer peut s'écrire sous la forme $\alpha n + \beta$, avec α et β réels, $\alpha > 0$.

- **Complexité log-linéaire ou linéarithmique** : $u_n \in \mathcal{O}(n \log_2 n)$. C'est la complexité de certains algorithmes de tri et pour les tris par comparaisons, il est impossible de faire mieux.

- **Complexité quadratique** : $u_n \in \mathcal{O}(n^2)$. C'est la complexité d'algorithmes construits avec deux boucles imbriquées comme certains algorithmes de tri.

Soit n la taille d'une donnée. On obtient une complexité quadratique en fonction de n si le nombre d'opérations à effectuer peut s'écrire $\alpha n^2 + \beta n + \gamma$, avec α , β et γ réels, $\alpha > 0$.

- **Complexité polynomiale** : $u_n \in \mathcal{O}(n^k)$, $k \geq 2$ fixé. Un algorithme d'une telle complexité n'est utilisable que dans quelques cas particuliers. La complexité est cubique si $k = 3$.

- **Complexité exponentielle** : $u_n \in \mathcal{O}(2^n)$ ou $u_n \in \mathcal{O}(k^n)$, $k > 1$. C'est la complexité de certains algorithmes de résolution de problèmes pour lesquels personne n'a encore trouvé d'algorithmes plus performants et qui sont en pratique inutilisables.

■ Cas typiques

Avec deux boucles `for` imbriquées, nous avons trois cas typiques. Dans un cas, la complexité est linéaire. Dans les deux autres cas elle est quadratique.

Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

Premier cas : `n` est la taille de la donnée, `k` est un nombre fixé.

```
for i in range(n):
    ...
    for j in range(k):
        ...
```

Il y a `n` passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis `k` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r . Donc pour chaque valeur de `i`, nous avons $q + k \times r = \alpha$ opérations. Le nombre total d'opérations est donc αn et la complexité est linéaire.

Deuxième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(n):
        ...
```

Il y a `n` passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis `n` passages dans la boucle interne. Dans la boucle interne, nous avons un

nombre fixe d'opérations r . Donc pour chaque valeur de i , nous avons $q + n \times r$ opérations et le nombre total d'opérations est alors $n(q + n \times r) = rn^2 + qn$. La complexité est quadratique.

Troisième cas : n est la taille de la donnée.

```
for i in range(n):  
    ...  
    for j in range(i):  
        ...
```

Il y a n passages dans la boucle externe. À chaque passage dans cette boucle, pour chaque valeur de i , il y a un nombre fixe d'opérations q puis i passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r . Donc pour chaque valeur de i , nous avons $q + i \times r$ opérations. Les valeurs de i sont successivement $0, 1, 2, \dots, n-1$. Le nombre total d'opérations est donc $q + (q + 1 \times r) + (q + 2 \times r) + \dots + (q + (n-1) \times r)$, soit $nq + r(1 + 2 + \dots + (n-1))$.

Le calcul de la somme des entiers est connu. Le résultat est $nq + r \frac{n(n-1)}{2}$.

Finalement le nombre d'opérations est $\frac{r}{2}n^2 + \left(q - \frac{r}{2}\right)n$, qui est de la forme $\alpha n^2 + \beta n$ et la complexité est quadratique.

Dans les chapitres précédents, divers niveaux de complexité sont présents. Au chapitre 1, la recherche séquentielle d'un élément dans une liste a une complexité linéaire. Dans le cas d'une liste triée, cette recherche à l'aide d'un algorithme dichotomique au chapitre 4 a une complexité logarithmique. Au chapitre 2, le tri à bulles a une complexité quadratique. C'est aussi le cas avec les algorithmes de tri sélection et insertion au chapitre 8. Dans ce même chapitre, une amélioration est apportée par le tri fusion qui a une complexité log-linéaire.

De manière générale, la dichotomie permet de passer d'un ordre n à un ordre $\log_2 n$.

■ Propriétés

- Des algorithmes permettant de résoudre un même problème, peuvent être rangés suivant leur ordre de complexité du plus efficace au moins efficace. Le classement des ordres de complexité doit être connu : $\mathcal{O}(1)$, $\mathcal{O}(\log_2 n)$, $\mathcal{O}(n)$, $\mathcal{O}(n \log_2 n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(n^2 \log_2 n)$, $\mathcal{O}(n^3)$, etc.
- Si deux blocs d'instructions successifs ont une complexité en $\mathcal{O}(u_n)$ alors la complexité totale est en $\mathcal{O}(u_n)$.
- Si on répète u_n fois un bloc d'instructions de complexité en $\mathcal{O}(v_n)$ alors la complexité totale est en $\mathcal{O}(u_n v_n)$. Si u_n a une valeur constante, la complexité totale est en $\mathcal{O}(v_n)$.
- Si deux blocs d'instructions successifs ont une complexité respectivement en $\mathcal{O}(u_n)$ pour le premier et en $\mathcal{O}(v_n)$ pour le second alors la complexité totale est en $\mathcal{O}(\max(u_n, v_n))$.

■ ■ Vrai/Faux

	Vrai	Faux
1. Pour prouver la terminaison d'un programme, on peut utiliser un invariant de boucle.	☐	☐
2. Si un algorithme qui utilise une boucle possède un invariant de cette boucle alors il est correct.	☐	☐
3. Si un algorithme termine, le résultat produit est celui attendu dans tous les cas ou dans aucun cas.	☐	☐
4. Des tests permettent de trouver toutes les éventuelles erreurs de codage d'une fonction.	☐	☐
5. Une assertion peut signaler un problème mais ne le résout pas.	☐	☐
6. Si un programme produit un résultat faux une seule fois sur un très grand nombre d'essais, c'est probablement une erreur de la machine.	☐	☐
7. Un algorithme qui effectue n recherches dichotomiques sur des listes de longueur n a une complexité en $\mathcal{O}(n \log_2 n)$.	☐	☐
8. Si deux fonctions ont une complexité linéaire, l'enchaînement des deux fonctions a une complexité quadratique.	☐	☐
9. Plus la complexité d'un algorithme est élevée, plus le résultat est précis.	☐	☐
10. Deux algorithmes de même complexité effectuent exactement le même nombre d'opérations.	☐	☐

■ ■ Énoncé des exercices

Exercice 10.1 : Dans la fonction suivante, les valeurs des variables a et b sont des entiers naturels :

```
def f(a, b):
    s, k = a, b
    while k > 0:
        s = s + 1
        k = k - 1
    return s
```

Quelle affirmation est fausse ?

1. La propriété $\ll s+k = a+b \gg$ est un invariant de la boucle `while`.
2. La valeur finale de k est 1.
3. La propriété $\ll k \geq 0 \gg$ est un invariant de la boucle `while`.
4. Le résultat renvoyé est égal à la somme $a+b$.

Exercice 10.2* : La fonction `div_euclid` doit renvoyer le quotient et le reste de la division euclidienne de m par n où m et n sont deux entiers naturels avec n non nul.

Le code de cette fonction comporte une erreur.

```
def div_euclid(m, n):
    """m et n sont deux entiers naturels, n est non nul,
    renvoie le quotient et le reste de la division de m par n
    une propriété invariante est m == n * q + r"""
    assert n != 0
    q, r = 0, m
    assert m == n * q + r
    while r > n:
        r = r - n
        q = q + 1
        assert m == n * q + r
    return q, r
```

1. Prouver la terminaison de l'algorithme.
2. Prouver que la propriété $m = nq + r$ est un invariant de la boucle. Peut-on en déduire que l'algorithme est correct ? Si ce n'est pas le cas, corriger le code.
3. Écrire un jeu de tests pour cette fonction.

Exercice 10.3 : Voici un code en Python :

```
x = 1
while x / 2 > 0:
    x = x / 2
```

Prouver la terminaison et préciser la valeur finale de x .

Exercice 10.4 : On considère la fonction `dichotomie` définie ci-dessous. Tester cette fonction avec des exemples simples afin de déterminer un bug et le corriger.

```
def dichotomie(f, a, b, eps):
    while (b - a) > eps:
        m = (a + b) / 2
        if f(a) * f(m) < 0:
            b = m
        else:
            a = m
    return (a + b) / 2
```

Exercice 10.5 :

1. L'objectif est de tester une fonction qui prend un entier naturel en paramètre et qui doit renvoyer `True` si le paramètre est un nombre premier et `False` sinon.

Pour cela, écrire une fonction `test1` qui utilise une liste de nombres premiers, comme `[2, 3, 5, 7, 11, 13, 17, 19]`, et contrôle pour tous ces nombres si la fonction passée en paramètre renvoie la bonne réponse, c'est-à-dire `True`.

2. Tester, avec la fonction `test1`, la fonction `premier` définie ci-dessous.

```
def premier(n):
    if n == 0 or n == 1:
        return False
    if n == 2:
        return True
    for d in range(2, n):
        if n % d == 0:
            return False
    return True
```

3. Écrire une fonction `test2` qui exécute un contrôle pour tous les nombres non premiers compris entre 0 et 10 inclus. Utiliser cette fonction pour tester la fonction `premier`.

Ce test doit montrer que le code ne produit pas le résultat attendu.

4. Effectuer la modification dans le code de la fonction `premier` pour corriger l'erreur qui a été constatée à la question précédente.

Exercice 10.6 : Voici un algorithme pour déterminer le plus grand élément d'un tableau de nombres non vide représenté par une liste. Prouver sa correction.

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


Exercice 10.7 : La fonction `pgcd` ci-dessous prend en arguments deux entiers naturels a et b non nuls et renvoie la valeur du pgcd de a et b . Prouver sa validité. Pour la correction, considérer l'invariant de boucle constitué par l'ensemble des diviseurs communs de a et b .

```
def pgcd(a, b):
    while b > 0:
        a, b = b, a % b
    return a
```

Exercice 10.8 : Déterminer le nombre d'opérations mathématiques effectuées par le programme suivant en fonction de n avec n entier strictement positif.

Quelle est la complexité de l'algorithme en fonction de n ?

```
c = 0
while n > 0:
    c = c + 1
    n = n // 10
```

Exercice 10.9* : Voici deux programmes pour calculer 2^n avec n entier naturel :

```
def puissance2(n):
    p = 1
    for k in range(1, n+1):
        p = 2 * p
    return p

def puissance2(n):
    p = 1
    b = 2
    while n > 0:
        n, r = n // 2, n % 2
        p = p * b ** r
        b = b * b
    return p
```

1. Dans le premier programme, quel est le nombre exact d'affectations et de multiplications effectuées dans le corps de la boucle en fonction de n ? En déduire le niveau de complexité.
2. Prouver la terminaison et la correction du deuxième programme en considérant l'invariant de boucle : $p \times b^n$.

Déterminer en fonction de n le nombre d'opérations élémentaires effectuées dans le corps de la boucle (compter les opérations mathématiques et les affectations).

Prouver que le niveau de complexité de cet algorithme est en $\mathcal{O}(\log_2 n)$.

Exercice 10.10* : L'objectif est de comparer deux algorithmes permettant d'évaluer la valeur de $P(x)$ pour une valeur de x donnée avec $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$.

La liste des coefficients est notée $a = [a_0, a_1, \dots, a_n]$; attention à l'ordre!

Le programme ci-dessous correspondant au premier algorithme.

```
def polynome(x, a):
    p = a[0]
    for i in range(1, len(a)):
        puis = 1
        for j in range(1, i+1):
            puis = puis * x
        p = p + a[i] * puis
    return p
```

1. Étudier la complexité de ce programme en fonction de n .
2. Le programme correspondant au deuxième algorithme, appelé *algorithme de Hörner*, est basé sur une écriture différente de $P(x)$: $P(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + xa_n) \dots))$. Donc ici, la liste des coefficients est parcourue dans l'ordre inverse; la variable p est initialisée avec a_n et nous n'utilisons qu'une seule boucle dans laquelle est effectuée une multiplication par x et l'addition du coefficient précédent. Écrire le programme et déterminer sa complexité.

Exercice 10.11 : Calcul d'une somme

1. Écrire un programme effectuant simplement avec deux boucles imbriquées le calcul de la somme

$$S = \sum_{i,j=0}^{n-1} (t[i] - t[j])^2 \text{ pour un tableau de nombres } t \text{ de longueur } n.$$

Quelle est la complexité de cet algorithme en fonction de n ?

2. L'écriture de la somme S est transformée à l'aide d'un calcul mathématique :

$$S = 2 \left((n-1)S_n^2 - 2n \sum_{j=1}^{n-1} t[j]S_j \right), \text{ avec } S_k = \sum_{p=0}^{k-1} t[p]$$

Écrire un programme utilisant cette formule. Quelle est la complexité de cet algorithme?

■ ■ Indications

Ex. 10.2

Penser à vérifier les conditions sur le reste r .

Ex. 10.3

Consulter si besoin la représentation des nombres au chapitre 12.

Ex. 10.10

Le principal est de bien gérer l'indice de boucle pour l'algorithme de Hörner.

■ ■ Corrigé du vrai/faux

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

Quelques explications

2. Il peut être seulement partiellement correct. De plus, une propriété peut être invariante sans satisfaire la spécification.
4. Il peut toujours rester une erreur dans certains programmes. Pour certaines fonctions il faudrait pouvoir effectuer une infinité de tests.
8. La complexité reste linéaire.

❑ Erreurs classiques et conseils.—

- Un invariant de boucle qui permet de prouver la correction n'est pas toujours facile à trouver. Une spécification peut donner des informations.
- Il ne faut pas confondre un invariant de boucle qui sert à prouver la correction et un variant de boucle qui sert à prouver la terminaison.
- Pour évaluer une complexité, un calcul exact du nombre d'opérations effectuées n'est pas nécessaire. Un ordre de grandeur suffit.

■ ■ Corrigé des exercices

Exercice 10.1

En sortie de boucle la valeur de k est 0. C'est donc la proposition 2 qui est fausse.

Exercice 10.2

1. La suite formée par les valeurs de r au cours des itérations est une suite d'entiers strictement décroissante : r étant initialisée à m , si $m \geq n$ alors la valeur de r est inférieure à celle de n en un maximum de $m - n$ étapes et la boucle est terminée.
2. Les valeurs initiales q et r sont respectivement à 0 et m . Donc la propriété $m = nq + r = n.0 + m$ est vérifiée avant le premier passage dans la boucle.

Avant une itération arbitraire, supposons que l'on ait $m = nq + r$ et montrons que cette propriété est encore vraie après cette itération. Soient q' la valeur de q et r' la valeur de r à la fin de l'itération. Nous devons montrer que $m = nq' + r'$.

On a $q' = q + 1$ et $r' = r - n$, alors $nq' + r' = n(q + 1) + (r - n) = nq + r = m$. La propriété est bien conservée.

En sortie de boucle, la propriété $m = nq + r$ est vérifiée, mais ce n'est pas suffisant pour une division euclidienne. Le reste r doit vérifier la condition $0 \leq r < n$.

Or, si le programme s'arrête, c'est que la condition $r > n$ n'est plus satisfaite, donc que $r \leq n$ et l'égalité $r = n$ reste possible. On obtient ce cas si m est un multiple de n . On corrige donc le code en remplaçant $r > n$ par $r \geq n$.

Il reste à montrer qu'en fin de boucle $r \geq 0$. Comme r est diminuée de n à chaque itération, à l'itération précédente la valeur de r est $r' = r + n$. Donc si $r < 0$ alors $r' < n$. Et donc la boucle se serait arrêtée à l'itération précédente, ce qui est absurde. On en déduit donc que $r \geq 0$.

En conclusion, le programme se termine avec $0 \leq r < n$ et la propriété $m = nq + r$ est vérifiée à chaque itération ; ceci prouve que l'algorithme effectue bien la division euclidienne de m par n .

3. On envisage pour m les cas $m = 0$, $m = 1$ et $m > 1$. Pour n , les cas sont $n = 1$ ou $n > 1$. Ensuite, pour les couples $(m; n)$, on peut envisager $m < n$, $m = n$, $m > n$, m multiple de n ou m diviseur de n . Ces tests auraient permis de détecter l'erreur dans la fonction `div_euclid`.

```
assert div_euclid2(0, 3) == (0, 0)
assert div_euclid2(1, 1) == (1, 0)
assert div_euclid2(1, 3) == (0, 1)
assert div_euclid2(4, 1) == (4, 0)
assert div_euclid2(3, 8) == (0, 3)
assert div_euclid2(3, 3) == (1, 0)
assert div_euclid2(8, 3) == (2, 2)
assert div_euclid2(6, 2) == (3, 0)
assert div_euclid2(2, 6) == (0, 2)
```

Exercice 10.3

Le nombre de flottants est fini. Les valeurs successives de x après la première division sont de type float et constituent une suite strictement décroissante jusqu'à atteindre la valeur x_0 du plus petit flottant strictement positif. Le quotient de x_0 par 2 vaut 0 et dans ce cas la boucle est terminée. La valeur finale de x est donc x_0 .

Exercice 10.4

On définit deux fonctions et on utilise l'affichage avec `print`.

```
def g(x):
    return x

print(dichotomie(g, -1, 1, 0.001))

def h(x):
    return x**3 - 5*x**2 + x - 5

print(dichotomie(h, 0, 10, 0.001))
```

Dans les deux cas, on obtient pour solution une valeur approchée de `b` à la place de 0 pour la première équation et à la place de 5 pour la deuxième équation.

On peut ajouter une assertion dans la boucle, `assert f(a) * f(b) <= 0`, pour constater qu'une propriété qui doit être invariante ne l'est pas. En effet si `f(a)` vaut 0, alors `a` prend la valeur de `m` non nulle et du même signe que la valeur de `b`. Ce cas se produit aussi si `f(m)` vaut 0 puisque alors, `a` prend la valeur de `m` et au passage suivant dans la boucle `f(a)` a la valeur 0.

Il faut remplacer `if f(a) * f(m) < 0` par `if f(a) * f(m) <= 0`.

Exercice 10.5

1. La fonction `test1` prend en paramètre une fonction `f`.

```
def test1(f):
    liste = [2, 3, 5, 7, 11, 13, 17, 19]
    for n in liste:
        if not f(n):
            return False
    return True
```

2. La fonction `premier` passe ces tests sans problème : `test1(premier)` renvoie `True`.

3. On définit la fonction `test2` en suivant le modèle de la fonction `test1`.

On constate un problème : `test2(premier)` renvoie `False`.

```
def test2(f):
    liste = [0, 1, 4, 6, 8, 9, 10]
    for n in liste:
        if f(n):
            return False
    return True
```

4. À l'examen du code nous pouvons remarquer que la dernière ligne `return True` de la fonction `premier` est mal indentée. Seul le premier passage est exécuté dans la boucle `for`.

Exercice 10.6

La terminaison est évidente avec cette boucle `for`.

La correction se prouve en considérant la propriété pour `i` prenant les valeurs de 1 à `len(tab)-1` :
« `maxi` est le plus grand élément de la liste `tab[0: i+1]` ».

Après le premier passage dans la boucle, avec `i` égal à 1, `maxi` est le plus grand élément de la liste `tab[0:2]` qui contient `tab[0]` et `tab[1]`.

Si la propriété est vraie après un passage, `maxi` est le plus grand élément de `tab[0:i+1]`, et au passage suivant l'élément `tab[i+1]` est comparé à `maxi` qui prend alors la valeur du plus grand des deux. Donc `maxi` est le plus grand élément de la liste `tab[0:i+2]`.

En sortie de boucle, `maxi` est le plus grand élément de `tab[0:len(tab)]`, donc de `tab`.

Exercice 10.7

Pour la terminaison, nous considérons la suite des valeurs de `b` : à chaque passage dans la boucle, la nouvelle valeur de `b` est le reste `r` de la division euclidienne de `a` par `b` et par définition $0 \leq r < b$. La suite des valeurs de `b` constitue donc une suite d'entiers naturels strictement décroissante et minorée par 0 ; elle converge en un nombre fini d'itérations vers 0.

Pour la correction, soit $D = D(a, b)$ l'ensemble des diviseurs communs de `a` et `b`. C'est l'invariant de boucle. Soit `r` le reste de la division euclidienne de `a` par `b` : $a = bq + r$.

Si `d` est un diviseur de `a` et `b` alors $a = kd$ et $b = k'd$, d'où $kd = k'dq + r$, soit $r = (k - k'q)d$. Donc `d` est un diviseur de `r`. Réciproquement si `d` est un diviseur de `b` et `r` alors `d` est un diviseur de `a`. Nous avons donc prouvé que $D(a, b) = D(b, r)$.

En fin de boucle, `b` = 0, $D = D(a, 0)$, et le pgcd est la valeur finale de `a`.

Exercice 10.8

À chaque passage dans la boucle, nous avons deux affectations avec une addition et une division entière et le nombre `n` perd un chiffre dans son écriture décimale. Le nombre de passages dans la boucle est donc égal au nombre de chiffres dans l'écriture décimale de `n`, soit $\text{Ent}(\log_{10}(n) + 1)$ où Ent désigne la partie entière. La complexité est en $\mathcal{O}(\log_{10} n)$, soit logarithmique.

Exercice 10.9

1. Il y a `n` passages dans la boucle (`k` prend toutes les valeurs de 1 à `n`), et à chaque passage nous comptons une multiplication et une affectation (`p = 2 * p`) ; le total est donc de $2n$ opérations et le niveau de complexité est en $\mathcal{O}(n)$.

2. Pour prouver la terminaison nous considérons la suite des valeurs de `n`. C'est une suite d'entiers positifs strictement décroissante qui atteint 0 après un nombre fini d'itérations (le nombre de chiffres dans l'écriture binaire de `n` puisque chaque division par 2 retire un chiffre à ce nombre).

Montrons que l'expression $p \times b^n$ est un invariant de la boucle. Considérons l'expression $p \times b^n$ avant un passage dans la boucle avec $n = 2q + r$. Lors du passage dans la boucle, les nouvelles valeurs de `p`, `b` et `n` sont $p \times b^r$, b^2 , et `q`. Or $p \times b^r \times (b^2)^q = p \times b^{2q+r} = p \times b^n$.

Avant l'entrée dans la boucle, $pb^n = 2^n$. Donc en sortie de boucle, puisque la valeur de la variable `n` est 0, on obtient pour valeur de `p` : $p = 2^n$.

À chaque passage dans la boucle, quatre affectations et cinq opérations sont effectuées. Ce nombre est constant et il y a `k` passages dans la boucle avec `k` égal à la partie entière de $(1 + \log_2 n)$. La complexité est donc logarithmique.

Exercice 10.10

1. La boucle interne est utilisée pour calculer les puissances de `x` et la boucle externe pour multiplier le résultat obtenu par le coefficient correspondant et l'ajouter à la valeur courante de $P(x)$ affectée à la variable `p`.

La variable `i` prend `n` valeurs. Pour chaque valeur de `i`, nous comptons une affectation avant la boucle interne puis une multiplication, une addition et une affectation après cette boucle. Pour

chaque valeur de i , la variable j prend i valeurs et il y a donc i passages dans cette boucle avec à chaque passage un nombre d'opérations constant. Le nombre total de passages dans la boucle interne est $1 + 2 + \dots + n$, soit de l'ordre de n^2 . Le niveau de complexité est donc en $\mathcal{O}(n^2)$.

2. Un programme correspondant à l'algorithme de Hörner est le suivant :

```
def horner(x, a):
    n = len(a)
    p = a[n-1]
    for i in range(1, n):
        p = p * x + a[n-1-i]
    return p
```

La variable i prend $n - 1$ valeurs. À chaque passage dans la boucle le nombre d'opérations effectuées est constant. Le niveau de complexité est donc en $\mathcal{O}(n)$.

L'algorithme de Hörner apparaît clairement comme le plus efficace.

Exercice 10.11

1. On utilise simplement la formule pour écrire un programme :

```
def somme(t):
    n = len(t)
    s = 0
    for i in range(n):
        for j in range(n):
            s = s + (t[i]-t[j])**2
    return s
```

Nous avons deux boucles imbriquées. La boucle externe est parcourue n fois. À chaque passage, la boucle interne est parcourue n fois et à chaque passage le nombre d'opérations effectuées est constant. La complexité est en $\mathcal{O}(n^2)$.

2. Un programme :

```
def somme(t):
    n = len(t)
    s = 0
    sn = 0
    for j in range(n-1):
        sn = sn + t[j] # calcul de la somme partielle
        s = s + t[j+1] * sn
    sn = sn + t[n-1]
    s = 2 * ((n-1) * sn * sn - 2 * n * s)
    return s
```

La boucle est parcourue $(n - 1)$ fois avec à chaque passage un nombre constant d'opérations. La complexité est en $\mathcal{O}(n)$.

■ Représentation des nombres

Chapitre **11**

Les entiers

UN SCIENTIFIQUE



Fils d'un commerçant toscan, de son vrai nom Léonard de Pise, **Fibonacci** (env.1180-env.1250) suit son père en Afrique du nord pour l'aider à faire ses comptes. Là, il découvre les chiffres arabes et la numération de position pour noter les entiers. Il en comprend aussitôt les immenses avantages. Soucieux de la faire connaître, il rédige un ouvrage, *Liber abaci*, dans lequel il en énonce les principes. Il faudra encore près de trois siècles pour qu'elle s'impose en Occident.

■ Un peu d'histoire

Au VI^e siècle, dans l'empire Gupta en Inde, est apparue la notation de position que nous utilisons toujours, même si la forme des chiffres a été modifiée. Un chiffre y possède un double sens, sa valeur nominale d'une part mais aussi celle qu'indique sa position déterminant s'il s'agit d'unité, de dizaine, de centaine, etc. Pour différencier, par exemple 27 et 207 l'utilisation du zéro était nécessaire. C'est ainsi qu'est apparu un nouveau concept que les Indiens nomment *sounya*, d'un mot signifiant *vide*.

Au début du IX^e siècle, le calife de Bagdad fit venir de nombreux savants à la *Maison de la sagesse* pour traduire les textes scientifiques grecs ou indiens. C'est ainsi que Mohammed al-Khwarizmi rédigea un livre présentant ce nouveau système de numération. Traduit en latin sous le nom de *Algoritmi de numero indorum*, cet ouvrage fit connaître cette notation en Occident.

OBJECTIFS

La numérisation d'une donnée consiste à la remplacer par un nombre. Comment sont représentés les nombres entiers dans une machine ?

- Connaître différentes représentations des entiers naturels :
 - ▶ comprendre l'écriture d'un entier naturel dans une base quelconque ;
 - ▶ maîtriser la représentation binaire d'un entier positif ;
 - ▶ savoir évaluer les nombres représentables avec des mots de taille fixe ;
 - ▶ distinguer l'implémentation dans un langage et la représentation en machine.
- Faire la distinction entre les entiers signés et les entiers non signés :
 - ▶ envisager différentes représentations des entiers signés ;
 - ▶ comprendre les raisons d'utiliser des mots de taille fixe ;
 - ▶ savoir déterminer la représentation en complément à deux.
- Comprendre la gestion des entiers multi-précision en Python :
 - ▶ découvrir la représentation des entiers en Python ;
 - ▶ être capable d'en déduire les avantages et les inconvénients ;
 - ▶ comprendre les conséquences sur les temps de calculs.

■ Numérisation

Toutes les données utilisées dans une machine ont été numérisées, donc transformées en nombres entiers. Le nombre 91 peut représenter l'entier 91. Mais il peut aussi représenter, (en suivant le code ASCII), la lettre "a" ou une quantité de rouge dans une image en couleur. Numériser une information, c'est la convertir en une suite de 0 et de 1. Cette suite représente de manière exacte ou approchée une donnée.

Ces nombres doivent être conservés en mémoire et manipulés. Il est important de comprendre comment ces entiers sont représentés en machine et de distinguer cette représentation de celle utilisée par un langage de programmation particulier.

Pour écrire des entiers, nous sommes habitués à utiliser une notation de position avec le système décimal. Le zéro exprime le vide ou l'absence et permet de distinguer par exemple 35 de 305.

■ Représentation des nombres

Nous utilisons en permanence la notation positionnelle en base dix. Lorsque nous écrivons 357, que nous lisons « trois-cent-cinquante-sept », c'est une simplification de l'écriture $3 \times 100 + 5 \times 10 + 7$.

Cette notation est probablement née en Inde où elle était utilisée entre les années 600 et 700 dans des opérations arithmétiques. Elle est arrivée en Perse, lorsque le mathématicien al-Khwarizmi a traduit les chiffres en langue Arabe. Les traductions des ouvrages d'al-Khwarizmi en latin permettront de répandre cette notation en Europe à partir des années 1200.

Un entier naturel peut donc s'écrire $c_n c_{n-1} \dots c_1 c_0 = c_n 10^n + c_{n-1} 10^{n-1} + \dots + c_1 10 + c_0$.

Avec la base dix, nous utilisons dix chiffres de 0 à 9. Il apparaît clairement que le rôle du zéro dans l'écriture positionnelle est capital. Sinon, comment distinguer 270 de 27, ou 7803 de 783 ?

En suivant le même principe, nous pouvons envisager une notation positionnelle en base b , où b est un entier quelconque avec $b \geq 2$, en utilisant b chiffres, de 0 à $b - 1$.

$$(a_n a_{n-1} \dots a_1 a_0)_b = a_n b^n + a_{n-1} b^{n-1} + \dots + a_1 b + a_0$$

$$\text{Par exemple : } (2301)_4 = 2 \times 4^3 + 3 \times 4^2 + 0 \times 4 + 1 = 180.$$

Une difficulté apparaît si $b > 10$, puisque nous ne disposons que de 10 chiffres dans notre écriture. En base seize par exemple, nous avons besoin de seize chiffres. Il a été choisi de compléter avec des lettres. Ainsi, les chiffres de zéro à quinze sont habituellement notés 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Si $b = 2$, on obtient le système binaire. Les deux chiffres 0 et 1 sont appelés bits, mot qui vient de l'anglais « binary digit ».

C'est le mathématicien Leibnitz qui est considéré à l'origine du développement du système binaire pour lequel il a précisé comment calculer en particulier pour les quatre opérations de base.

Certains des premiers ordinateurs calculaient en utilisant l'arithmétique décimale. C'est à partir des travaux de von Neumann dans les années 1940 que le calcul en binaire est devenu la norme.

■ Le binaire

L'utilisation du binaire a plusieurs raisons. Une première raison est qu'avec une machine utilisant des circuits électroniques, il est plus simple et plus sûr de ne considérer que deux états dans l'observation de ces circuits et donc de construire une machine qui n'utilise que deux symboles pour son fonctionnement. À ces deux états correspondent les deux chiffres 0 et 1. Une deuxième raison

est que ces deux états ou ces deux chiffres peuvent correspondre aux deux valeurs d'une propriété, faux ou vrai, ce qui permet de faire le lien entre l'arithmétique binaire et le calcul logique. Ainsi les variables qui ne peuvent prendre que deux valeurs 0 ou 1 sont appelées des *variables booléennes* du nom du mathématicien britannique George Boole qui a développé au XIX^e une algèbre liée à la théorie des ensembles. Les trois opérations logiques addition, multiplication et négation correspondant à la réunion, l'intersection et le complémentaire sur l'ensemble des parties d'un ensemble de référence.

Les valeurs 0 et 1 correspondent aussi aux valeurs de la fonction caractéristique d'une partie A d'un ensemble E : $K_A(x) = 1$ si $x \in A$ et 0 sinon. Ainsi le produit de deux bits correspond à la conjonction de deux valeurs booléennes et à l'intersection de deux parties.

Voici quelques remarques complémentaires qui militent pour le binaire.

En base deux, une multiplication est guère plus compliquée qu'une addition.

Le passage d'une base $b = 2$ à une base $b = 2^p$ est toujours simple. Cela consiste à regrouper les bits par p bits. Par exemple, $(a_n a_{n-1} \dots a_1 a_0)_2 = (A_q \dots A_1 A_0)_{2^2}$ avec $A_k = a_{2k+1} a_{2k}$.

Dans une addition en base b , les retenues éventuelles valent toujours un. Mais dans une multiplication en base b , il y a $b - 1$ retenues possibles. En base deux, la seule retenue possible vaut 1 comme pour l'addition.

Donc, que ce soit pour le fonctionnement de la machine ou pour la simplicité des calculs, le binaire semble être un bon choix. Il permet de réaliser un lien entre la physique, le calcul numérique, la logique booléenne et la théorie des ensembles.

■ Vocabulaire

Une suite composée de 0 et de 1 s'appelle un *mot*. Les chiffres 0 et 1 sont les *lettres* et l'ensemble $\{0, 1\}$ est l'*alphabet*. La taille d'un mot s'exprime en bits. Un mot de huit bits est un *octet* (ou *byte*). Un mot est l'*unité de base* pour la machine ou le processeur. On compte en bits ou en octets. Avec seize bits, soit deux octets, on parle de *word*, avec trente-deux bits, soit quatre octets, de *dword* (pour double word) et avec soixante-quatre bits, soit huit octets, de *qword* (pour quadruple word). Suivant les architectures des ordinateurs, la taille du *mot* ou du *word* change. Dans les machines récentes un word a une taille de 64 bits. Dans les architectures plus anciennes un word a une taille de 32 bits.

Le type `char`, qui permet de représenter un caractère dans certains langages, utilise huit bits depuis le début des années 1970 et les premiers processeurs Intel. Ce type permet de représenter 256 caractères ($256 = 2^8$), en particulier ce que l'on trouve sur les touches d'un clavier usuel.

Note : en Python `ord("a")` vaut 91 et `chr(91)` vaut "a".

■ Entiers naturels

■ Mots de taille fixe

Nous avons l'habitude d'écrire des nombres comme 8, 17, 235, 7813. Si nous décidons d'utiliser exactement quatre chiffres pour représenter un nombre, nous les écrivons 0008, 0017, 0235, 7813. Les entiers naturels représentables avec quatre chiffres sont alors tous les nombres entiers allant de 0 (0000) à 9999, soit $10^4 = 10000$ nombres.

La mémoire d'une machine ressemble à une feuille de papier quadrillée sur laquelle nous pouvons écrire un chiffre 0 ou 1 dans chaque petit carré. Chaque carré est numéroté, c'est son adresse. Les machines numériques n'utilisent que le binaire pour stocker des données sous forme de suites composées uniquement de 0 et de 1. Nous pouvons écrire en binaire des nombres comme 1, 11, 101,

1011, soient 1, 3, 5, 11 en décimal. Pour stocker ces nombres en machines il faudrait inscrire les chiffres de chaque nombre et pour chaque nombre, son nombre de chiffres. Sinon il est impossible de savoir ce que représente 10111 écrit sur une ligne de la feuille.

1	0	1	1	1
---	---	---	---	---

Ce sont les nombres 1011 et 1, les nombres 101 et 11 ? De plus si un nombre occupe deux cases et que nous n'avons plus besoin de ce nombre, nous pouvons l'effacer mais nous ne disposons que de deux cases contigües pour écrire un nouveau nombre. Il est donc plus efficace de décider que tous les nombres sont stockés avec exactement le même nombre de chiffres. Par exemple si nous décidons d'utiliser des mots de taille quatre bits, alors 1, 3, 5, 11 s'écrivent respectivement 0001, 0011, 0101, 1011. Et nous avons ci-dessous les nombres 1011 et 0001 sans ambiguïté possible.

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

Avec quatre bits, nous pouvons écrire seulement $2^4 = 16$ nombres. Avec un octet, nous pouvons représenter $2^8 = 256$ nombres. C'est suffisant pour coder les différents caractères figurant sur un clavier d'ordinateur. Les calculs étant effectués par un processeur, la taille des mots est liée à la capacité du processeur. Actuellement les calculs avec des nombres sur 64 bits sont courants.

Le langage Python gère des entiers plus longs, et leur taille n'est limitée que par la capacité de la mémoire.

■ Méthodes

Sous forme de tableau, nous obtenons pour le nombre 8050 :

...	$10000 = 10^4$	$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$
...		8	0	5	0
...	0	8	0	5	0

L'utilisation du chiffre 0 est déterminante.

L'écriture de 5326 en base dix est constituée des restes obtenus dans les divisions euclidiennes successives par 10 jusqu'à obtenir un quotient nul.

$$\begin{array}{r}
 5326 \div 10 = 532 \text{ rest } 6 \\
 532 \div 10 = 53 \text{ rest } 2 \\
 53 \div 10 = 5 \text{ rest } 3 \\
 5 \div 10 = 0 \text{ rest } 5
 \end{array}$$

En base deux le principe est le même qu'en base dix.

Par exemple, l'écriture de 11 en base deux, ou l'écriture binaire de 11, est constituée des restes obtenus dans les divisions euclidiennes successives par 2 jusqu'à obtenir un quotient nul.

$$\begin{array}{r}
 11 \div 2 = 5 \text{ rest } 1 \\
 5 \div 2 = 2 \text{ rest } 1 \\
 2 \div 2 = 1 \text{ rest } 0 \\
 1 \div 2 = 0 \text{ rest } 1
 \end{array}$$

On obtient l'écriture en base deux en lisant les restes de bas en haut, soit 1011_2 , soit $r_4r_3r_2r_1$.

Écrivons les calculs sous une autre forme :

$$11 = 5 \times 2 + 1 = (2 \times 2 + 1) \times 2 + 1 = ((1 \times 2) \times 2 + 1) \times 2 + 1 = 1 \times 2^3 + 1 \times 2 + 1$$

$$\text{d'où, } 11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

Nous pouvons utiliser un tableau comme en base dix :

...	$16 = 2^4$	$8 = 2^3$	$4 = 2^2$	$2 = 2^1$	$1 = 2^0$
...		1	0	1	1

Note :

- le bit (ou chiffre) le plus à gauche est appelé bit de poids fort ;
- le bit (ou chiffre) le plus à droite est appelé bit de poids faible.

En informatique, tout est écrit en base deux. Les données écrites sont souvent regroupées en base huit ou seize. La présentation en base seize améliore grandement la lisibilité.

Base deux	$\longleftrightarrow$	Base seize
1101	$\longleftrightarrow$	D
0011	$\longleftrightarrow$	3
1101 0011	$\longleftrightarrow$	D3

Justification de la méthode

Le nombre seize s'écrit 10000 en base deux et 10 en base seize : $10000_2 = 10_{16}$;

$$\text{donc } (a_7a_6a_5a_4a_3a_2a_1a_0)_2 = (a_7a_6a_5a_4)_2 \times 10000_2 + (a_3a_2a_1a_0)_2 = (A_1)_{16} \times 10_{16} + (A_0)_{16}$$

$$\text{d'où : } (a_7a_6a_5a_4a_3a_2a_1a_0)_2 = (A_1A_0)_{16}.$$

Remarque : pour multiplier par deux un nombre écrit en base deux, il suffit d'ajouter un zéro à droite du nombre : $1011_2 \times 10_2 = 10110_2$. Cette propriété est vraie dans une base b quelconque : le produit d'un nombre par b s'obtient en ajoutant un zéro à droite du nombre.

■ Représentation en machine

Un entier naturel s'écrit sans signe et on parle d'*entier non signé*.

Dans une machine, si on utilise un octet pour représenter un entier naturel, on peut représenter les entiers naturels de 0 (0000 0000 en base deux) à 255 (1111 1111 en base deux). Le nombre 45 par exemple est représenté par 0010 1101.

Si on utilise seize bits, soit deux octets, on peut représenter les entiers naturels jusqu'à 65535 (1111 1111 1111 1111 en base deux) et dans ce cas 45 est représenté par 0000 0000 0010 1101.

Avec n bits, on peut représenter les nombres entre 0 et $2^n - 1$, c'est-à-dire tout nombre k qui s'écrit :

$$k = \sum_{i=0}^{n-1} b_i 2^i \quad \text{avec } b_i \in \{0, 1\}$$

En général, un entier est codé sur 4 octets, (soit 32 bits), ou sur 8 octets, (soit 64 bits).

Addition de nombres binaires

L'addition est l'une des opérations les plus simples pour un ordinateur. Il suffit de savoir exécuter les opérations $0 + 0$, $1 + 0$, $0 + 1$ et $1 + 1$ avec une éventuelle retenue.

Pour additionner deux nombres en binaire on peut poser l'addition comme on le fait en base dix, avec le système de retenue.

On utilise les résultats : $0 + 0 = 0$, $1 + 0 = 0 + 1 = 1$ et $1 + 1 = 10$.

Par exemple :

$$\begin{array}{rcccccc}
 & 1 & 1 & 0 & 1 & & \text{treize} \\
 + & 1 & 0 & 0 & 1 & & \text{neuf} \\
 \hline
 1 & & & & 1 & & \text{retenues} \\
 \hline
 1 & 0 & 1 & 1 & 0 & & \text{vingt-deux}
 \end{array}$$

Attention, si la taille des entiers est limitée, par exemple avec quatre bits, alors dans l'addition ci-dessus le bit à gauche est perdu. Donc pour la machine, la somme de 1101 et de 1001 vaut 0110. Autrement dit, si nous demandons à la machine de calculer $13 + 9$, elle nous répond 6 !

■ Entiers signés

Considérons un ordinateur qui utilise l'arithmétique décimale et des nombres écrits avec quatre chiffres. Dix-milles nombres sont donc représentables pas des mots de 0000 à 9999. Si nous souhaitons représenter des nombres négatifs, nous pouvons envisager plusieurs possibilités.

On pourrait représenter les nombres de $+0$ à $+9999$ et les nombres de -0 à -9999 . On utilise un symbole de plus et il y a deux zéros. Mais si nous n'autorisons que quatre chiffres, un chiffre est monopolisé pour le signe et il n'en reste que trois pour la valeur absolue du nombre.

Une autre possibilité serait de traduire tous les nombres n de -5000 à 4999 par $n + 5000$. Une difficulté majeure avec ce choix est alors de programmer des opérations.

Si nous effectuons $9999 + 0001$ en éliminant le bit de retenue, nous obtenons 0000. Autrement dit : $0000 - 0001 = 9999$. De même $0000 - 0002 = 9998$, etc. Nous pourrions donc dire que les nombres positifs sont représentés par les nombres de 0000 à 4999 et les négatifs par les nombres de 5000 à 9999. Tous les nombres commençant par 5, 6, 7, 8, ou 9 sont négatifs et représentent les nombres de -5000 à -1 . Les négatifs n sont représentés par $n + 10^4$.

Ce type de représentation est celui utilisé avec les angles sur le cercle trigonométrique gradué de degré en degré. Les angles de -180 à 179 correspondent à des angles de 0 à 359 : les angles α positifs à α , les angles α négatifs à $\alpha + 360$. Ainsi l'angle -1 correspond à 359 .

Si nous parcourons le cercle dans le sens direct à partir de 0 , chaque graduation correspond à un nombre positif, dans l'ordre $0, 1, 2, \dots, 178, 179, 180, 181, \dots, 358, 359$ (ceci correspond aux entiers naturels). Chacune de ces graduations peut aussi correspondre à un entier relatif, dans l'ordre $0, 1, 2, \dots, 178, 179, -180, -179, \dots, -3, -2, -1$. Ce principe s'applique aussi dans la lecture de l'heure. Nous pouvons dire "dix heures quarante" ou "onze heures moins vingt". Le négatif -20 correspond à 40 et -1 correspond à 59 .

Remarque : on remplace chaque chiffre c par le chiffre $9 - c$ dans un nombre à quatre chiffres, puis on lui ajoute 1 ; par exemple $n = 3415$ donne 6584 puis $p = 6585$; alors $n + p = 10000$, soit 0 en ne gardant que quatre chiffres. Donc p représente $-n$. Cette représentation s'appelle *la représentation en complément à 10^4* .

■ Représentation en complément à deux des entiers signés.

De manière générale, avec n bits, la représentation en machine d'un entier négatif r , est l'écriture binaire de la différence $2^n - (-r)$. Cette représentation s'appelle le complément à 2^n , souvent abrégé en complément à deux. Pratiquement, pour trouver la représentation d'un entier négatif r , on prend

l'écriture binaire de $-r$ (qui est un entier positif), on inverse les bits de cette écriture et on ajoute 1. Inverser les bits et ajouter 1 sont des opérations simples pour la machine. Dans toutes ces écritures, la taille des mots, le nombre de bits utilisés, est fixée.

Exemple avec une taille de six bits, pour obtenir le codage de -12 :

001100 (codage de 12 en binaire sur 6 bits)

110011 (complément à un, on inverse chaque bit)

110100 (on ajoute 1) : représentation de -12 en complément à deux sur 6 bits.

Examinons la capacité de ce système. Avec n bits, on peut représenter les nombres compris entre -2^{n-1} et $2^{n-1} - 1$. Les nombres négatifs ont tous le bit de poids fort égal à 1.

Les écritures binaires des entiers naturels de 0 à $2^{n-1} - 1$ représentent les entiers relatifs positifs ou nul correspondants. Les écritures binaires des entiers naturels de 2^{n-1} à $2^n - 1$ représentent les entiers relatifs négatifs de -2^{n-1} à -1 .

Ainsi avec un octet, soit huit bits, on peut écrire les entiers naturels n de 0 à 255 et donc représenter les entiers relatifs de $-2^7 = -128$ à $2^7 - 1 = 127$. Les nombres n de 0 à 127, (de 0000 0000 à 0111 1111 en base deux), servent à représenter les entiers relatifs positifs ou nul r avec $n = r$ et les nombres n de 128 à 255, (de 1000 0000 à 1111 1111 en base deux), représentent les entiers relatifs négatifs r avec $n = r + 256$.

Avec quatre octets, on peut écrire les entiers naturels n de 0 à $2^{32} - 1$ et donc représenter les entiers relatifs de -2^{31} à $2^{31} - 1$. Les nombres n de 0 à $2^{31} - 1$ servent à représenter les entiers relatifs positifs ou nul r avec $n = r$ et les nombres n de 2^{31} à $2^{32} - 1$ représentent les entiers relatifs négatifs r avec $n = r + 2^{32}$.

Remarque : quel que soit le nombre n de bits utilisés, le nombre -1 est représenté par le nombre $2^n - 1$ écrit en binaire, donc par une suite de 1.

Résumé de la représentation avec trois bits :

codage binaire	000	001	010	011	100	101	110	111
entiers non signés	0	1	2	3	4	5	6	7
entiers signés	0	1	2	3	-4	-3	-2	-1

■ Entiers multi-précision

■ Programmation

La plupart des processeurs calculent avec des nombres binaires de taille limitée à 32 ou 64 bits. Avec certains langages, le nombre d'octets avec lequel on travaille sur les entiers peut être précisé (un octet, deux octets, quatre octets, huit octets). Les calculs sont en général gérés directement par le processeur. Il y a certains risques qu'il faut connaître.

Pour simplifier, supposons que les entiers sont codés sur deux octets. Si le langage nous permet de travailler uniquement sur des entiers naturels, des entiers non signés, nous avons à notre disposition les nombres de 0 à 65535. Si notre programme est amené à demander le calcul $65500 + 40$, le résultat de l'addition sera 4. (En binaire, le résultat est 1 000... 0100 et le 1 à gauche disparaît). De même le résultat du produit 32×2048 sera 0. Avec des entiers signés, toujours sur deux octets, nous disposons des nombres de -32768 à 32767 . Donc, par exemple, le calcul $32000 + 1000$ aura pour résultat -32536 .

Pour traiter ce genre de problème, par exemple le dépassement de capacité dans une addition, on peut faire une remarque simple sur les entiers signés. Si les deux opérandes sont du même signe,

il y a dépassement si le résultat n'est pas du même signe. Par exemple, avec quatre bits, $6 + 2$ donne un nombre négatif : $0110 + 0010 = 1000$. Il faut alors gérer ce type de problème.

Si les deux opérandes ne sont pas du même signe, il n'y a jamais de dépassement.

Par exemple $6 - 2$ donne bien 4 (la soustraction $6 - 2$ est remplacée par l'addition $6 + (-2)$), soit : $0110 - 0010 = 0110 + 1110 = 0100$.

Par contre, avec deux entiers signés p et q sur n bits, si $p + q \geq 2^{n-1}$ ou si $p + q < -2^{n-1}$, il y a un dépassement de capacité positif dans le premier cas, négatif dans le second cas.

■ Langage Python

Lorsque nous programmons en langage Python, nous n'avons pas, à priori, à nous soucier de la représentation des entiers.

Remarque : avec la représentation des entiers signés en complément à deux, il est facile d'augmenter le nombre de bits utilisés pour la représentation. Si l'entier est positif, on ajoute devant le nombre de 0 nécessaires, si l'entier est négatif, on ajoute devant le nombre de 1 nécessaires.

Le langage Python gère la taille, illimitée dit-on, mais limitée quand même par la capacité de la mémoire. Cependant, si un nombre dépasse la taille maximale utilisable par le processeur, ce nombre est découpé en deux ou plusieurs parties et Python s'occupe des différentes opérations à effectuer. Cela prend alors plus de temps et d'espace en mémoire et peut ralentir sérieusement un programme.

On peut vérifier avec le programme suivant que la première boucle s'exécute environ mille fois plus vite que la deuxième. Le temps affiché est environ 1/1000 de seconde pour la première boucle et plus d'une seconde pour la seconde. La différence s'explique par la gestion des entiers longs dans la deuxième boucle.

```
from time import time

top = time()
n = 0
for i in range(1800):
    n = n * 2 ** i
print(time() - top)

top = time()
n = 2**180
for i in range(1800):
    n = n * 2 ** i
print(time() - top)
```

■ Compléments

En Python, une instruction comme $x = 537$ définit une variable x dont la valeur est 537. Plus précisément, le nom x est lié à l'objet entier dont la valeur est 537 (voir l'annexe en fin d'ouvrage). En mémoire, diverses informations sont enregistrées en plus de la valeur. Sur une machine en 64 bits, ces informations sont stockées sur des suites, ou blocs, de huit octets. La valeur peut être très grande et être stockée sur plusieurs blocs. Le nombre de blocs utilisés est donc lui aussi enregistré en mémoire.

Dans la représentation ci-dessous, chaque case représente un octet écrit en hexadécimal.

01	00	00	00	00	00	00	00	19	02	00	00
----	----	----	----	----	----	----	----	----	----	----	----

L'ordre d'enregistrement des octets dépend de la machine : on parle d'ordre *little endian* ou *big endian*. Dans l'exemple, le bloc de huit octets représente le nombre 1, pour un bloc utilisé. Le nombre 19 en hexadécimal représente le nombre $1 \times 16 + 9$ soit 25. Le bloc de quatre octets représente donc le nombre $25 + 2 \times 256^1 + 0 \times 256^2 + 0 \times 256^3 = 537$.

Si Python autorise la manipulation d'entiers sans taille limite autre que celle de la mémoire de la machine, la machine travaille elle avec des entiers codés sur des mots de taille fixe. Cette taille est limitée en particulier par le processeur (32 ou 64 bits) et les bus qui transmettent les données entre les mémoires et le processeur.

Nous pouvons obtenir facilement certaines informations avec le module `sys`.

```
>>> import sys
>>> sys.maxsize
9223372036854775807
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

Le nombre `maxsize` est un entier donnant la valeur maximale de la taille des entiers gérés par la machine, soit $2^{31} - 1$ sur une plateforme 32 bits et $2^{63} - 1$ sur une plateforme 64 bits.

Le nombre `bits_per_digit` est le nombre de bits utilisés pour chaque mot en Python. Les entiers Python sont stockés en interne en base 2^{30} .

Le nombre `sizeof_digit` est la taille en octets d'un mot Python, la taille d'un bloc.

Le nombre maximal de blocs de 4 octets est donc de l'ordre de $2^{63}/4$, ce qui nous donne une idée du plus grand entier utilisable par Python.

Pour un entier positif, on trouve donc en mémoire le nombre de blocs de 4 octets utilisés k . L'entier est ensuite codé suivant sa taille (le nombre de blocs de 4 octets) sur 4 octets, ou 8 octets, ou 12 octets, etc. Chaque bloc de 4 octets est un nombre compris entre 0 et $2^{30} - 1$ bornes comprises. Le dernier bloc ne peut être 0.

La valeur du nombre est donc $\text{bloc}[0] + \text{bloc}[1] \times 2^{30} + \text{bloc}[2] \times 2^{60} + \dots$

Par exemple avec $n = 537 + 5 \times 2^{30} + 7 \times 2^{60}$, on obtient trois blocs :

03	00	00	00	00	00	00	00	19	02	00	00	05	00	00	00	07	00	00	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Les huit premiers octets donnent le nombre de blocs. Ensuite quatre octets pour le nombre 537, quatre octets pour le nombre 5 et quatre octets pour le nombre 7.

■ Calculs

On pourrait utiliser des nombres de type float pour représenter des grands nombres mais même les calculs les plus simples sont alors des calculs approchés (voir chapitre 12).

Alors que les calculs effectués avec des entiers sont des calculs exacts. Cependant, pour des entiers très grands, stockés en Python sur plusieurs blocs, il faut gérer les opérations qui sont données à exécuter au processeur, en particulier utiliser des algorithmes de calcul performants.

Prenons par exemple le produit de deux entiers positifs $p = a + b \times 2^{30}$ et $q = c + d \times 2^{30}$. Une simple application de la distributivité donne pour résultat $r = ac + (ad + bc) \times 2^{30} + bd \times 2^{60}$.

Il y a donc quatre produits à calculer avec des nombres compris entre 0 et 2^{30} . La valeur de chaque produit est comprise entre 0 et 2^{60} et le résultat final est compris entre 0 et 2^{120} .

Un algorithme comme celui de Karatsuba, développé dans les années 1960, utilise une stratégie plus performante.

■ Algorithme de Karatsuba

On utilise la base dix pour simplifier la présentation et on considère $p = a10^m + b$ et $q = c10^m + d$.

Par exemple, avec $m = 2$, si $n = 6738$ alors $n = 67 \times 10^2 + 38$.

La stratégie utilisée lorsqu'on pose la multiplication des nombres p et q est une application de la distributivité. On obtient $pq = ac10^{2m} + (ad + bc)10^m + bd$

On est ramené aux quatre produits ac , ad , bc et bd et en terme de complexité on peut montrer que l'on ne gagne rien.

On écrit alors $pq = ac10^{2m} + (ac + bd - (a - b)(c - d))10^m + bd$. Sous cette forme il n'y a que trois produits à calculer : ac , bd et $(a - b)(c - d)$. On montre alors que la gain en terme de complexité est effectif.

Pour l'algorithme, nous séparons les produits à calculer en transformant l'expression :

$$pq = (10^{2m} + 10^m)ac + 10^m(b - a)(c - d) + (10^m + 1)bd$$

Cette formule peut être utilisée dans un processus récursif.

```
def karatsuba(p, q, m=9):
    if p < q:
        p, q = q, p
    if q == 0:
        return 0
    if q <= 10**m:
        return p * q
    a, b = p // 10**m, p % 10**m
    c, d = q // 10**m, q % 10**m
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    b_a_c_d = karatsuba(b-a, c-d)
    r = (10**(2*m) + 10**m) * ac + 10**m * b_a_c_d + (10**m + 1) * bd
    return r
```

La valeur par défaut du paramètre m est 9. Cela correspond à une taille de mot 10^9 , soit un peu moins que 2^{30} .

L'algorithme est de type *diviser pour régner*.

■ ■ Vrai/Faux

	Vrai	Faux
1. Si l'écriture binaire d'un entier naturel se termine par n zéros, alors cet entier est divisible par 2^n .	☐	☐
2. En base huit, on utilise les chiffres de 1 à 8.	☐	☐
3. Avec n bits, les entiers représentables à l'aide de leur écriture binaire sont inférieurs à 2^n .	☐	☐
4. En binaire sur n bits, pour diviser un entier naturel par 2, on supprime le bit à la fin et on ajoute un 1 devant.	☐	☐
5. L'entier 170 s'écrit AA en hexadécimal.	☐	☐
6. Si les entiers non signés sont codés sur des mots de quatre bits, alors $1011 + 0101 = 0000$.	☐	☐
7. Avec des mots de quatre bits, on peut représenter tous les entiers relatifs de -8 à 8 bornes comprises.	☐	☐
8. On code les entiers relatifs en complément à deux sur 5 bits. L'entier 10 est codé par 01010 et l'entier -10 par 10101.	☐	☐
9. Les entiers non signés sont codés sur 8 bits. Il y a un dépassement de capacité si on calcule la somme de deux nombres commençant par 1.	☐	☐
10. Les entiers signés sont codés en complément à 2. Si la somme de deux nombres commençant par 0 est un nombre commençant par 1, il y a un dépassement de capacité.	☐	☐

■ ■ Énoncé des exercices

Exercice 11.1 : Les nombres sont écrits en binaire.

1. Calculer la somme $100110 + 001101$ en posant l'addition.
2. Traduire le calcul en décimal.

Exercice 11.2 : Quelle affirmation est exacte ?

1. Un nombre occupe 8 fois moins de place en mémoire s'il est représenté par des octets plutôt que par des bits.
2. Un nombre écrit en hexadécimal comporte 8 fois moins de chiffres que s'il est écrit en binaire.
3. Un nombre pair a une écriture binaire qui se termine par un 0.
4. Un nombre pair a une écriture hexadécimale qui se termine par un 0.

Exercice 11.3 : Quelle est la valeur en binaire de 1001×111 ?

1. 111111
2. 101010
3. 100111
4. 111001

Exercice 11.4 : On utilise des mots de cinq bits pour coder les entiers relatifs en complément à deux. Comment est codé le nombre -2 ?

1. 10010.
2. 01111.
3. 10110.
4. 11110.

Exercice 11.5 : Codage d'un entier naturel

1. Écrire une fonction qui prend en paramètre un entier naturel exprimé en base dix et renvoie l'écriture en base deux de ce nombre. Le paramètre en entrée est de type `int` et la valeur renvoyée en sortie de type `str`.
2. Écrire une fonction qui prend en paramètre un entier naturel exprimé en base deux et renvoie l'écriture en base dix de ce nombre. Le paramètre en entrée est de type `str` et la valeur renvoyée en sortie de type `int`.

Exercice 11.6 : Écrire une fonction `bin_kbits` qui prend en paramètres deux entiers naturels n et k avec $0 \leq n < 2^k$ et renvoie l'écriture de n sur k bits. La valeur renvoyée est de type `str`.

Exercice 11.7 : Un nombre est stocké sous la forme d'un quadruplet, par exemple $(13, 1, 0, 0)$. Les quatre nombres sont les valeurs décimales des quatre octets qui servent à stocker la valeur du nombre. Suivant l'orientation big endian, ce nombre a pour valeur décimale $13 \times 256^3 + 1 \times 256^2 = 218169344$. Suivant l'orientation little endian, ce nombre a pour valeur décimale $13 + 1 \times 256^1 = 269$.

1. Écrire une fonction `decimal_be` qui prend en paramètre un quadruplet comme ci-dessus et renvoie la valeur décimale du nombre en suivant l'orientation big endian.
2. Écrire une fonction `decimal_le` qui prend en paramètre un quadruplet comme expliqué ci-dessus et renvoie la valeur décimale du nombre en suivant l'orientation little endian.

Exercice 11.8 : On utilise cinq bits pour coder les entiers relatifs.

1. Combien de nombres peut-on coder et lesquels ?
2. Comment est codé le nombre 6 ?
3. Comment est codé le nombre -7 ?

Exercice 11.9 : Dépassement

1. Combien de secondes peut-on écrire sur un mot de 32 bits signé ?
2. Combien de secondes (environ) se sont écoulées depuis le 1er janvier 1970 à 0 h ?

Exercice 11.10 : Entiers en Python

1. Donner un ordre de grandeur de la place occupée en mémoire, évaluée en Go, par 2^{30} blocs de quatre octets chacun ?
2. Donner un ordre de grandeur du plus grand entier que l'on peut stocker en mémoire si on dispose de 4 Go.

Exercice 11.11 : Codage d'un entier relatif

Écrire une fonction qui prend en paramètres un entier relatif r exprimé en base dix et un entier naturel non nul n et renvoie le codage de r sur n bits en complément à deux. Le résultat renvoyé est de type `str`. Il n'est pas nécessaire de vérifier que le nombre r peut être codé sur n bits.

Exercice 11.12 : Liste des chiffres

1. Écrire une fonction qui prend en paramètre un entier naturel et renvoie la liste des chiffres de son écriture binaire rangés dans l'ordre inverse. Pour $n = 11$, on obtient la liste `[1, 1, 0, 1]`
2. Écrire une fonction qui prend en paramètre la liste des chiffres de l'écriture binaire d'un entier naturel rangés dans l'ordre inverse et renvoie cet entier naturel.

Exercice 11.13 :** Écrire une fonction `ajoute` qui prend en paramètres deux nombres n et m , la base utilisée b et un entier i et renvoie le nombre $n + m * b^{**i}$. Chaque nombre est représenté par la liste inversée des chiffres de son écriture en base b (comme dans l'exercice précédent).

Exercice 11.14* : Chaque nombre est représenté comme dans les deux exercices précédents.

1. Écrire une fonction `produit1` qui prend en paramètres deux nombres, dont le second s'écrit avec un seul chiffre, et la base utilisée et renvoie le produit des deux nombres.
2. Écrire une fonction `produit` qui prend en paramètres deux nombres et la base utilisée et renvoie le produit des deux nombres. Utiliser les fonctions `ajoute` de l'exercice 13 et `produit1`.

■ ■ Indications

Ex. 11.5

1. *Utiliser l'algorithme des restes avec les divisions euclidiennes successives.*

Ex. 11.13

Poser l'opération pour comprendre les différents mécanismes. Il faut penser à traiter tous les cas, avec les retenues. Tester la fonction en cours d'écriture.

Ex. 11.14

Poser l'opération pour comprendre comment utiliser les fonctions `ajoute` et `produit1`.

■ ■ Corrigé du vrai/faux

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

Quelques explications

4. On ajoute un 0 devant.
5. $170 = 10 \times 16 + 10$ et 10 est représenté par la lettre A.
6. $1011 + 0101$ vaut 10000 et on élimine le premier bit.
7. On ne peut représenter que seize nombres (de -8 à 7 inclus).
8. Pour le codage de -10 , on inverse les bits et on ajoute 1. On obtient 10110.

❑ Erreurs classiques et conseils.—

- En base b , les b chiffres vont de 0 à $b - 1$. Si n est la longueur d'une chaîne de caractères, les indices vont de 0 à $n - 1$. Avec `range(n)`, on obtient tous les entiers de 0 à $n - 1$. Le principe est toujours le même.
- Dans le codage des entiers relatifs avec n bits en complément à deux, on code le nombre -2^{n-1} mais pas le nombre 2^{n-1} .
- Le bit de poids fort, le premier bit, donne une indication sur le signe d'un entier signé.

■ ■ Corrigé des exercices

Exercice 11.1

1. On pose l'addition. La troisième ligne est utilisée pour les retenues.

$$\begin{array}{rcccccc} & 1 & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & & & & \\ \hline 1 & 1 & 0 & 0 & 1 & 1 & \end{array}$$

2. En décimal, $38 + 13 = 51$.

Exercice 11.2

Quelle que soit la représentation, il n'y a que des 0 et des 1 stockés en mémoire. Le nombre 2 est pair et s'écrit 2 en hexadécimal. Son écriture binaire est 10. La bonne réponse est la 3.

Exercice 11.3

On peut écrire : $1001 \times 111 = 1001 \times (100 + 10 + 1)$.

$1001 \times 111 = 100100 + 10010 + 1001 = 111111$ ($9 \times 7 = 63$). La bonne réponse est la 1.

Exercice 11.4

Le nombre 2 est codé en binaire par 00010. On inverse les bits, soit 11101, et on ajoute 1 pour obtenir 11110. Ou alors, on sait que -1 est toujours codé par une suite de 1, donc par 11111 et on retranche 1. La bonne réponse est la 4.

Exercice 11.5

1. On utilise la méthode des divisions euclidiennes successives en conservant les restes dans une chaîne de caractères. Il faut être attentif à l'ordre d'écriture.

La fonction est nommée `base2`.

```
def base2(n):
    if n == 0:
        return "0"
    b = ""
    while n != 0:
        r = n % 2
        n = n // 2
        b = str(r) + b # le reste est ajouté devant la chaîne
    return b
```

2. Il suffit d'une simple boucle. Si $n = (b_n b_{n-1} \dots b_1 b_0)_2$ alors $n = b_n \times 2^n + \dots + b_1 \times 2 + b_0$.

Une autre manière d'écrire le programme est de commencer par transformer l'expression à calculer : $n = 2 \times (2 \times (\dots 2 \times (2 \times b_n + b_{n-1}) + b_{n-2} \dots) + b_1) + b_0$.

Dans ce cas, on initialise une variable $n = 0$, puis on parcourt les bits du nombre et à chaque passage dans la boucle, on multiplie la valeur courante de n par 2 et on ajoute le bit suivant. Les valeurs successives de n sont alors b_n , puis $2 \times b_n + b_{n-1}$, puis $2 \times (2 \times b_n + b_{n-1}) + b_{n-2}$, etc.


```
def base10(b):
    n = 0
    for bit in b:
        n = 2 * n + int(bit)
    return n
```

Exercice 11.6

```
def bin_kbits(n, k):
    b = str()
    q = n
    while q != 0:
        q, r = q // 2, q % 2
        b = str(r) + b
    b = (k - len(b)) * '0' + b
    return b
```

Exercice 11.7

1. On peut remarquer le résultat suivant :

$$x_0 \times 256^3 + x_1 \times 256^2 + x_2 \times 256^1 + x_3 \times 256^0 = x_3 + 256(x_2 + 256(x_1 + 256x_0)).$$

```
def decimal_be(b): # si big endian
    n = len(b)
    d = b[0]
    for i in range(1, n):
        d = b[i] + 256 * d
    return d
```

- 2.

```
def decimal_le(b): # si little endian
    n = len(b)
    d = b[n-1]
    for i in range(1, n):
        d = b[n-1-i] + 256 * d
    return d
```

Exercice 11.8

1. Avec cinq bits pour coder les entiers relatifs, il est possible de coder $2^5 = 32$ nombres, donc les entiers relatifs de -16 à 15 .
2. Sur cinq bits, le nombre 6 est codé par 00110.

3. Sur cinq bits, le nombre -7 est codé par 1 1001. On a inversé les bits du codage de 6.

Exercice 11.9

1. Avec 32 bits, on peut écrire les nombres de -2^{31} à $2^{31} - 1$. La valeur maximale positive est donc $2^{31} - 1 = 2147483647$.

2. Calcul fait le 5 juin 2021 à 11 h, soit après 51 ans, 161 jours et 11 heures :

$$51 \times 365 \times 24 \times 60 \times 60 + 161 \times 24 \times 60 \times 60 + 11 \times 60 \times 60 = 1622286000.$$

Donc, dans moins de 17 ans, on ne pourra plus utiliser un mot de 32 bits signé pour compter les nombres de secondes écoulées depuis le 1er janvier 1970 à 0 h.

Exercice 11.10

On utilise $2^{30} = 8^{10}$ ou l'approximation $2^{10} \simeq 10^3$, d'où $2^{30} \simeq 10^9$.

1. $4 \times 2^{30} = 4 \times (2^{10})^3 \simeq 4 \times (10^3)^3 = 4 \times 10^9$.

Valeur exacte : 32 212 254 720 bits = 4 026 531 840 octets, soit environ 4 Go.

2. Le plus grand entier est de l'ordre de $(2^{30})^{2^{30}} = (8^{10})^{10^9} = 8^{10^{10}}$.

Exercice 11.11

Avec un codage sur n bits, on code les nombres positifs r de 0 à $2^{n-1} - 1$ par l'écriture en binaire de r et les négatifs r de -2^{n-1} à -1 par l'écriture en binaire de $r + 2^n$.

Le programme doit tester si r est négatif.

Il faut penser à ajouter éventuellement des 0 pour obtenir n bits.

```
def relatif(r, n):
    if r < 0:
        r = r + 2 ** n
    if r == 0:
        return n * "0"
    b = ""
    q = r
    while q != 0:
        reste = q % 2
        q = q // 2
        b = str(reste) + b
    return (n - len(b)) * "0" + b
```

Pour tester la fonction utiliser des entiers r et n vérifiant : $-2^{n-1} \leq r < 2^{n-1}$.

Exercice 11.12

1.

```
def chiffres(n):
    if n == 0:
        return [0]
    rep = []
    while n != 0:
        rep.append(n % 2)
        n = n // 2
    return rep
```

2. Une simple boucle permet d'accéder à chaque chiffre.

```
def entier(chiffres):
    n = 0
    for i in range(len(chiffres)):
        n = n + chiffres[i] * 2**i
    return n
```

Exercice 11.13

```
def ajoute(n, m, b, i): # somme de n + m * b**i
    rep = list(n) # rep contient la réponse
    if m == [0]:
        return rep
    for k in range(len(m) + i - len(n)):
        rep.append(0) # si m * b**i plus long que n
    retenue = 0
    for k in range(len(m)): # sommes des chiffres n[i+k] + m[k]
        s = rep[i] + m[k] + retenue
        if s >= b: # traitement de la retenue
            rep[i] = s - b
            retenue = 1
        else:
            rep[i] = s
            retenue = 0
        i = i + 1
    while i < len(n) and retenue == 1: # si n plus long que m * b**i
        s = n[i] + retenue
        if s >= b:
            rep[i] = s - b
            retenue = 1
        else:
            rep[i] = s
            retenue = 0
        i = i + 1
    if retenue > 0: # s'il reste une retenue finale
        rep.append(retenue)
    return rep

a = [1, 1, 0, 1]
b = [1, 0, 0, 1]
assert ajoute([0], b, 2, 0) == b
assert ajoute(a, [1], 2, 0) == [0, 0, 1, 1]
assert ajoute(a, b, 2, 0) == [0, 0, 1, 0, 1]
```


1.

```
def produit1(n, c, b):
    if c == [0]:
        return [0]
    if c == [1]:
        return n
    # en base deux, c'est terminé
    retenue = 0
    prod = len(n) * [0]
    for i in range(len(n)):
        p = n[i] * c[0] + retenue
        prod[i] = p % b
        retenue = p // b
    if retenue > 0:
        prod.append(retenue)
    return prod

assert produit1([1, 0, 1, 1], [0], 2) == [0] # en base 2
assert produit1([1, 0, 1, 1], [1], 2) == [1, 0, 1, 1]
assert produit1([2, 3, 4], [2], 10) == [4, 6, 8] # en base 10
assert produit1([3, 4, 5], [3], 10) == [9, 2, 6, 1]
assert produit1([2, 3, 4], [5], 16) == [10, 15, 4, 1] # en base 16
assert produit1([2, 3, 4], [6], 16) == [12, 2, 9, 1]
```

2. On procède comme lorsqu'on pose l'opération.

```
def produit(n, m, b):
    prod = []
    for i in range(len(m)):
        p = produit1(n, [m[i]], b) # [m[i]] est un nombre à un chiffre
        prod = ajoute(prod, p, b, i)
    return prod

assert produit([1, 0, 1], [1, 1], 2) == [1, 1, 1, 1]
assert produit([1, 1, 1], [1, 1], 2) == [1, 0, 1, 0, 1]
assert produit([7, 2, 8], [5, 2], 10) == [5, 7, 6, 0, 2]
assert produit([4, 1], [4], 16) == [0, 5]
```

Chapitre 12

Les flottants

UN SCIENTIFIQUE



Simon Stevin (1548-1620) est connu à la fois pour ses découvertes en mathématiques et ses talents d'inventeur. Après avoir servi dans l'armée des Pays-Bas il publie en 1585 *L'arithmétique*, un ouvrage dans lequel se trouve exposée pour la première fois la théorie des fractions décimales. On lui doit aussi des travaux sur la statique et sur l'hydrostatique. Comme ingénieur, il conçoit un char à voile et participe à la confection de fortifications.

■ Un peu d'histoire

Bien qu'existant depuis longtemps, la numération de position ne s'implante en Occident qu'à la Renaissance. On peut alors envisager de la généraliser à des nombres non-entiers, ce qui permet d'éviter les fractions.

Simon Stevin établit le premier la notion de développement décimal mais ses notations sont lourdes. En 1592 Giovanni Magini propose d'utiliser un point pour séparer les parties entières et décimales là où Snell et Neper préfèrent mettre une virgule.

L'utilisation des tables de logarithmes décimaux a rendu utile de déterminer les chiffres significatifs d'un nombre puisque, par exemple, les logarithmes des nombres 1,23 et 12,3 ont la même partie décimale, appelée mantisse.

Pour des raisons assez proches, l'expression des nombres en virgule flottante s'est avérée nécessaire pour exprimer les chiffres significatifs d'un nombre, l'ordre de grandeur étant indiqué par ailleurs.

OBJECTIFS

Certaines données ne peuvent pas être représentées par un entier ou un couple d'entiers. Comment représenter des nombres réels quelconques dans une machine ?

- Connaître différents ensembles de nombres :
 - ▶ distinguer les nombres décimaux des nombres flottants ;
 - ▶ maîtriser différentes écritures d'un nombre ;
 - ▶ comprendre pourquoi la majorité des réels ne sont pas représentables dans un mot machine.
- Réfléchir à la représentation d'un réel en machine :
 - ▶ connaître la représentation par signe, mantisse, exposant ;
 - ▶ savoir déterminer une représentation d'un réel dans des cas simples ;
 - ▶ déduire des conséquences de cette représentation.
- Retenir les risques posés par la manipulation de flottants :
 - ▶ être capable d'organiser un calcul pour diminuer les risques d'erreurs ;
 - ▶ savoir comment effectuer la comparaison de flottants.

Nous savons représenter en machine un nombre entier si ce nombre est compris entre deux bornes qui dépendent du nombre d'octets utilisés. Il n'y en a qu'un nombre fini. Pour les nombres réels, c'est plus compliqué. Ces nombres sont de différents types : entier comme 3, décimal comme $-4,25$ ou $0,1$, rationnel comme $1/3$, irrationnel comme $\sqrt{2}$ et π . Il y a en a une infinité même entre deux bornes quelconques. De plus avec des mots de taille fixe, nous ne pouvons stocker qu'un nombre fini de décimales. Donc, même entre deux bornes comme 1 et 2, nous ne pouvons représenter qu'un nombre fini de réels de manière exacte, et pour tous les autres nous utilisons une valeur approchée.

■ Ensembles de nombres

■ En mathématiques

Les ensembles de nombres $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{D}$, $\mathbb{Q}$, $\mathbb{R}$ sont étudiés en mathématiques. Ces ensembles contiennent tous une infinité de nombres. Certains s'écrivent en base dix avec une infinité de chiffres après la virgule et on ne peut en représenter qu'une valeur approchée avec des mots de taille fixe. L'utilisation du binaire limite aussi la représentation de certains nombres puisque des nombres qui s'écrivent en base dix avec un nombre fini de chiffres après la virgule, s'écrivent en base deux avec un nombre infini de chiffres après la virgule.

Une représentation d'un nombre en virgule fixe consiste à représenter un nombre avec un nombre fixe de chiffres après la virgule. On peut écrire par exemple $32,57$ ou $0,86$ ou $-621,74$ en fixant deux chiffres après la virgule dans une représentation décimale en virgule fixe.

Avec une représentation décimale en virgule flottante, (floating-point numbers en anglais), on peut utiliser le nombre $1,34$ pour représenter les nombres $1,34 \times 10^2$ soit 134 , $1,34 \times 10^{-5}$ soit $0,0000134$, etc. Le nombre de chiffres utilisés est fixe, la virgule peut être déplacée.

■ Les nombres décimaux

Les nombres décimaux ont un statut particulier qui est lié à l'écriture en base dix. Un nombre décimal peut s'écrire $n/10^p$, où n et p sont des entiers relatifs. En notation décimale positionnelle, il peut s'écrire avec un nombre fini de chiffres après la virgule. En binaire, on utilise une écriture semblable $n/2^p$. En base trois on utiliserait une écriture comme $n/3^p$.

Par exemple, le nombre décimal qui s'écrit en base dix $15/10$, soit $1,5$, s'écrit en base deux $1111/1010$ ou $11/10$, soit $1,1$. Dans les deux bases, le nombre de chiffres après la virgule est fini. Ce ne serait pas le cas en base trois.

Considérons le nombre $7/3$ qui n'est pas un décimal. Dans une écriture en base dix ou en base deux, le nombre de chiffres après la virgule est infini. Ce n'est pas le cas en base trois puisque ce nombre $(6/3 + 1/3)$ s'écrit $2,1$.

Considérons le décimal $16/10$, soit $1,6$ en base dix. En base deux, le nombre de chiffres après la virgule est infini.

Un nombre qui s'écrit $n/2^p$ peut s'écrire $5^p n/10^p$. Donc si un nombre a une écriture finie en base deux, c'est aussi le cas en base dix.

La réciproque est fausse. Si $n/10^p = m/2^e$, alors $n \times 2^e = m \times 10^p$ et l'arithmétique nous dit que n doit être divisible par 5. Donc parmi les nombres $n/10$ où n est un entier compris entre 10 et 20, seul les nombres $10/10$, $15/10$ et $20/10$ ont une écriture finie en base deux.

En Python, on dispose de deux types de nombres. Les nombres de type `int` représentent des entiers de manière exacte. Ils constituent un ensemble fini inclus dans $\mathbb{Z}$. Avec des mots de n bits, on peut représenter en complément à deux tous les entiers signés de -2^{n-1} à $2^{n-1} - 1$. Les nombres de type `float` servent à représenter de manière exacte ou approchée des nombres réels.

■ Le type float

L'ensemble des nombres de type `float` en Python est inclus dans $\mathbb{D}$. On peut dire aussi que c'est une partie finie de l'ensemble des nombres réels. Ce sont des nombres écrits en virgule flottante avec des mots de taille fixe, et on abrège en flottants. Pour la suite, on appelle *flottant* un nombre de type `float`. Il y a un plus petit nombre de type `float`, un plus grand nombre, et entre les deux, tous les nombres de type `float` sont répartis de manière non uniforme. Par exemple une partie importante de ces nombres est dans l'intervalle $[-1; +1]$.

Le nombre deux est un entier, un décimal, un rationnel, un réel. Il peut s'écrire sous forme entière 2, sous forme décimale 2,0 ou 200/100, mais aussi comme une fraction 6/3, ou avec un radical $\sqrt{4}$. Ce nombre deux en Python est de type `int` si on l'écrit par exemple 2, ou 6//3, et de type `float` si on l'écrit 2.0, 200/100, 200*1e-2, 6/3, `sqrt(4)`.

Le nombre deux est représenté de manière exacte en machine, ce qui entraîne que l'expression `2 == 2.0` vaut `True`. Le nombre décimal 0,1 n'a pas de représentation exacte avec le type `float`. Lorsqu'on écrit 0.1 en Python, c'est le nombre de type `float` le plus proche de 0,1 qui est enregistré en machine et utilisé pour les calculs. C'est pourquoi l'expression `0.1+0.1+0.1 == 0.3` vaut `False` par exemple. Mais aussi parce-que 0.3 n'a pas non plus de représentation exacte en machine avec le type `float`.

En fait 0.3 est représenté en machine par 0.299999999999999889..., et 0.1 est représenté par 0.1000000000000000055... Donc si on calcule `0.1+0.1+0.1` on obtient avec les arrondis la représentation 0.3000000000000000444... qui n'est pas celle de 0.3.

Le seul nombre représenté de manière exacte parmi 0.1, 0.2, 0.3, ..., 0.9 est 0.5. Cela s'explique par le fait que 0,5 peut s'écrire 0,1 en binaire ($= 1/2$), comme 0,25 s'écrit 0,01 en binaire ($= 1/2^2$).

Répartition

L'image des graduations sur une règle, un double décimètre par exemple, donne une idée de la répartition des flottants. Il y a peu de graduations par rapport à l'infinité de nombres réels compris entre 0 et 20. Donc n'importe quel réel compris entre deux graduations sera approché par l'une des deux valeurs.

La principale différence avec la répartition des flottants est que ces derniers ne sont pas répartis de manière uniforme entre deux bornes comme sur la règle, mais ils deviennent de plus en plus rares à mesure qu'on s'éloigne de 0.

Il y en a environ 2^{52} flottants entre 1 et 2 mais il n'y en a aucun entre 9007199254740992 et 9007199254740994 (2^{53} et $2^{53} + 2$).

```
>>> 2.0 == 2
True
>>> 2.0**53 == 2**53
True
>>> 2.0**53+1 == 2**53+1
False
>>> 2.0**53+1 == 2.0**53
True
```


La différence entre deux flottants consécutifs (sur une architecture 64 bits), dans $[1; 2]$ est 2^{-52} , celle entre deux flottants consécutifs dans $[2; 4]$ est 2^{-51} , etc. Celle entre deux flottants consécutifs dans $[2^p; 2^{p+1}]$ est 2^{p-52} . Et donc les flottants supérieurs à 2^{52} sont tous des entiers (au sens mathématique).

La moitié des flottants est composée de flottants négatifs, l'autre moitié de flottants positifs. Parmi les positifs, la moitié est composée de flottants inférieurs à 1, l'autre moitié de flottants supérieurs à 1. Les flottants supérieurs à 1 sont (ou représentent) en grande majorité des entiers, avec des trous de plus en plus grands entre deux flottants. Par exemple il n'y a aucun flottant entre les deux nombres 2^{1023} et $2^{1023} + 2^{970}$:

```
2.0**1023 == 2.0**1023 + 2.0**970
True
```

■ Les nombres de type float

Nous sommes habitués à la notation positionnelle en base dix. Et comme avec les entiers, la notation positionnelle en base b avec $b \geq 2$ suit le même principe.

$$(a_n \dots a_1 a_0 a_{-1} \dots a_{-p})_b = a_n b^n + \dots + a_1 b + a_0 + a_{-1} b^{-1} + \dots + a_{-p} b^{-p}.$$

$$\text{Par exemple : } (213, 03)_4 = 2 \times 4^2 + 1 \times 4^1 + 3 \times 1 + 0 \times 4^{-1} + 3 \times 4^{-2} = 371/16.$$

En base b , nous utilisons b chiffres, de 0 à $b - 1$. Ici encore, il apparaît clairement que le rôle du zéro dans l'écriture positionnelle est capital, par exemple pour distinguer 213,3 de 213,03.

■ Écriture d'un nombre

Exemple d'une écriture d'un nombre à virgule en base deux :

$$101,101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}, \text{ soit } 5,625 \text{ en base dix.}$$

Réciproquement, si on donne l'écriture en base dix, on commence par séparer la partie entière de la partie décimale :

$$13,875 = 13 + 0,875, \text{ et } 13_{10} = 1101_2$$

La méthode consiste ensuite à effectuer des multiplications par deux successives :

$$\begin{aligned} 0,875 &= 1,75 \times 2^{-1} = 1/2 + 0,75 \times 2^{-1} = 1/2 + 1,5 \times 2^{-2} = 1/2 + 1/4 + 0,5 \times 2^{-2} \\ &= 1/2 + 1/4 + 1 \times 2^{-3} = 1/2 + 1/4 + 1/8, \text{ soit } 0,111_2 \text{ en base deux.} \end{aligned}$$

Finalement le nombre s'écrit $1101,111_2$.

Considérons le nombre $1000/1010 = 10^3/(10^3 + 10)$ en base dix.

Ce nombre s'écrit $0,990099009900\dots$

De manière générale, le nombre $1000/1010$ écrit en une base b quelconque, soit $b^3/(b^3 + b)$, s'écrit toujours $0, (b-1)(b-1)00(b-1)(b-1)00(b-1)(b-1)00\dots$

En base deux $1000/1010$ s'écrit $0,110011001100\dots$ puisque $2-1=1$. C'est exactement l'écriture binaire du nombre dont l'écriture décimale est $8/10$. Cette écriture comporte un nombre de chiffres infinis après la virgule et la séquence 1100 est répétée indéfiniment. En divisant par 8, donc en déplaçant la virgule de trois places vers la gauche, on obtient l'écriture de $1/10$ qui est aussi infinie, soit $0,000110011001100\dots$

On peut en déduire que des nombres comme $2/10$, $4/10$ ont une écriture infinie, mais aussi $16/10$, $32/10$, etc. Et plus généralement, tous les nombres de la forme $2^n/10$ en base dix ont une écriture infinie en base deux.

On a une propriété encore plus générale.

Propriété

Si un nombre a une écriture infinie en base deux, alors le nombre obtenu après l'ajout d'un entier, le retrait d'un entier, une multiplication par deux ou une division par deux, a aussi une écriture infinie en base deux.

Par exemple, puisque $8/10$ a une écriture infinie en base deux, on en déduit que c'est le cas pour les nombres suivants : $4/10$, $2/10$, $1/10$, $16/10$ donc $6/10$, $3/10$, $1 + 4/10$ donc $14/10$, donc $7/10$, $1 + 8/10$ soit $18/10$ donc $9/10$.

■ Représentation d'un nombre

La représentation en virgule flottante dans un ordinateur ressemble à la notation scientifique en numération binaire. On représente un nombre x par : un signe (+ ou -), une mantisse a et un exposant p (entier relatif), soit en base dix $x = \pm a \times 10^p$ où $1 \leq a < 10$ et en base deux $x = \pm a \times 2^p$ où $1 \leq a < 2$. En machine le signe est remplacé par 0 ou 1.

Il suffit d'un bit en machine pour coder le signe. Le choix fait est 0 pour les nombres positifs et 1 pour les nombres négatifs.

Observons l'écriture en base deux de quelques nombres décimaux.

2^i	...	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	...
3	0	0	0	1	1	0	0	0	0
5	0	0	1	0	1	0	0	0	0
4,5	0	0	1	0	0	1	0	0	0
0,25	0	0	0	0	0	0	1	0	0

Le nombre 3 s'écrit 11 soit $1,1 \times 10$ en binaire, $(1,5 \times 2)$ en décimal).

Le nombre 5 s'écrit 101 soit $1,01 \times 100$ en binaire, $(1,25 \times 2^2)$ en décimal).

Le nombre 4,5 s'écrit 100,1 soit $1,001 \times 100$ en binaire, $(1,125 \times 2^2)$ en décimal).

Le nombre 0,25 s'écrit 0,01 soit $1,0 \times 0,01$ en binaire, $(1,0 \times 2^{-2})$ en décimal).

Ces quatre nombres peuvent donc s'écrire sous la forme $1, \dots \times 2^p$ en décimal. La partie $1, \dots$ est la mantisse, p est l'exposant. Cette mantisse appartient à $[1; 2[$ et son écriture binaire commence donc toujours par un 1. Il n'est donc pas nécessaire de coder ce 1 en machine.

Prenons le cas du nombre 0,2 et cherchons son écriture binaire. Il s'agit de l'écrire sous la forme $1, \dots \times 2^p$ en décimal. $0,2 = 0,4 \times 2^{-1} = 0,8 \times 2^{-2} = 1,6 \times 2^{-3}$. Il faut donc écrire 0,6 en binaire. Or $0,6 = 1,2 \times 2^{-1}$ et il faut écrire 0,2 en binaire. Nous commençons à "tourner en rond". Ceci signifie que l'écriture binaire nécessite une infinité de décimales exactement comme pour l'écriture en base dix de $1/3 = 1,333333 \dots$. Nous constatons à nouveau que ce nombre ne peut pas avoir une représentation exacte en machine.

Or, $0,2 = 3,2 \times 2^{-4} = 3 \times 2^{-4} + 0,2 \times 2^{-4} = 3 \times 2^{-4} + (3 \times 2^{-4} + 0,2 \times 2^{-4}) \times 2^{-4} = \dots$

Et nous pouvons montrer que $0,2 = 3 \times 2^{-4} + 3 \times 2^{-8} + 3 \times 2^{-12} + 3 \times 2^{-16} + \dots$

Son écriture en binaire est donc 0,0011 0011 0011 $\dots$.

Remarquons que $0,1 = 0,2 \times 2^{-1}$. Donc le problème est identique pour 0,1, il n'a pas de représentation exacte en machine. Son écriture binaire s'obtient à partir de celle de 0,2 en déplaçant la virgule d'un cran vers la gauche (division par deux), soit 0,0001 1001 1001 $\dots$.

Pour représenter les nombres réels sur des mots de taille fixe, nous utilisons donc des valeurs, exactes pour certains nombres, mais approchées pour la plupart.

Une valeur est un nombre décimal qui a une écriture sous forme scientifique en base deux, soit $s m \times 2^p$, avec $1 \leq m < 2$ et $p_{min} \leq p \leq p_{max}$.

Les valeurs de la forme $s m \times 2^p$, avec $0 \leq m < 1$ sont aussi autorisées afin de pouvoir coder le zéro et avoir plus de valeurs proches de zéro.

Normes

Différentes normes existent comme la norme IEEE-754 qui est la plus courante. Elle définit précisément le codage des nombres en virgule flottante et les standards pour les représenter en machine et calculer en binaire. Il ne s'agit pas de retenir les détails de cette norme ou d'une autre, mais de comprendre certaines conséquences des choix qui sont faits pour une norme.

Avec des mots de 64 bits, la norme IEEE-754 précise la règle suivante :

- un bit pour le signe, 0 pour le signe + et 1 pour le signe - ;
- 11 bits pour l'exposant décalé e qui vaut $p + 1023$ avec la condition $-1022 \leq p \leq 1023$, donc $1 \leq e \leq 2046$ (les valeurs 0 et 2047 sont réservées pour coder par exemple $-\infty$ ou $+\infty$) ;
- 52 bits pour la mantisse tronquée m' qui vaut $m - 1$ avec la condition $1 \leq m < 2$.

Ces trois parties sont codées en binaire et concaténées pour former un mot de 64 bits.

Le plus grand nombre flottant a une mantisse égale en binaire à $1,111 \dots 1$ avec 52 chiffres égaux à 1 après la virgule et un exposant égal à 1023. Sa valeur est donc $(1 + 2^{-1} + 2^{-2} + \dots + 2^{-52}) \times 2^{1023}$ soit $2^{1023} + 2^{1022} + \dots + 2^{971}$.

En Python :

```
>>> s = 0
>>> for i in range(971, 1024):
    s = s + 2.0 ** i

>>> s
1.7976931348623157e+308
```

Un nombre réel compris entre ce plus grand nombre et son opposé est alors représenté en machine par le flottant qui en est la meilleure approximation possible.

Exemples

Le réel 20 s'écrit $+1,25 \times 2^4$.

Le signe est + donc le premier bit vaut 0.

L'exposant est 4 donc l'exposant décalé est $4 + 1023 = 1027$, soit 100 0000 0011 en binaire.

La mantisse est 1,25 qui s'écrit en binaire 1,01. On garde la partie décimale 01 et on complète avec des zéros. Le codage de 20 est donc : 0 100 0000 0011 0100 0000 ... 0000.

Le réel -0.375 s'écrit $-1,5 \times 2^{-2}$.

Le signe est - donc le premier bit vaut 1.

L'exposant est -2 donc l'exposant décalé est $-2 + 1023 = 1021$, soit 011 1111 1101 en binaire.

La mantisse est 1,5 qui s'écrit en binaire 1,1. On garde la partie décimale 1 et on complète avec des zéros. Le codage de $-0,375$ est donc : 1 011 1111 1101 1000 0000 ... 0000.

Le réel 0,1 s'écrit $1,6 \times 2^{-4}$. Le signe est +, donc le premier bit vaut 0. L'exposant décalé vaut $-4 + 1023 = 1019$, soit 011 1111 1011. La mantisse de 1,6 se déduit de celle de 0,8 rencontrée auparavant. La mantisse tronquée s'écrit en binaire : 1001 1001 ... 1001 1001 ... Il s'agit de l'arrondir. Comme le 53ème chiffre est un 1, on arrondit par valeur supérieure, (1001,1 est arrondi à 1010), et donc on code : 1001 1001 ... 1001 1010.

La valeur approchée représentant 0,1 est donc : 0011 1111 1011 1001 1001 ... 1001 1010.

Pour calculer $0,1 + 0,1$, nous calculons $1,6 \times 2^{-4} + 1,6 \times 2^{-4} = 1,6 \times 2^{-3}$. Nous effectuons la somme des deux mantisses :

$$\begin{array}{r}
 1, \quad 1 \quad 0 \quad 0 \quad 1 \quad \dots \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \\
 + \quad 1, \quad 1 \quad 0 \quad 0 \quad 1 \quad \dots \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \\
 \hline
 1 \quad 1 \quad \quad \quad 1 \quad 1 \quad \dots \quad 1 \quad \quad \quad 1 \quad 1 \quad \quad \quad 1 \quad 1 \quad \quad \quad 1 \quad \quad \quad \text{retenues} \\
 \hline
 1 \quad 1, \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0
 \end{array}$$

Ce nombre représente $1 + 0,6 + 1 + 0,6 = 2(1 + 0,6)$.

On divise alors le résultat par 2, donc on décale la virgule d'un cran vers la gauche et on supprime le 0 à droite en trop. On obtient alors 1,001 1001 ... 1001 1010 qui est bien la mantisse dans la représentation de 1,6 ou de 0,2.

Calculons $0,2 + 0,1 = 1,6 \times 2^{-3} + 1,6 \times 2^{-4} = 1,6 \times 2^{-3} + 0,8 \times 2^{-3}$. Nous allons ainsi comprendre pourquoi en machine $0,2 + 0,1$ donne 0,30000000000000004. Pour cela nous ajoutons à la mantisse de la représentation de 1,6 la moitié de cette mantisse (pour 0,8).

Nous posons l'addition de ces deux nombres en binaire.

$$\begin{array}{r}
 1, \quad 1 \quad 0 \quad 0 \quad 1 \quad \dots \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \\
 + \quad 0, \quad 1 \quad 1 \quad 0 \quad 0 \quad \dots \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \\
 \hline
 1 \quad 1 \quad \quad \quad 1 \quad 1 \quad \dots \quad \quad \quad 1 \quad 1 \quad \quad \quad 1 \quad 1 \quad \quad \quad \text{retenues} \\
 \hline
 1 \quad 0, \quad 0 \quad 1 \quad 1 \quad 0 \quad \dots \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1
 \end{array}$$

On divise le résultat par 2 donc on supprime le 1 à droite en trop et comme on arrondit, on obtient 1,0011 0011 ... 0011 0100. Or, $0,3 = 1,2 \times 2^{-2}$ et 0,2 s'écrit 0,0011 0011 ... 0011 (voir plus haut). Donc 1,2 est représenté par la mantisse 1,0011 0011 ... 0011 0011 et la différence entre les deux mantisses 1,0011 0011 ... 0011 0100 et 1,0011 0011 ... 0011 0011 est 2^{-52} . En multipliant par 2^{-2} on obtient la différence entre $0,2 + 0,1$ et 0,3 qui est 2^{-54} , soit environ $5,55 \times 10^{-17}$.

Dans l'interpréteur Python :

```
>>> (0.2 + 0.1) - 0.3
5.551115123125783e-17
>>> 2 ** (-54)
5.551115123125783e-17
```

Nous avons des erreurs d'arrondi mais aussi des troncatures.

Reprenons la mantisse trouvée pour $0,2 + 0,1$ et ajoutons la mantisse de 0,1 divisée par 4 pour avoir le même exposant -2 (car $0,1 = 1,6 \times 2^{-4} = 0,4 \times 2^{-2}$).

$$\begin{array}{r}
 1, \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \\
 + \quad 0, \quad 0 \quad 1 \quad 1 \quad 0 \quad \dots \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \\
 \hline
 \quad \quad 1 \quad 1 \quad \quad \quad \dots \quad 1 \quad 1 \quad \quad \quad 1 \quad 1 \quad \quad \quad 1 \quad \quad \quad \text{retenues} \\
 \hline
 1, \quad 1 \quad 0 \quad 0 \quad 1 \quad \dots \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0
 \end{array}$$

Nous obtenons la mantisse représentant 1,6 qui code le nombre $0,4 = 1,6 \times 2^{-2}$. Donc en Python, $(0.2+0.1)+0.1$ donne exactement la représentation de 0.4.

Les résultats qui suivent sont utiles pour les calculs.

- L'écart entre deux nombres flottants consécutifs compris entre 1 et 2 est $\epsilon = 2^{-52}$.
- L'écart entre 2^{52} et 2^{53} est $\epsilon \times 2^{52} = 1$, donc il n'y a aucun flottant entre ces deux nombres.
- Avec un codage sur 64 bits, le plus petit nombre positif non nul est $1,0 \times 2^{-1022}$.

Remarque

Le système qui vient d'être décrit concerne les nombres dit « normaux ». La mantisse de ces nombres appartient à l'intervalle $[1; 2[$ et cela ne permet pas de coder le nombre 0.

C'est pourquoi on définit des nombres dits « subnormaux » pour lesquels la mantisse appartient à l'intervalle $[0; 1[$. Le plus petit nombre positif non nul est alors $2^{-52} \times 2^{-1022} = 2^{-1074}$.

■ Calculs

■ Calculs approchés

La règle pour calculer avec des flottants est la prudence. Les calculs effectués avec des flottants sont des calculs approchés.

L'addition n'est pas associative compte tenu des nombres flottants existants. Par exemple, il n'existe pas de flottants entre 2^{53} et $2^{53} + 2$.

Donc pour la machine, $2.0 ** 53 + 1.0 + 1.0$ vaut $2.0 ** 53 + 1.0$ soit $2.0 ** 53$.

Alors que $2.0 ** 53 + (1.0 + 1.0)$ vaut $2.0 ** 53 + 2.0$.

De manière générale $(a + b) + c \neq a + (b + c)$.

La multiplication n'est pas associative non plus. Par exemple les nombres $(3 \times 10^{-1}) \times 10^{16}$ et 3×10^{15} sont égaux. Mais $3.0 * 10^{**(-1)} * 10^{**(16)}$ et $3.0 * 10^{**(15)}$ ne sont pas égaux. Leur différence vaut exactement 0.5 en machine.

Il est difficile de prévoir un résultat. Par exemple $(1e15 + 1) - 1e15$ a pour valeur 1, alors que $(1e16 + 1) - 1e16$ a pour valeur 0.

Un autre exemple : l'expression $2 * 0.1 == 0.2$ a la valeur `True` alors que 0.1 et 0.2 ne sont pas représentés de manière exacte. Et l'expression $3 * 0.2 == 0.6$ a la valeur `False`.

Des bugs sont possibles à cause de la propagation d'erreurs (arrondis) ou de dépassement de capacité (overflow) qui bloque le programme.

On peut par exemple constater une accumulation d'erreurs en programmant une simple suite.

Voici un code qui calcule les termes d'une suite définie par $u_0 = 1/3$ et pour tout $n > 0$, $u_n = 3(4u_{n-1}/3 - 1/3)$:

```
u = 1/3
for i in range(40):
    u = 3 * ((4*u/3) - 1/3)

print(u)
```

On exécute ce code et on obtient pour u_{40} la valeur 0.3333333333333333. On peut supposer que c'est une approximation de $1/3$ et c'est le résultat attendu.

Cette suite est clairement constante, un calcul montre que tous les termes valent $1/3$. Et on peut vérifier que l'expression $4/9 - 1/3 == 1/9$ a la valeur `True`.

Nous simplifions l'expression pour obtenir $u_n = 4u_{n-1} - 1$ et programmons la suite définie par $u_0 = 1/3$ et $u_{n+1} = 4 \times u_n - 1$.

```
u = 1/3
for i in range(40):
    u = 4 * u - 1

print(u)
```

Nous obtenons pour u_{40} , la valeur : `-22369621.0`.

En fait u_{27} prend la valeur 0.0 et ensuite les termes sont de la forme $u_{n+1} = 4u_n - 1$, donc les valeurs sont -1 , -5 , -21 , etc.

Une explication est donnée par la valeur de l'expression $4/3 - 3/3 == 1/3$ qui est `False`.

Le langage Python, gère des entiers très grands et permet de faire des calculs exacts avec des nombres qui dépassent la valeur maximale utilisée par le processeur. Ces calculs sont décomposés, peuvent nécessiter un temps non négligeable, mais ils sont exacts. Pour les flottants, ce n'est pas le cas, il y a un plus grand nombre.

```
>>> 1.0e308
1e+308
>>> 1.0e309
inf
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1024
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    2.0**1024
OverflowError: (34, 'Result too large')
```

Lorsque dans un calcul la valeur maximale est dépassée, le programme s'arrête et une erreur est signalée : `OverflowError`. Cela peut avoir des conséquences graves comme l'autodestruction de la fusée Ariane 5, lancée le 4 juin 1996, 39 secondes après le décollage. Certains tests n'avaient pas été menés pour des raisons d'économie.

■ Comparaisons

La précision étant limitée, pour comparer deux nombres flottants, on ne teste pas une égalité entre les deux nombres. On vérifie s'ils sont suffisamment proches, ce « suffisamment » dépendant du contexte. En effet, s'il n'y a pas de flottant entre deux flottants a et c , tous les réels compris entre a et c sont représentés par a ou c .

Par exemple, 10.0^{16} et $10.0^{16} + 1$ sont égaux pour la machine. Et dans la résolution de certains problèmes, si on ne prend en compte dans les résultats que six chiffres significatifs, $a = 10.0^{16}$ et $b = 10.0^{16} + 1$ sont considérés comme égaux car $(b - a) \leq b \times 10^{-6}$

Pour comparer des flottants, on utilise une précision absolue et une précision relative. Un test de comparaison a donc la forme : `abs(a-b) <= max(absolue, relative * max(abs(a), abs(b)))`. La précision absolue est utile pour comparer les nombres proches de 0, sinon on peut s'en passer. C'est ainsi qu'est codée la fonction `isclose` dans le module `math`.

```
def estproche(a, b, tol_rel=1e-09, tol_abs=0.0):
    return abs(a-b) <= max(tol_rel * max(abs(a), abs(b)), tol_abs)
```

Pour une tolérance de 5% par exemple, il suffit de prendre `tol_rel = 0.05`.

■ Inégalités

On considère les flottants `x = 2.0 - 2.0 ** (-52)` et `y = 2.0`, soit deux flottants consécutifs tels que `x + y` est le même flottant que `2 * y`.

On obtient alors pour `(x + y) ** 2` la valeur 16.0 et pour `2 * (x**2 + y**2)` la valeur 15.9999...98. On a donc deux nombres x et y tels que $(x + y)^2 > 2(x^2 + y^2)$.

```
>>> x = 2.0 - 2.0 ** (-52)
>>> y = 2.0
>>> x < y
True
>>> x + y == 2 * y
True
>>> (x + y) ** 2
16.0
>>> 2 * (x**2 + y**2)
15.999999999999998
>>> (x + y) ** 2 > 2 * (x**2 + y**2)
True
```

On pourrait en déduire l'inégalité $(x+y)^2 \geq 2(x^2+y^2)$. Or, on démontre exactement le contraire en mathématiques : pour tout réels x et y : $2(x^2 + y^2) \geq (x + y)^2$.

En effet, $(x-y)^2 \geq 0$ soit, $x^2 + y^2 - 2xy \geq 0$, donc, $x^2 + y^2 \geq 2xy$, d'où $2x^2 + 2y^2 \geq x^2 + y^2 + 2xy$. Et enfin $2(x^2 + y^2) \geq (x + y)^2$.

Conclusion

- ▶ Deux réels distincts peuvent être égaux pour la machine.
- ▶ Deux expressions qui produisent le même résultat en mathématiques peuvent produire des résultats distincts en machine.
- ▶ Deux expressions qui produisent des résultats en mathématiques ordonnés dans un sens peuvent produire des résultats ordonnés dans le sens contraire en machine.

La plus grande prudence est donc recommandée.

■ ■ Vrai/Faux

	Vrai	Faux
1. En Python, il y a plus de nombres de type <code>int</code> que de nombres de type <code>float</code> .	☐	☐
2. Tous les décimaux appartenant à $[1; 2]$ ont une représentation exacte en flottant.	☐	☐
3. Avec des mots de n bits, on peut représenter 2^n flottants au maximum.	☐	☐
4. Avec la représentation signe, mantisse, exposant, pour effectuer une division par deux en binaire, on retire à la mantisse le bit de poids faible.	☐	☐
5. Avec la représentation signe, mantisse, exposant, en binaire, une puissance de deux a une mantisse égale à 1.	☐	☐
6. Avec la représentation signe, mantisse, exposant, en binaire, on obtient l'opposé d'un nombre en changeant le bit de poids faible.	☐	☐
7. Avec 1 bit pour le signe, 3 bits pour l'exposant et 6 bits pour la mantisse, on peut coder 512 flottants positifs.	☐	☐
8. L'expression <code>1 + 2.0**(-5) == 1</code> a la valeur <code>True</code> si le codage des flottants utilise une mantisse codée sur quatre bits.	☐	☐
9. Si les flottants sont codés avec une mantisse sur dix bits alors l'expression <code>1 + 10.0**(-3) == 1</code> a la valeur <code>True</code> .	☐	☐
10. Avec deux flottants x et y , si $x < y$, alors $x + y < 2y$.	☐	☐

■ ■ Énoncé des exercices

Exercice 12.1 : L'expression $0.2 + 0.1 > 0.3$ a la valeur True. Quelle en est la raison ?

1. C'est une erreur de la machine.
2. C'est parce-que la machine n'utilise que 32 bits au lieu de 64 pour coder les flottants.
3. C'est parce-que 0,1 ne peut pas être représenté en flottant de manière exacte.
4. C'est parce-que $>$ signifie $\geq$ pour l'interpréteur Python.

Exercice 12.2 : Le programme qui suit calcule 2^i puis 1.01^i pour 30000 valeurs de i entier ; deviner, avant de tester le programme sur machine, quel sera le calcul le plus rapide ?

```
from time import time

top = time()
for i in range(30000):
    c = 2 ** i
print(time() - top)

top = time()
for i in range(30000):
    c = 1.01 ** i
print(time() - top)
```

Tester le programme. Quelle peut être l'explication du comportement observé ?

Exercice 12.3 : On représente des nombres réels avec une écriture signe, mantisse, exposant en binaire. On utilise six bits $b_0b_1b_2b_3b_4b_5$. Le bit b_0 représente le signe : 0 pour +, 1 pour -. Les bits $b_1b_2b_3$ représentent la mantisse tronquée. Les bits b_4b_5 représentent l'exposant. Par exemple, l'écriture 010010 représente un nombre positif. Sa mantisse vaut 1,100 soit 1,5 en décimal et l'exposant 10 vaut 2. Le nombre représenté est donc $1,5 \times 2^2$ soit 6.

1. Quel est le plus petit nombre positif et le plus grand nombre que l'on peut écrire ainsi ?
2. Combien de nombres appartenant à l'intervalle $[1; 2[$ peut-on écrire ?
3. Quel est le successeur de 2 ?

Exercice 12.4 : Les flottants sont utilisés pour des calculs approchés. L'algorithme de Briggs par exemple est employé pour le calcul d'une valeur approchée du logarithme d'un nombre.

Il est basé sur l'approximation : $\log u \simeq u - 1$ pour u suffisamment proche de 1. Si $x^{1/2^n} - 1 \simeq 0$, alors $\log x^{1/2^n} \simeq x^{1/2^n} - 1$, d'où $(1/2^n) \log x \simeq x^{1/2^n} - 1$. On en déduit : $\log x \simeq 2^n(x^{1/2^n} - 1)$.

Programmer cet algorithme et le tester avec quelques valeurs.

Exercice 12.5 : Pour calculer une valeur approchée de $\ln 2$, l'algorithme de Brouncker utilise l'égalité $\ln 2 = \int_1^2 1/x \, dx$. Un découpage sous la courbe entre $x = 1$ et $x = 2$ permet après calculs d'obtenir une valeur approchée de l'intégrale : $\sum_{k=1}^n \frac{1}{(2k-1)2k}$. Programmer cet algorithme.

Exercice 12.6 : Écrire un programme permettant de trouver la plus grande puissance de deux, soit $x = 2^p$, telle que les flottants représentant x et $x + 1$ ont la même valeur. Trouver à l'aide d'un programme le flottant successeur de x .

Exercice 12.7 : Écrire le nombre 3,625 en binaire.

Exercice 12.8 : Les flottants sont codés suivant la norme IEEE-754 sur 64 bits, soit 1 bit pour le signe, 11 bits pour l'exposant décalé et 52 bits pour la mantisse tronquée.

1. Comment est codé le nombre -4.5 ?
2. Quel est le nombre réel codé par 1011 1111 1110 1000 0000 ... 0000 ?

Exercice 12.9 :** Codage d'un flottant

Écrire une fonction qui détermine et renvoie le codage d'un flottant x qui est exprimé en base dix. L'entrée en paramètre est de type float et la sortie de type str. On utilise le codage sur 64 bits présenté dans le cours. Les différentes étapes sont décrites ci-dessous :

- Le paramètre de la fonction, de type float, représente un nombre x .
- Le signe de x est déterminé et stocké par $s = '0'$ ou $s = '1'$ puis x est changé en $|x|$.
- Calcul de l'exposant et de la mantisse. Pour cela, si $x \geq 2$, effectuer des divisions par 2 successives, si $x < 1$ effectuer des multiplications par 2 successives, en remplaçant à chaque fois la valeur de x par le résultat obtenu, et dans les deux cas jusqu'à obtenir un nombre x tel que $1 \leq x < 2$. L'exposant est alors le nombre de divisions ou de multiplications effectuées et la mantisse est le nombre x final.
- Calcul de l'exposant décalé qui est codé en binaire sur 11 bits (voir chapitre 11). Le résultat est stocké dans une chaîne e .
- Calcul de la mantisse tronquée $x = x - 1$ qui doit être écrite en binaire sur 52 bits et stockée dans une chaîne m . Pour cela, multiplier x par 2; si $x \geq 1$, ajouter '1' à m et retrancher 1 à x , sinon ajouter '0' à m ; reproduire ce schéma 52 fois.
- La chaîne concaténée $s + e + m$ est renvoyée.

■ ■ Indications

Ex. 12.6 _____

Le flottant successeur de x a une valeur entière.

Ex. 12.7 _____

Séparer la partie entière et la partie décimale.

Ex. 12.9 _____

Écrire la fonction par étape et la tester après chaque étape.

■ ■ Corrigé du vrai/faux

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

Quelques explications

4. On retranche 1 à l'exposant (si c'est possible).
6. C'est le bit de poids fort qu'il faut changer.
7. $2^{3+6} = 2^9 = 512$ flottants positifs.
8. La précision entre 1 et 2 est limitée à 2^{-4} .
9. La précision entre 1 et 2 est limitée à 2^{-10} et $10^{-3} > 2^{-10}$.
10. Voir l'exemple du cours.

❑ Erreurs classiques et conseils.—

- Avec n'importe quel type de codage sur des mots de taille fixe, on ne peut coder qu'un nombre fini de réels.
- En général, un flottant n'est qu'une valeur approchée d'un réel. Les calculs ne peuvent donc pas être exacts.
- Il faut être très attentif aux comparaisons entre flottants et en général tester avec une égalité n'est pas une bonne idée.

■ ■ Corrigé des exercices

Exercice 12.1

La somme des deux flottants $0.2 + 0.1$ donne 0.30000000000000004 . C'est la réponse 3.

Exercice 12.2

Il semble plus simple de calculer 2^i que 1.01^i . Mais Python peut calculer avec des entiers très grands et pour les calculs de 2^i , une grande partie du temps est utilisée pour stocker et manipuler ces entiers. Pour le calcul de 1.01^i , Python utilise des valeurs approchées et calcule donc très rapidement. (Plus d'une seconde pour le calcul des 2^i et 0,01 s pour le calcul des 1.01^i).

Exercice 12.3

1. Le plus petit nombre positif s'écrit 0000000, c'est 1 ($1,0 \times 2^0$).

Le plus grand nombre s'écrit 011111, c'est 15 ($1,875 \times 2^3$).

2. La mantisse est dans l'intervalle $[1; 2[$. Donc le signe doit être positif ($b_0 = 0$) et l'exposant doit être nul ($b_4 b_5 = 00$). La mantisse tronquée $b_2 b_3 b_4$ peut prendre huit valeurs de 000 à 111. Il y a donc huit nombres appartenant à $[1; 2[$. Le plus petit est 1, le plus grand est 1,875.

3. Le nombre 2 s'écrit $1,0 \times 2^1$, donc est représenté par 000010. Son successeur est représenté par 000110 qui représente $1,125 \times 2^1 = 2,25$.

Exercice 12.4

```
def briggs(x):
    n = 0
    while abs(x - 1) > 0.0001:
        x = x ** 0.5
        n = n + 1
    x = x - 1
    return x * 2 ** n

from math import log
print(briggs(2), log(2))
print(briggs(3), log(3))
```

Exercice 12.5

```
def brouncker(n):
    s = 0
    for i in range(n):
        s = s + 1 / (2*i+1) / (2*i+2)
    return s

from math import log
print(brouncker(1000), log(2))
```

```
def puissance():
    x = 1.
    while x + 1 != x:
        x = x * 2
    return x

def successeur(x):
    s = 1.
    while x + s == x:
        s = s + 1
    return x + s
```

La fonction `puissance` renvoie le nombre 9007199254740992.0, soit 2^{53} .

La valeur de `successeur(puissance())` est 9007199254740994.0, soit $2^{53} + 2$.

Le résultat peut dépendre de l'architecture de la machine.

Exercice 12.7

En binaire, 3 s'écrit 11. Il reste à écrire la partie décimale 0,625. Pour cela on multiplie par 2, on obtient 1,25. On garde le 1 et on recommence avec 0,25. On multiplie par 2, on obtient 0,5. On garde le 0 et on recommence avec 0,5. On multiplie par 2, on obtient 1,0, on garde le 1 et c'est terminé. Les nombres gardés constituent dans l'ordre la partie après la virgule de 0,101 qui est l'écriture binaire de 0,625. Donc l'écriture binaire de 3,625 est 11,101.

On peut aussi remarquer que $3,625 = 3 + 0,5 + 0,125 = 3 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$.

Exercice 12.8

1. Codage du nombre -4.5 :

$-4.5 = -1,125 \times 2^2$; donc 1 pour le signe négatif, puis 100 0000 0001 pour l'exposant décalé ($2 + 1023 = 1025$), et 0010 0 ... 0 pour la mantisse tronquée ($0,125 = 1/8$).

Conclusion : le codage est 1100 0000 0001 0010 0 ... 0

2. Nombre réel codé par 1011 1111 1110 1000 0000 ... 0000 :

le 1er bit est 1 donc le signe est $-$, ensuite l'exposant décalé vaut 1022 donc l'exposant est $1022 - 1023 = -1$, enfin la mantisse tronquée vaut $1/2 = 0,5$.

Conclusion : le nombre est $-1,5 \times 2^{-1} = -0,75$.

Exercice 12.9

```
def codage(x):
    # écriture du signe
    s = '0'
    if x < 0:
        s = '1'
        x = -x
    # calcul de l'exposant et de la mantisse
    exp = 0
    while x >= 2:
```



```

        x = x / 2
        exp = exp + 1
    while x < 1:
        x = x * 2
        exp = exp - 1
    # exposant décalé
    exp = exp + 1023
    # en binaire sur 11 bits
    e = ''
    while exp > 0:
        e = str(exp % 2) + e
        exp = exp // 2
    e = (11 - len(e)) * '0' + e
    # mantisse tronquée
    x = x - 1
    m = ''
    for i in range(52):
        x = x * 2
        if x >= 1:
            m = m + '1'
            x = x - 1
        else:
            m = m + '0'
    return s + ' ' + e + ' ' + m

```

On peut tester avec les nombres de l'exercice précédent en écrivant par exemple `codage(-4.5)`.

■ Les graphes

Vocabulaire et représentations

UN SCIENTIFIQUE



Le mathématicien français **Claude Berge** (1926-2002) s'intéresse à la théorie des jeux. Après avoir passé l'année 1956 à l'université de Princeton, il jette les bases de la théorie moderne des graphes. En 1960, il travaille au CNRS à la mise au point de programmes sur les jeux d'échecs. Berge a tissé des liens entre les mathématiques et la littérature en participant à la création de l'Oulipo avec entre autres Raymond Queneau et Georges Perec.

■ Un peu d'histoire

Le mathématicien suisse Leonhard Euler montre, en 1735, l'impossibilité de passer une fois et une seule par chacun des sept ponts de la ville de Königsberg en revenant à son point de départ. Ce problème est souvent considéré comme la naissance de la théorie des graphes. Le physicien Gustav Kirchhoff utilise les graphes en 1847 pour résoudre un problème d'électricité qu'Arthur Cayley élargira.

Pourtant, la notion de graphe a longtemps été considérée comme un simple amusement mathématique. Le développement de l'informatique et de l'algorithmique depuis les années 1950 a permis un essor très important à cette discipline auquel ont participé d'éminents mathématiciens comme Claude Berge. On trouve des applications de cette théorie dans d'innombrables domaines comme les réseaux informatiques mais aussi la rationalisation des services de distribution.

OBJECTIFS

La théorie des graphes est utilisée en particulier pour répondre à des questions sur différents types de réseaux devenus omniprésents aujourd'hui. Seule l'informatique permet traiter des réseaux dont les tailles sont gigantesques.

- Comprendre la structure relationnelle d'un graphe :
 - ▶ reconnaître cette structure ;
 - ▶ être capable de modéliser une situation à l'aide d'un graphe ;
 - ▶ choisir une structure de graphe orienté ou pas pour répondre à un problème.
- Connaître le vocabulaire des graphes :
 - ▶ traduire des situations avec le vocabulaire des graphes ;
 - ▶ distinguer les graphes pondérés, orientés ;
 - ▶ comprendre les notions de connexité, de cycle.
- Être capable d'implémenter un graphe :
 - ▶ savoir manipuler une matrice d'adjacence et des listes d'adjacence ;
 - ▶ comprendre les avantages et inconvénients en termes de complexité ;
 - ▶ savoir passer d'une implémentation à une autre.

■ Définitions, vocabulaire

■ Graphe en mathématiques

Un graphe est un objet mathématique. Cet objet est constitué d'un ensemble de points noté S et d'un ensemble de couples (x, y) ou de paires $\{x, y\}$ avec x et y éléments de S , noté A .

Un couple est ordonné, une paire ne l'est pas. Avec des couples, nous parlons de graphe orienté, avec des paires de graphe non orienté.

Un couple (x, y) traduit une relation entre x et y . Cela n'entraîne pas l'existence d'une relation entre y et x .

Une paire $\{x, y\}$ traduit une relation symétrique entre x et y . Si x est en relation avec y alors y est en relation avec x .

■ Vocabulaire général

- Un *graphe* est constitué de *sommets* et d'*arêtes*.
- L'*ordre* d'un graphe est le nombre de sommets.
- Si nous supposons un sens de parcours des arêtes, nous disons que le graphe est *orienté*. Les arêtes ont un *sens*. Dans ce cas une arête s'appelle aussi un *arc*.
- Une arête, ou un arc si le graphe est orienté, relie deux sommets qui sont dits *adjacents*. Ces deux sommets sont les *extrémités* de l'arête. On dit aussi que les deux sommets sont *voisins*.
- Si le graphe est orienté, une arête a une extrémité initiale, (ou un sommet de départ, ou une origine, ou un prédécesseur), et une extrémité finale, (ou un sommet d'arrivée, ou une extrémité, ou un successeur).
- Le nombre de voisins d'un sommet est son *degré*. C'est le nombre d'arêtes liées à ce sommet. On dit que ces arêtes lui sont *incidentes*. Un sommet peut avoir un degré 0 (aucune arête), on dit qu'il est *isolé*.
Si un graphe est orienté, un sommet a alors un degré *entrant* (nombre d'arcs dont il est l'extrémité finale) et un degré *sortant* (nombre d'arcs dont il est l'extrémité initiale). Son degré est la somme de son degré entrant et de son degré sortant.
- Si deux sommets quelconques sont adjacents, (un sommet quelconque est relié directement à tous les autres), le graphe est *complet*.
- Une arête du type $\{x, x\}$ s'appelle une *boucle*.
- Un graphe est simple si deux sommets quelconques sont reliés par au plus une arête.

Dans un graphe non orienté, une arête ne peut relier que deux sommets distincts. Il n'y a pas de boucle.

Notation

Un graphe est défini par un ensemble de sommets et un ensemble d'arêtes. Donc on note souvent un graphe $G(S, A)$, avec S l'ensemble des sommets et A l'ensemble des arêtes, ou $G(V, E)$, avec V pour *vertex* (sommet en anglais, vertices au pluriel) et E pour *edge* (arête).

Le degré d'un sommet s se note $d(s)$. Si le graphe est orienté, le degré sortant se note $d_+(s)$ et le degré entrant se note $d_-(s)$.

On appelle aussi les sommets des *nœuds* ou des *points*, et les arêtes des *liens* ou des *lignes*.

Un graphe permet de modéliser un *réseau* : réseau de transport (routier, ferroviaire, aérien), réseau internet, réseau social, réseau électrique.

■ Chemin, connexité

On se déplace dans un graphe en passant successivement d'un sommet à un sommet adjacent. On considère un graphe non orienté.

- Un chemin de s_0 à s_n est une séquence $(s_0, s_1, \dots, s_n)$ où deux sommets consécutifs sont adjacents. Ce chemin permet de relier s_0 à s_n où s_0 est le sommet de départ et s_n est le sommet d'arrivée.

Un chemin peut être donné par la séquence des sommets ou par la séquence des arêtes empruntées $(a_1, a_2, \dots, a_n)$.

- La longueur d'un chemin $(s_0, s_1, \dots, s_n)$ est le nombre d'arêtes utilisées pour aller du sommet de départ s_0 au sommet d'arrivée s_n , c'est donc n .
- La distance entre deux sommets est la longueur minimale d'un chemin reliant ces deux sommets.
S'il n'existe pas de chemin reliant deux sommets, la distance entre ces deux sommets est infinie.
- Un graphe est *connexe* si pour tout couple de sommets il existe un chemin reliant ces deux sommets.
- Un *cycle* est un chemin tel que le sommet de départ et le sommet d'arrivée sont identiques.

Dans un graphe orienté, une arête a un sens et ne peut être empruntée que dans ce sens. On parle alors de *chaîne* plutôt que de chemin pour relier deux sommets.

■ Poids, étiquette

Ajouter des poids ou des étiquettes aux arêtes d'un graphe apporte des informations supplémentaires.

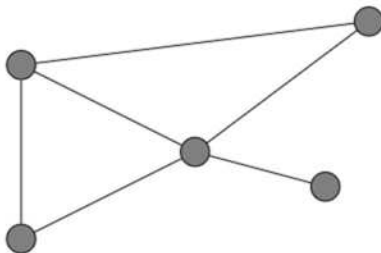
- Un graphe est *pondéré* si un nombre, un *poids*, est associé à chaque arête ou à chaque arc.
- Un graphe est *étiqueté* si un texte, une *étiquette*, est associé à chaque arête ou à chaque arc.

Dans un réseau routier par exemple, un poids peut être le nombre de kilomètres d'une route liant deux lieux, une étiquette peut être le nom de cette route.

■ Représentations

■ Représentation schématique

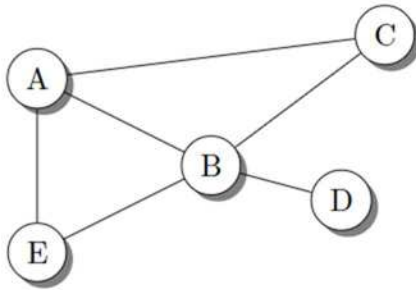
La manière la plus simple de représenter un graphe est de faire un dessin. Les sommets sont représentés par des points, les arêtes par des lignes, chacune reliant deux points.



Ce graphe est non orienté et a cinq sommets. Il est connexe et d'ordre 5.

Les sommets sont représentés par des points mais attention, un sommet peut représenter une entité complexe : une personne dans un réseau social, un carrefour dans un réseau routier, un routeur dans un réseau internet.

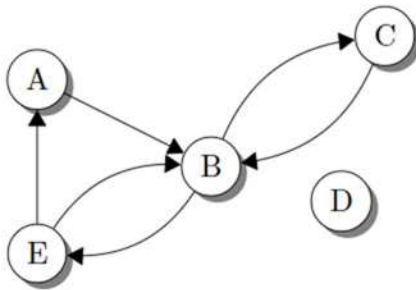
Il est plus pratique de nommer les sommets.



Le degré du sommet A est 3. Ses voisins sont B, C et E. L'arête [AB] a pour extrémités A et B. [AC] est un chemin de longueur 1, [AEBD] est un chemin de longueur 3, [ABD] est un chemin de longueur 2. [ACBEA] est un cycle.

L'ensemble des sommets est $S = \{A, B, C, D, E\}$.

L'ensemble des arêtes est $A = \{\{A, B\}, \{A, C\}, \{A, E\}, \{B, C\}, \{B, D\}, \{B, E\}\}$.



Ce graphe est orienté, non connexe. Le sommet D est isolé. Le degré entrant de B est 3, le degré sortant est 2. [ABC] est une chaîne. [ABEA] est une chaîne fermée.

■ Listes et matrice d'adjacence

On peut représenter un graphe non orienté en précisant pour chacun de ses sommets la liste de ses voisins. Ces listes s'appellent des *listes d'adjacence*. L'ordre d'écriture n'a pas d'importance.

Dans le cas de graphes orientés, on peut présenter des *listes de successeurs* ou des *listes de prédécesseurs* ou les deux.

On obtient pour l'exemple de graphe non orienté dessiné plus haut les listes d'adjacence qui suivent :

A : B, C, E
B : A, C, D, E
C : A, B
D : B
E : A, B

On obtient pour l'exemple de graphe orienté dessiné plus haut les listes de successeurs suivantes :

A : B
B : C, E
C : B
D :
E : A, B

En mathématiques, on peut associer à un graphe une matrice carrée (n, n) , ou un tableau, où n est le nombre de sommets. Les sommets sont numérotés de 1 à n . Il s'agit d'un tableau à n lignes et n colonnes. À l'intersection d'une ligne i et d'une colonne j le nombre représente la présence ou l'absence d'une arête entre les sommets i et j : 1 pour la présence, 0 pour l'absence.

La matrice correspondant au graphe non orienté dessiné précédemment est la suivante :

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	0	0	0
E	1	1	0	0	0

La diagonale ne contient que des 0 et est un axe de symétrie du tableau. On dit que la matrice est symétrique.

Le même graphe peut être représenté par des matrices différentes. Elles dépendent de l'ordre des sommets qui est pris en compte.

En mathématiques, la matrice se note ainsi :

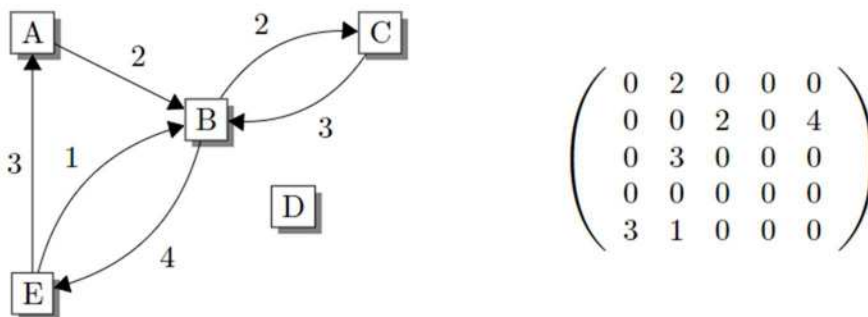
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

On appelle cette matrice une *matrice d'adjacence*.

Dans le cas du graphe orienté présenté plus haut, on obtient la matrice suivante qui n'est pas symétrique :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Dans le cas de graphe pondéré ou étiqueté, on place les informations le long des arêtes. On peut utiliser la matrice en remplaçant les 1 par les poids par exemple.



Pour les listes d'adjacence ou de successeurs, on complète avec les informations. Avec l'exemple précédent on obtient :

A : (B, 2)
 B : (C, 2), (E, 4)
 C : (B, 3)
 D :
 E : (A, 3), (B, 1)

Il faut savoir passer des listes d'adjacence à la matrice d'adjacence et réciproquement.

Listes d'adjacence :

A : C, D
B : C
C : A, B, D
D : A, C

Matrice d'adjacence :

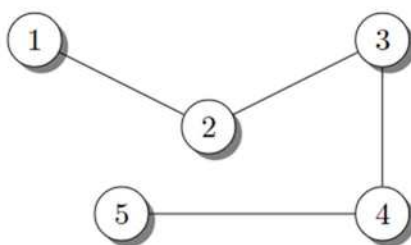
$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

■ Exemples particuliers

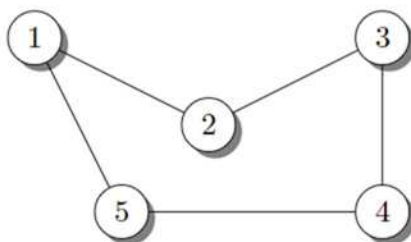
La représentation par un dessin permet sur des cas très simples de reconnaître une propriété.

Considérons un graphe à n sommets numérotés de 1 à n .

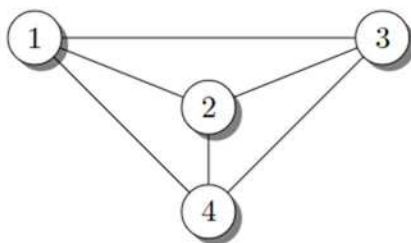
- Si, pour $i \neq 1$ et $i \neq n$, chaque sommet i est adjacent au sommet $i - 1$ et au sommet $i + 1$ et uniquement à ces deux sommets, et si les sommets 1 et n ne sont pas adjacents, on dit que le graphe est une chaîne. Le nombre d'arêtes vaut $n - 1$.



- Si, pour $i \neq 1$ et $i \neq n$, chaque sommet i est adjacent au sommet $i - 1$ et au sommet $i + 1$ et uniquement à ces deux sommets, et si les sommets 1 et n sont adjacents, on dit que le graphe est un cycle. Le nombre d'arêtes vaut n .



- Si chaque sommet est voisin de chacun des autres sommets, le graphe est dit complet.



Si n est le nombre de sommets, alors le nombre d'arêtes est $n(n - 1)/2$.

■ Structures de données

Une *structure de donnée* permet de représenter des données dans une machine. Elle définit les opérations qui permettent de les manipuler, leur stockage en mémoire, et présente différentes caractéristiques.

Par exemple, la structure de liste en Python est linéaire, non homogène, dynamique et mutable : les éléments sont ordonnés, de type quelconque, la longueur et les éléments sont modifiables.

Il existe d'autres structures linéaires comme les tuples, les chaînes de caractères, les piles et les files présentées au chapitre 14, mais aussi des structures non linéaires. Parmi celles-ci, les plus courantes sont les arbres et les graphes.

Pour les graphes, nous parlons de structure relationnelle. En effet un graphe est défini par un ensemble d'éléments et un ensemble de relations entre ses éléments. Une relation est une paire d'éléments.

Une structure indexée est utilisée pour les listes en Python. On peut utiliser une structure récursive pour les graphes.

Le choix d'une structure de données dépend du type des informations utilisées, de la manière de stocker une donnée, de la quantité de mémoire à utiliser. La structure doit être adaptée à des algorithmes qui utilisent des opérations comme la lecture, l'insertion, la suppression de donnée.

■ Implémentation

■ Avec la matrice d'adjacence

En Python, on représente généralement une matrice (n, n) par une liste contenant n listes de longueur n . Chacune de ces n listes représente une ligne de la matrice. Les éléments de chacune de ces n listes sont les n coefficients d'une ligne. Si on numérote les lignes de la matrice de 1 à n , il faut faire attention aux indices qui en Python commencent à 0.

On définit donc un graphe en écrivant par exemple $G = [[0, 0, 1, 1], [0, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]$ si le graphe est défini par la matrice d'adjacence :

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Dans le cas d'un graphe orienté ou pondéré, le principe est le même. Pour un graphe pondéré, les coefficients 1 sont remplacés par les poids.

■ Avec les listes d'adjacence

Si nous utilisons les listes d'adjacence, nous avons plusieurs possibilités de représentation en Python.

- Avec une liste de listes : chaque élément de la liste est une liste contenant un sommet et la liste de ses voisins. Par exemple :

```
G = [{"A", ["B", "E"]}, {"B", ["A", "C", "E"]}, {"C", ["B"]}, {"D", []}, {"E", ["A", "B"]}]
```

- Avec un dictionnaire où les clés sont les sommets et les valeurs correspondant aux clés sont les listes de voisins.

$G = \{ "A" : ["B", "E"], "B" : ["A", "C", "E"], "C" : ["B"], "D" : [], "E" : ["A", "B"] \}.$

Dans le cas d'un graphe orienté, on peut utiliser les deux types de représentation ci-dessus.

Si le graphe est pondéré, on complète les listes avec les poids. Exemple d'un graphe non orienté pondéré : $G = \{ "A" : [("B", 2), ("E", 5)], "B" : [("A", 2), ("C", 1), ("E", 3)], "C" : [("B", 1)], "D" : [], "E" : [("A", 5), ("B", 3)] \}.$

Dans l'exemple qui suit, un graphe non orienté est implémenté à l'aide d'un dictionnaire et le code est complété par une fonction `deg` qui renvoie le degré d'un sommet.

```
graphe = {
    'A': ['B', 'E'],
    'B': ['A', 'C', 'E'],
    'C': ['B'],
    'D': [],
    'E': ['A', 'B']
}

def deg(g, s): # g est un graphe, s est un sommet
    if s in g:
        return len(g[s])
```

Ce graphe présente 5 sommets et 4 arêtes. Nous testons la fonction `deg` :

```
>>> deg(graphe, 'A')
2
>>> deg(graphe, 'B')
3
>>> deg(graphe, 'D')
0
>>> deg(graphe, 'U')
>>>
```

Nous pouvons créer des fonctions pour ajouter des sommets et des arêtes, obtenir l'ensemble des sommets et l'ensemble des arêtes. Ces fonctions sont proposées en exercice ainsi que le passage d'une matrice d'adjacence aux listes d'adjacence et réciproquement.

Remarque

Si on suppose que tous les sommets sont numérotés $0, 1, \dots, n-1$, alors on peut représenter le graphe par une liste dont les éléments sont des listes indicées par les indices $0, 1, \dots, n-1$. La liste d'indice i est la liste des voisins du sommet i .

Pour la traduction en langage Python, les sommets sont de type `str`. Par exemple :

```
sommets = ['A', 'B', 'C', 'D', 'E']
voisins = [['B', 'E'], ['A', 'C', 'E'], ...]
```


Pour obtenir les voisins de 'B' par exemple, il faut parcourir la liste des sommets pour trouver l'indice i de 'B' puis demander dans la liste des voisins l'élément d'indice i .

La représentation par $G = [['A', ['B', 'E']], ['B', ['A', 'C', 'E']], \dots]$ consiste à regrouper les sommets et les voisins.

Mais la recherche des voisins d'un sommet n'est pas plus efficace. Ce qui est le cas par contre avec un dictionnaire car l'accès à une clé est optimisé.

On définit un graphe : $G = \{ 'A': ['B', 'E'], 'B': ['A', 'C', 'E'], \dots \}$.

Il suffit alors d'écrire $G['B']$ pour obtenir les voisins de 'B' en temps constant.

■ Applications

Le mathématicien Suisse Leonhard Euler (1707-1783) a grandement participé au développement de la théorie des graphes. Il a résolu en particulier le *problème des sept ponts de Königsberg* (1735).

De nombreuses situations dans la vie courante se modélisent sous forme de graphes.

Dans un réseau social les liens d'amitié constituent les arêtes d'un graphe. Les utilisateurs sont représentés par les sommets du graphe. Si la distance entre deux sommets vaut 1, les deux sommets sont voisins donc les deux utilisateurs sont amis.

Un réseau routier avec des points de passages et des distances entre chaque point peut être modélisé par un graphe. On cherche alors le nombre minimal de points par lesquels passer pour aller d'une ville à une autre ou le nombre minimal de kilomètres entre deux villes données. Les mêmes types de questions se posent avec un réseau constitué par des stations de métro ou d'autobus. C'est aussi le cas avec un réseau électrique.

En bio-informatique on utilise un graphe d'intervalles : un intervalle est représenté par un sommet, une arête relie deux sommets si les intervalles ont une intersection non vide.

On utilise des graphes en chimie pour représenter des molécules depuis Arthur Cayley en 1874.

Le système du World Wide Web fonctionne avec des pages web reliées entre elles par des liens hypertextes. Les pages sont les sommets et les liens sont les arêtes. Ces liens sont pondérés afin de donner une importance à chaque page. On peut utiliser pour cela le nombre de demandes d'une page, le nombre de liens qui conduisent à cette page, le nombre de liens contenus dans cette page, ou tout autre critère pertinent.

Le réseau ARPANET, ancêtre de l'Internet, date de 1969, un peu plus de 50 ans. C'est le premier réseau qui a utilisé un système de transmission toujours en vigueur sur Internet.

Le réseau Internet est découpé en sous-réseaux eux-mêmes découpés en sous-réseaux, etc. Des routeurs permettent de connecter différents réseaux entre eux et chaque machine est reliée à un sous-réseau. Une machine est un sommet, une connexion (filaire, wifi, etc) est une arête. Plusieurs protocoles entre en jeu pour transmettre des données entre deux machines. L'objectif est de choisir la meilleure route entre deux machines.

Le fonctionnement est comparable à celui du réseau postal permettant de transmettre un objet entre deux points du monde.

Que ce soit sur Internet, dans un réseau postal ou un réseau routier, un objectif est le déplacement (le plus rapidement possible) d'un point à un autre. Autrement dit, il s'agit de parcourir un chemin dans un graphe.

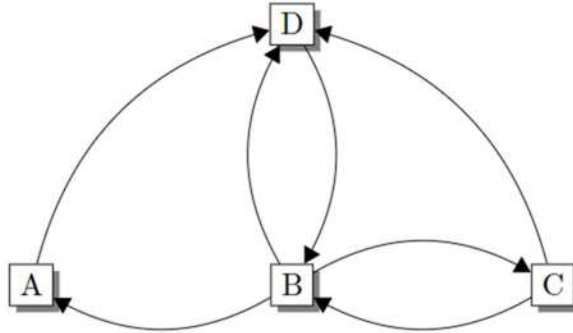
Le nombre d'appareils connectés sur Internet par exemple, donc de sommets d'un graphe, se comptant en milliards, il est capital d'avoir des algorithmes de parcours d'un graphe qui soient les plus efficaces possibles.

■ ■ Vrai/Faux

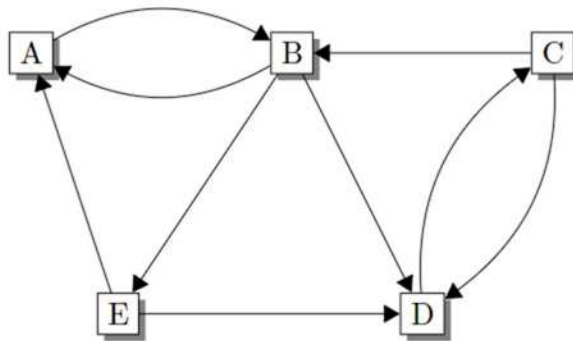
	Vrai	Faux
1. Un graphe ne peut pas avoir plus de sommets que d'arêtes.	☐	☐
2. Un graphe est une structure linéaire.	☐	☐
3. Une matrice d'adjacence d'un graphe à n sommets contient $2n$ coefficients.	☐	☐
4. Une matrice d'adjacence d'un graphe orienté est symétrique.	☐	☐
5. Le nombre d'arêtes d'un graphe à n sommets est la moitié de la somme des degrés des sommets.	☐	☐
6. L'ordre d'un graphe est le nombre d'arcs ou d'arêtes.	☐	☐
7. Un graphe est connexe si chaque sommet est adjacent à tous les autres.	☐	☐
8. Un chemin qui passe deux fois par le même sommet contient un cycle.	☐	☐
9. Des listes d'adjacence peuvent être stockées uniquement dans une liste.	☐	☐
10. Un graphe non orienté d'ordre 5 a au maximum 10 arêtes reliant des sommets distincts.	☐	☐

■ ■ Énoncé des exercices

Exercice 13.1 : Écrire la matrice d'adjacence associée au graphe ci-dessous.



Exercice 13.2 : Écrire les listes de successeurs du graphe suivant :



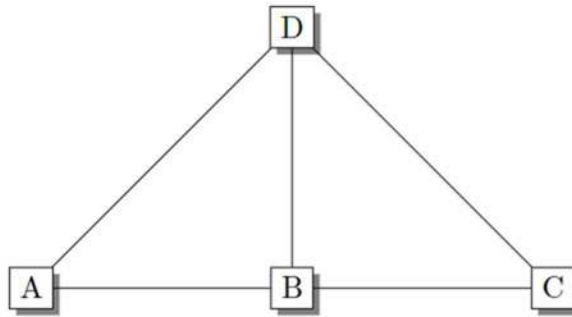
Exercice 13.3 : Tracer des graphes associés aux matrices d'adjacence données.

1. Un graphe non orienté avec $M_1 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$

2. Un graphe orienté avec $M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$

3. Un graphe orienté avec $M_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Exercice 13.4 : On considère le graphe dessiné ci-dessous :



1. Écrire la définition de la liste `m` qui représente la matrice d'adjacence du graphe. Utiliser la représentation du cours.
2. Écrire une fonction qui prend en paramètre une matrice `m` représentant un graphe et renvoie le nombre d'arêtes du graphe.
3. Écrire un dictionnaire en Python, où les clés sont les sommets, pour représenter ce graphe à l'aide des listes d'adjacence.
4. Écrire une fonction `sommets` qui prend en paramètres un sommet `s` et un graphe `g`, sous la forme d'un dictionnaire, et renvoie la liste des sommets liés par une arête au sommet `s`.

Exercice 13.5 : Les graphes sont supposés non orientés et non pondérés. Une variable `g` est définie pour représenter un graphe. Un sommet est représenté par un caractère comme 'A'.

1. Écrire une fonction qui prend en paramètres un graphe `g` représenté par un dictionnaire des listes d'adjacence et un point `s` et ajoute le point au graphe en tant que sommet isolé.
2. Écrire une fonction qui prend en paramètres un graphe `g` représenté par une liste des listes d'adjacence et un point `s` et ajoute le point au graphe en tant que sommet isolé.
3. Écrire une fonction qui prend en paramètres un graphe `g` représenté par une matrice d'adjacence et complète la matrice pour ajouter au graphe un sommet isolé.

Exercice 13.6 : Les conditions sont celles de l'exercice précédent.

1. Écrire une fonction qui prend en paramètres un graphe `g` représenté par un dictionnaire des listes d'adjacence et une arête `a` et ajoute l'arête au graphe. Une arête est représentée par une liste de deux sommets `[s1, s2]`. Les sommets `s1` et `s2` sont supposés appartenir au graphe.
2. Écrire une fonction qui prend en paramètre un graphe `g` représenté par une liste des listes d'adjacence et une arête `a` et ajoute l'arête au graphe. Une arête est définie comme dans la question précédente.
3. Écrire une fonction qui prend en paramètre un graphe `g` représenté par une matrice d'adjacence et une arête `a` et complète la matrice pour ajouter l'arête au graphe. Une arête est représentée par une liste contenant les deux indices des extrémités de l'arête.

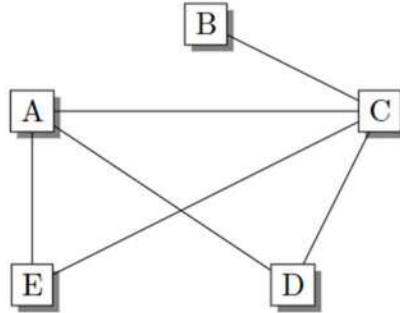
Exercice 13.7 : Les conditions sont celles des deux exercices précédents.

1. Écrire une fonction qui prend en paramètre un graphe `g` représenté par un dictionnaire des listes d'adjacence et renvoie la liste des sommets.
2. Écrire une fonction qui prend en paramètre un graphe `g` représenté par une liste des listes d'adjacence et renvoie la liste des sommets.

Exercice 13.8 : Les conditions sont celles des trois exercices précédents.

Écrire une fonction qui prend en paramètres un graphe g représenté par une matrice d'adjacence et un dictionnaire d et renvoie la liste des arêtes. Le dictionnaire contient des clés qui sont les indices et les valeurs associées qui sont les noms des sommets.

Exercice 13.9* : Cinq personnes sont connectées à un même réseau social. Les liens directs entre deux personnes sont représentées par le graphe G dessiné ci-dessous.



1. Écrire la matrice d'adjacence M associée à G . Définir une liste m qui représente la matrice. Les éléments de m sont des listes dont les éléments sont les coefficients des lignes de M .
2. Écrire une fonction `puissance2` qui prend en paramètre une liste m représentant une matrice carrée M et renvoie une liste représentant la matrice M^2 . Calculer M^2 .

Rappel mathématique : si $M = (a_{i,j})$, avec $1 \leq i \leq n$ et $1 \leq j \leq n$ alors $M^2 = (c_{i,j})$ avec pour tout i et pour tout j , $c_{i,j} = \sum_{k=1}^n a_{i,k} \times a_{k,j}$.

Exercice 13.10* : Conversions

On dispose d'un graphe non orienté sous la forme de listes d'adjacence, par exemple :

$g = \{ "A" : ["B", "D"], "B" : ["A", "C", "D"], "C" : ["B"], "D" : ["A", "B"] \}$

1. La fonction `conversion1` prend en paramètre un tel graphe et renvoie la matrice d'adjacence correspondante. Compléter cette fonction.

La matrice obtenue avec le graphe g est représentée par la liste :

$[[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0]]$.

```

def conversion1(g):
    sommets = {}
    n = 0
    for s in g:
        sommets[s] = n
        n = n + 1
    mat = [n * [0] for i in range(n)]
    ...
    ...
    ...
    return mat
  
```

2. La fonction `conversion2` prend en paramètre une matrice d'adjacence et renvoie le graphe correspondant. On doit donc retrouver le graphe g à partir de la matrice m :

$[[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0]]$.

Compléter la fonction `conversion2`.

```
def conversion2(m):
    sommets = {}
    n = len(m)
    for i in range(n):
        sommets[i] = chr(65 + i)
    g = {}
    for i in range(n):
        g[sommets[i]] = []
        ...
        ...
        ...
    return g
```

■ ■ Indications

Ex. 13.9

Pour la question 2, revoir éventuellement au chapitre 7 la manipulation des listes de listes.

Ex. 13.10

La première partie de la fonction `conversion1` sert à numéroté les sommets. La première partie de la fonction `conversion2` sert à retrouver les nom des sommets à partir de leur numéro.

■ ■ Corrigé du vrai/faux

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

Quelques explications

- 3. La matrice contient n^2 coefficients.
- 5. Chaque arête est comptée deux fois dans les degrés.
- 10. Il y a plusieurs manières de le vérifier, par exemple avec le nombre d'éléments de la matrice d'adjacence qui contient au plus 20 coefficients non nuls.

☐ Erreurs classiques et conseils.—

- Le vocabulaire doit être parfaitement connu.
- Il est important de bien distinguer des graphes orientés ou pas, des graphes pondérés ou pas.
- Un dessin est très souvent utile même quand on réfléchit sur un graphe contenant un grand nombre de sommets.

■ ■ Corrigé des exercices

Exercice 13.1

Les sommets sont ordonnés A, B, C, D.

La matrice d'adjacence est :
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Exercice 13.2

Les listes de successeurs sont les suivantes :

A : B

B : A, D, E

C : B, D

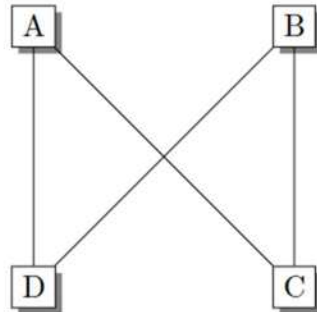
D : C

E : A, D

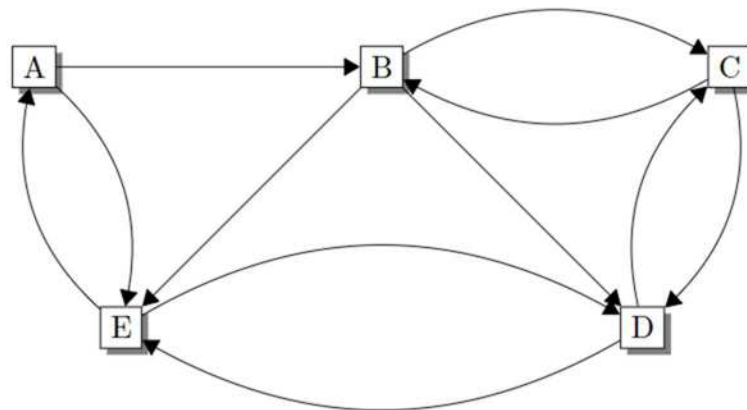
Exercice 13.3

Les noms des sommets n'ont aucune importance dans les représentations.

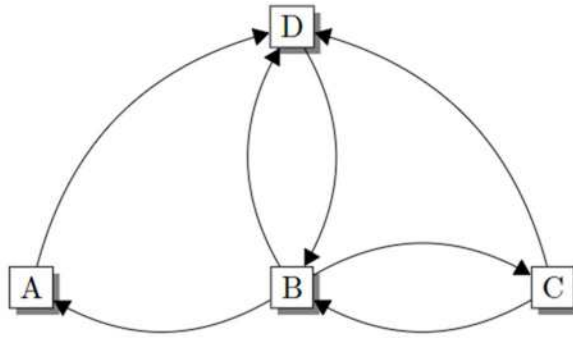
1. Un graphe correspondant à la matrice M_1 .



2. Un exemple de graphe correspondant à la matrice M_2 :



3. Un exemple de graphe correspondant à la matrice M_3 :



Exercice 13.4

1. Définition de la matrice d'adjacence :

`m = [[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 1], [1, 1, 1, 0]]`

2. Une fonction `nb_aretes` :

```

def nb_aretes(m):
    nb = 0
    n = len(m)
    for i in range(n):
        for j in range(i+1, n):
            if m[i][j] == 1:
                nb = nb + 1
    return nb

```

3. Représentation du graphe par les listes d'adjacence avec un dictionnaire :

`graphe = {"A" : ["B", "D"], "B" : ["A", "C", "D"], "C" : ["B", "D"], "D" : ["A", "B", "C"]}`

4. Une fonction `sommets` :

```

def sommets(s, g):
    # on test si s est une clé de g
    if s in g:
        return g[s]

```

On exécute le fichier après y avoir défini le graphe. Puis on écrit `sommets("B", graphe)` dans l'interpréteur, et on obtient l'affichage `['A', 'C', 'D']`.

Exercice 13.5

1. Fonction `ajoute_sommet1` :

```

def ajoute_sommet1(g, s):
    g[s] = []

```

2. Fonction ajoute_sommet2 :

```
def ajoute_sommet2(g, s):  
    g.append([s, []])
```

3. Pour la fonction ajoute_sommet3, on complète chaque ligne avec un 0 et on ajoute une ligne de 0 :

```
def ajoute_sommet3(g):  
    for ligne in g:  
        ligne.append(0)  
    ligne = [0 for i in range(len(g)+1)]  
    g.append(ligne)
```

On peut tester les fonctions sur un exemple :

```
g1 = {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']}  
ajoute_sommet1(g1, 'D')  
  
g2 = [['A', 'B', 'C'], ['B', 'A', 'C'], ['C', 'A', 'B']]  
ajoute_sommet2(g2, 'D')  
  
m = [[0, 1, 1], [1, 0, 1], [1, 1, 0]]  
ajoute_sommet3(m)
```

Exercice 13.6

1. Fonction ajoute_arete1 :

```
def ajoute_arete1(g, a):  
    g[a[0]].append(a[1])  
    g[a[1]].append(a[0])
```

2. Fonction ajoute_arete2 :

```
def ajoute_arete2(g, a):  
    for i in range(len(g)):  
        if g[i][0] == a[0]:  
            g[i][1].append(a[1])  
        if g[i][0] == a[1]:  
            g[i][1].append(a[0])
```


3. Fonction ajoute_arete3 :

```
def ajoute_arete3(g, a):  
    g[a[0]][a[1]] = 1  
    g[a[1]][a[0]] = 1
```

Utilisation des fonctions :

```
g1 = {'A': ['B', 'D'], 'B': ['A', 'C'], 'C': ['B'], 'D': ['A']}  
ajoute_arete1(g1, ['B', 'D'])  
  
g2 = [['A', ['B', 'D']], ['B', ['A', 'C']], ['C', ['B']], ['D', ['A']]]  
ajoute_arete2(g2, ['B', 'D'])  
  
m = [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]  
ajoute_arete3(m, [1, 3])  
ajoute_arete3(m, [1, 3])
```

Exercice 13.7

1. Fonction sommet1 :

```
def sommets1(g):  
    liste = []  
    for s in g:  
        liste.append(s)  
    return liste
```

2. Fonction sommet2 :

```
def sommets2(g):  
    liste = []  
    for adj in g:  
        liste.append(adj[0])  
    return liste
```

Utilisation des fonctions :

```
g1 = {'A': ['B', 'D'], 'B': ['A', 'C'], 'C': ['B'], 'D': ['A']}  
print(sommets1(g1))  
g2 = [['A', ['B', 'D']], ['B', ['A', 'C']], ['C', ['B']], ['D', ['A']]]  
print(sommets2(g2))
```

```
def aretes(g, d):
    n = len(g)
    liste = []
    for i in range(n):
        for j in range(i+1, n):
            if g[i][j] == 1:
                liste.append([d[i], d[j]])
    return liste

m = [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]
d = {0: 'A', 1: 'B', 2: 'C', 3: 'D'}
print(aretes(m, d))
```

1. On écrit la matrice d'adjacence M :

$$M = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

On définit $m = [[0, 0, 1, 1, 1], [0, 0, 1, 0, 0], [1, 1, 0, 1, 1], [1, 0, 1, 0, 0], [1, 0, 1, 0, 0]]$.

2. Fonction puissance2 :

```
def puissance2(m):
    n = len(m)
    c = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            p = 0
            for k in range(n):
                p = p + m[i][k] * m[k][j]
            c[i][j] = p
    return c

m = [[0, 0, 1, 1, 1], [0, 0, 1, 0, 0], [1, 1, 0, 1, 1],
      [1, 0, 1, 0, 0], [1, 0, 1, 0, 0]]
print(puissance2(m))
```

Remarque ; les coefficients de la matrice M^2 sont liés au nombre de chemins de longueur 2 reliant deux sommets.

Exercise 13.10

1. Suite du code de la fonction `conversion1` :

```
def conversion1(g):
    ...

    mat = [n * [0] for i in range(n)]
    for s in sommets:
        for voisin in g[s]:
            mat[sommets[s]][sommets[voisin]] = 1
    return mat
```

Test :

```
>>> g = {"A" : ["B", "D"], "B" : ["A", "C", "D"], "C" : ["B"],
        "D" : ["A", "B"]}
>>> conversion1(g)
[[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0]]
```

2. La fonction `conversion2` :

```
def conversion2(m):
    sommets = {}
    n = len(m)
    for i in range(n):
        sommets[i] = chr(65 + i)
    g = {}
    for i in range(n):
        g[sommets[i]] = []
        for j in range(len(m[i])):
            if m[i][j] == 1:
                g[sommets[i]].append(sommets[j])
    return g
```

Test :

```
>>> m = [[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0]]
>>> conversion2(m)
{'A': ['B', 'D'], 'B': ['A', 'C', 'D'], 'C': ['B'], 'D': ['A', 'B']}
```


Chapitre 14

Algorithmes

UN SCIENTIFIQUE



Au début du IX^e siècle à Bagdad, le Calife abbasside Al-Mamoun, attire les plus grands savants du monde arabo-musulman. C'est là que le mathématicien persan **Mohammed ibn Mousa Al-Khwarizmi** (788-850) rédige son ouvrage *Kitab al jabr wa muqabala* (le livre de la remise en place et de la comparaison) qui sera diffusé dans les universités de l'Occident médiéval. Un autre livre, connu seulement dans sa traduction latine, explique le maniement des chiffres dans la numération de position venue d'Inde.

■ Un peu d'histoire

Les algorithmes sont présents dès la haute Antiquité. On en rencontre déjà chez les Babyloniens pour résoudre des équations simples et les Égyptiens en élaborent pour effectuer des additions, des multiplications et même des divisions. Dans l'Antiquité grecque, les nombres sont l'objet de différents algorithmes comme celui proposé par Héron d'Alexandrie qui permet d'obtenir une valeur approchée de la racine carrée d'un nombre.

Au début du IX^e siècle de notre ère, al-Khwarizmi établissait une méthode permettant de trouver les racines de toute équation du second degré. Il distinguait six cas différents car il refusait d'envisager des nombres négatifs. Ses ouvrages ont été traduits en latin et son nom fut latinisé en *Algoritmus* d'où est issu le mot *algorithme*. Pour lui donner une apparence savante, un h a été ajouté faisant de ce mot une anagramme de *logarithme*.

Les graphes servent dans de nombreux domaines à modéliser des situations. Des algorithmes performants ont été développés pour répondre à différentes questions.

OBJECTIFS

- Acquérir des notions sur les principaux algorithmes utilisés avec les graphes :
 - ▶ assimiler les principes de parcours d'un graphe ;
 - ▶ concevoir une structure adaptée à un parcours, une pile ou une file ;
 - ▶ savoir utiliser un parcours pour trouver un chemin, repérer un cycle, déterminer la connexité.
- Appliquer une méthode de recherche d'un plus court chemin entre deux sommets :
 - ▶ traduire des situations avec le vocabulaire des graphes ;
 - ▶ connaître et savoir implémenter l'algorithme de Dijkstra ;
 - ▶ comprendre la notion d'heuristique utilisée par l'algorithme A*.

Considérons un graphe dont les sommets représentent des villes à visiter et les arêtes les routes reliant ces villes. Une stratégie de visite est appliquée. On dispose d'une liste d'attente dans laquelle on inscrit les villes en attente de visite. On commence par y placer la ville de départ. Ensuite, tant qu'il reste des villes dans la liste d'attente, on en choisit une et on la retire de la liste. On pense avant d'oublier à inscrire toutes les villes voisines non visitées dans la liste d'attente, et on visite la ville qui a été choisie si cela n'a pas déjà été fait. Les villes visitées sont inscrites dans une liste afin de garder une trace du parcours effectué.

À chaque itération, une ville est visitée et ceci assure la terminaison de l'algorithme.

C'est l'étape « choisir une ville dans la liste d'attente » qui définit l'ordre de parcours.

Si on applique le principe d'une file d'attente à un guichet, c'est la première inscrite qui est choisie. Si on applique le principe d'une pile d'assiettes manipulées une à une, c'est la dernière inscrite qui est choisie. « On retire une assiette sur la pile ». On peut ajouter un critère qui donne une valeur à chaque ville et on applique alors le principe d'une file de priorité : on choisit la ville qui a la plus grande valeur.

■ Piles et files

Une structure linéaire sur un ensemble E est une suite d'éléments de E . Chaque élément a une place bien précise dans cette suite qui est ordonnée.

En Python, une structure de pile, « stack » en anglais, et une structure de file, « queue » en anglais, sont des structures linéaires qui peuvent être construites à l'aide des listes. Il suffit d'appliquer des conditions d'utilisation restreintes. Pour une file, on n'autorise que l'insertion d'un élément d'un côté et la suppression de l'autre, en suivant le principe « premier entré, premier sorti », en anglais FIFO, acronyme de « First In First Out ». Pour une pile, les seules opérations autorisées sont l'insertion et la suppression d'un élément uniquement à la fin de la liste. C'est le principe « dernier entré, premier sorti », en anglais LIFO, acronyme de « Last In First Out ».

Ces structures sont très présentes en informatique. Une pile peut servir à mémoriser de l'information, elle est utilisée dans la programmation récursive. Un navigateur ou un traitement de texte mémorise les actions effectuées afin de pouvoir revenir en arrière pas à pas. Les pages ou les fichiers ouverts sont également mémorisés. Un système d'exploitation gère les processus à l'aide d'une file. Une imprimante gère une file de documents en attente d'impression.

Trois fonctions traduisent les opérations de base sur les piles :

- une fonction qui crée une pile vide ;
- une fonction qui ajoute un élément sur la pile (en anglais « push ») ;
- une fonction qui retire l'élément sur la pile, le sommet, et le renvoie (en anglais « pop »).

D'autres fonctions peuvent fournir des informations sur une pile. Les plus utilisées sont :

- une fonction qui permet de savoir si une pile est vide ;
- une fonction qui donne la taille de la pile ;
- une fonction qui donne le sommet de la pile.

Nous représentons une pile par une liste en Python. À priori, la taille n'est pas limitée et nous disons que la pile a une *capacité non bornée*.

Voici un code pour les fonctions `creer_pile`, `empiler` et `depiler` :

```
def creer_pile():
    """Crée une pile vide"""
    return []

def empiler(p, x):
    """Ajoute un élément x sur la pile p"""
    p.append(x)

def depiler(p):
    """Renvoie le sommet de la pile p si elle est non vide,
    le sommet est retiré de la pile"""
    if len(p) > 0:
        return p.pop()
```

Remarque : la fonction `empiler` ne renvoie rien.

Nous pouvons aussi coder quelques fonctions auxiliaires :

```
def pile_vide(p):
    """Renvoie un booléen
    True si la pile est vide et False sinon"""
    return p == []

def taille(p):
    """Renvoie la taille de la pile p"""
    return len(p)

def sommet(p):
    """Renvoie le sommet de la pile p,
    le sommet n'est pas retiré de la pile"""
    if len(p) > 0:
        return p[len(p)-1]
```

Utilisation des fonctions :

```
>>> p = creer_pile():
>>> empiler(p, "A")
>>> empiler(p, "B")
>>> empiler(p, "C")
>>> taille(p)
```

```

3
>>> sommet(p)
'C'
>>> depiler(p)
'C'

```

À aucun moment un utilisateur n'a besoin de savoir comment sont codées les fonctions. Il lui suffit simplement de suivre les spécifications.

En pratique une pile n'a pas une capacité illimitée. Il peut y avoir un débordement lorsque la capacité maximale est dépassée. Plusieurs manières sont envisageables pour représenter une pile à capacité bornée par une liste, par exemple en limitant la longueur de la liste.

Une file peut aussi être implémentée à l'aide d'une liste en Python. Les opérations de bases sont semblables à celles codées pour les piles. La principale différence concerne le retrait d'un élément. La méthode `pop` utilisée pour retirer le sommet de la pile le fait en temps constant. Sa complexité est en $\mathcal{O}(1)$. (C'est en fait une « complexité amortie », il y a des cas où ce n'est pas exact). Par contre pour retirer le premier élément d'une liste, on utilise la méthode `pop` en précisant l'indice de l'élément, soit `pop(0)`. Or le retrait du premier élément ne se fait pas en temps constant. La complexité est en $\mathcal{O}(n)$ si n est la taille de la liste.

Une méthode est d'utiliser une liste avec deux variables dont les valeurs sont les indices de début et de fin de la file.

Il est aussi possible d'implémenter une file à l'aide de deux piles (et réciproquement une pile à l'aide de deux files). Cette méthode est proposée en exercice.

La bibliothèque standard en Python propose une implémentation des files et des piles dans le module `queue`. L'utilisation est simple.

```

from queue import *

q = Queue(5) # une file
q.put(3) # pour placer dans la file
q.put(6)
q.put(12)
a = q.get() # pour retirer de la file
print(a)
print(q.empty()) # file vide
print(q.qsize()) # taille de la file

p = LifoQueue(5) # une pile
p.put(3) # pour placer dans la pile
p.put(6)
p.put(12)
a = p.get() # pour retirer de la pile
print(a)
print(p.empty()) # pile vide
print(p.qsize()) # taille de la pile

```

Il existe aussi dans le module `collections` la structure `deque` (pour *double-ended queue*). Un objet de type `deque` a un comportement similaire à celui d'une liste mais les ajouts et les retraits sont rapides à chaque extrémité. On peut donc l'utiliser pour une file ou une pile.

```
from collections import deque

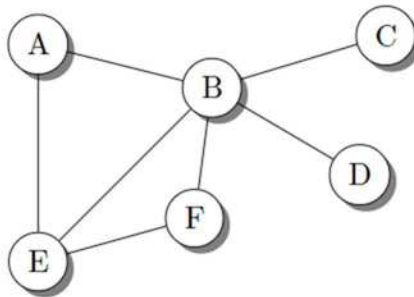
file = deque([1, 2, 3])
file.append(4)
print(file)
file.popleft()
print(file)

pile = deque([1, 2, 3])
pile.append(4)
print(pile)
pile.pop()
print(pile)
```

■ Parcours d'un graphe

■ Introduction

Pour tester différents programmes, nous utilisons le graphe représenté par le dessin ci-dessous.



Pour définir ce graphe, nous utilisons un dictionnaire des listes d'adjacence.

```
g = {'A': ['B', 'E'],
     'B': ['A', 'C', 'D', 'E', 'F'],
     'C': ['B'],
     'D': ['B'],
     'E': ['A', 'B', 'F'],
     'F': ['B', 'E']}
```

De nombreux algorithmes ont été étudiés pour répondre à des questions sur les graphes, en particulier des questions d'optimisation. Ces algorithmes sont utilisés pour résoudre des problèmes

issus de domaines divers. De manière générale, il s'agit de trouver un ensemble de sommets ou de chemins vérifiant une propriété.

Pour traiter les différentes questions, nous disposons de deux types de méthodes, deux manières de parcourir un graphe pour effectuer une recherche.

- Parcours d'un graphe en profondeur d'abord : à partir d'un sommet, on passe à un de ses voisins, puis à un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins.
- Parcours d'un graphe en largeur d'abord : à partir d'un sommet, on explore tous ses voisins immédiats. Puis à partir d'un voisin, on explore tous ses voisins immédiats sauf ceux déjà explorés. Et ainsi de suite.

Complexité

De manière générale, si n est le nombre de sommets et m le nombre d'arêtes, on montre que la complexité en temps dans le pire des cas d'un parcours en profondeur ou d'un parcours en largeur est de l'ordre de $(n + m)$.

■ Parcours d'un graphe

■ Parcours en profondeur d'abord

► Programme récursif

Une méthode récursive est la plus adaptée. On utilise une liste `visite` pour stocker les éléments visités et une liste `attente` pour stocker les voisins d'un sommet non encore visités. On ajoute un dictionnaire `marque` pour marquer les sommets visités. C'est le plus efficace. Sinon il faudrait pour chaque sommet parcourir entièrement la liste `visite` afin de vérifier s'il a déjà été visité ou pas, ce qui augmente la complexité temporelle.

La fonction `parcours_dfs` (dfs pour depth first search) prend en paramètre un graphe, le sommet initial où commence la parcours, une liste et un dictionnaire vides.

```
def parcours_dfs(graphe, sommet, visite, marque):
    if sommet not in marque:
        visite.append(sommet)
        marque[sommet] = True
    attente = [s for s in graphe[sommet] if s not in marque]
    for s in attente:
        parcours_dfs(graphe, s, visite, marque)
    return visite
```

Avec le graphe défini en exemple, on obtient dans l'interpréteur :

```
>>> parcours_dfs(g, "A", [], {})
['A', 'B', 'C', 'D', 'E', 'F']
```

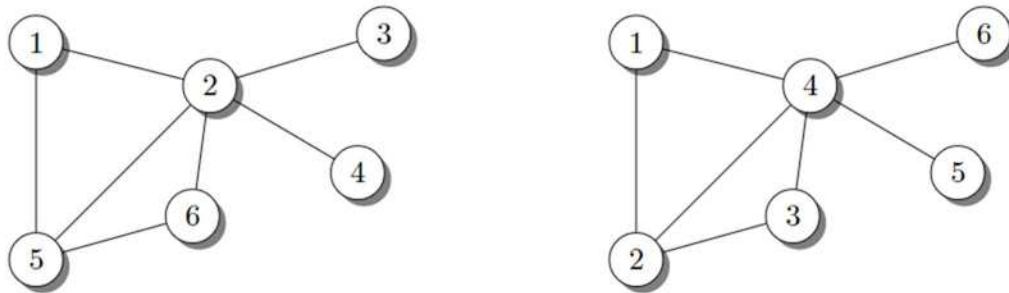
► Programme itératif

On utilise une pile pour placer les sommets en attente. Pour les piles, on utilise le module `collections`.

```
def iteratif_dfs(graphe, sommet):
    visite = []
    marque = {}
    attente = deque() # une pile
    attente.append(sommet)
    while len(attente) > 0:
        sommet = attente.pop()
        if sommet not in marque:
            visite.append(sommet)
            marque[sommet] = True
            for s in graphe[sommet]:
                if s not in marque:
                    attente.append(s)
    return visite
```

Les programmes récursif et l'itératif ne donnent pas le même résultat avec le graphe exemple. Mais il s'agit bien dans les deux cas d'un parcours en profondeur d'abord : à chaque étape, on s'éloigne du sommet initial (en nombre d'arêtes) d'une unité si c'est possible.

Les numéros indiquent l'ordre de parcours dans les deux cas.



Remarque :

Avec l'une des deux fonctions de parcours et un sommet S initial, on obtient tous les sommets que l'on peut joindre à partir de S. Si on obtient tous les sommets d'un graphe non orienté, cela signifie que le graphe est connexe. Sinon l'ensemble des sommets que l'on peut joindre est la classe de connexité de S.

► Recherche d'un plus court chemin

Un parcours permet de déterminer un chemin entre le sommet initial et un sommet quelconque du graphe (connexe non orienté). Il suffit de mémoriser le prédécesseur de chaque sommet placé en attente. Lorsqu'on atteint le sommet final, on obtient le chemin en remontant les prédécesseurs un à un jusqu'au chemin initial.

Un autre problème est la recherche du plus court chemin entre deux sommets. On suppose le graphe non orienté, non pondéré. Il s'agit de trouver un chemin allant d'un sommet initial à un sommet final avec un nombre d'arêtes minimum.

Cette recherche peut s'effectuer avec un parcours en profondeur d'abord. On utilise une liste pour mémoriser un chemin en cours et une autre liste pour mémoriser le plus court des chemins étudiés.

```
def pcc_dfs(graphe, debut, fin, chemin = [], pluscourt = None):
    chemin = chemin + [debut]
    if debut == fin:
        return chemin
    for s in graphe[debut]:
        if s not in chemin:
            if pluscourt == None or len(chemin) < len(pluscourt):
                nouveau = pcc_dfs(graphe, s, fin, chemin, pluscourt)
                if nouveau != None:
                    pluscourt = nouveau
    return pluscourt

print(pcc_dfs(g, 'A', 'F'))
```

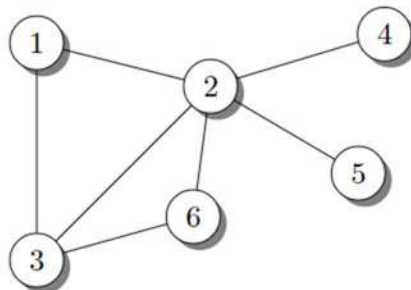
■ Parcours en largeur d'abord

Il suffit de prendre le programme itératif de parcours en profondeur et de remplacer la pile par une file pour placer les sommets en attente. On parle de BFS pour Breadth First Search.

```
from collections import deque

def iteratif_bfs(graphe, sommet):
    visite = []
    marque = {}
    attente = deque()
    attente.append(sommet)
    while len(attente) > 0:
        sommet = attente.popleft()
        if sommet not in marque:
            visite.append(sommet)
            marque[sommet] = True
            for s in graphe[sommet]:
                if s not in marque:
                    attente.append(s)
    return visite
```

On obtient avec le graphe exemple le parcours ['A', 'B', 'E', 'C', 'D', 'F'].



Comme avec un parcours en profondeur, on visite tous les sommets d'un graphe connexe et on peut obtenir un chemin reliant deux sommets quelconque.

► Recherche d'un cycle

Un parcours en largeur d'abord peut aussi permettre de repérer la présence d'un cycle, c'est-à-dire un chemin, où toutes les arêtes sont différentes, qui se termine à son sommet de début. Le graphe est supposé non orienté.

Pour cela, on associe un niveau à tous les sommets, `None` pour initialiser. On commence le parcours en un sommet auquel on donne le niveau 0. Puis on visite ses voisins, auxquels on donne le niveau 1 et ainsi de suite.

Cette méthode utilise une file. Cette file est vide et on y place le sommet de départ choisi dont le niveau est 0. La suite de l'algorithme est décrite ci-dessous :

Tant que la file n'est pas vide :

- on retire l'élément en tête de file, soit `S` ;
- on visite ses voisins `s` et pour chaque voisin :
 - si son niveau est `None`, on lui donne le niveau de `S` augmenté de 1 et on le place à la suite dans la file ;
 - si son niveau est supérieur ou égal à celui de `S`, on a trouvé un cycle et on renvoie `True` ;
 - si son niveau est strictement inférieur à celui de `S`, on ne fait rien.

Si la file est vide, on renvoie `False`, le graphe ne contient aucun cycle.

```
def cycle_bfs(graphe, sommet):
    niveaux = {s : None for s in graphe}
    niveaux[sommet] = 0
    file = deque()
    file.append(sommet)
    while len(file) > 0:
        sommet = file.popleft()
        for s in graphe[sommet]:
            if niveaux[s] == None:
                niveaux[s] = niveaux[sommet] + 1
                file.append(s)
            elif niveaux[s] >= niveaux[sommet]:
                return True
    return False
```

La fonction exécutée avec le graphe exemple renvoie `True` quel que soit le sommet initial. Il est intéressant de tester ce programme avec un graphe qui ne contient aucun cycle.

Un parcours en largeur d'abord peut être utilisé aussi pour la recherche du plus court chemin entre deux sommets.

Cette recherche d'un chemin le plus court entre deux sommets d'un graphe est un problème courant qui a été longuement étudié. Les algorithmes présentés dans la suite sont classiques.

■ Algorithme de Dijkstra et algorithme A*

Le graphe est pondéré avec des poids positifs. Ceci signifie qu'on ne compte plus le nombre d'arêtes composant un chemin mais le *coût* de ce chemin, c'est-à-dire la somme des poids des arêtes.

Remarque 1 : compter simplement le nombre d'arêtes comme dans les parties précédentes, revient à affecter un coût identique à chaque arête.

Remarque 2 : le problème « trouver le plus court chemin entre un sommet initial I et un sommet final F » semble à priori plus simple que le problème « trouver le plus court chemin entre un sommet initial I et chacun des autres sommets du graphe ». Pourtant, dans certains cas, on résout le deuxième problème pour résoudre le premier problème.

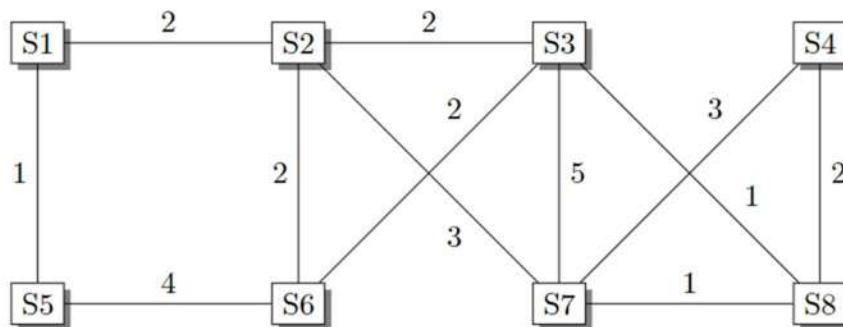
■ Algorithme de Dijkstra

Edsger Dijkstra a publié son algorithme en 1959. Il utilise un parcours en largeur et calcule le plus court chemin entre un sommet et chacun des autres sommets. On suppose que le graphe est connexe et non orienté.

Principe : si le plus court chemin entre deux sommets D et A passe par un sommet I , alors la partie de ce chemin entre D et I est le plus court chemin de D à I et la partie entre I et A est le plus court chemin entre I et A . À chaque étape, on effectue donc le meilleur choix possible. C'est un algorithme glouton. L'algorithme est semblable à celui d'un parcours en largeur d'abord, mais au lieu d'utiliser une file pour les sommets en attente, on utilise une file de priorité. Cela signifie qu'on extrait le sommet ayant la priorité, dans ce cas c'est celui qui correspond à la distance minimale.

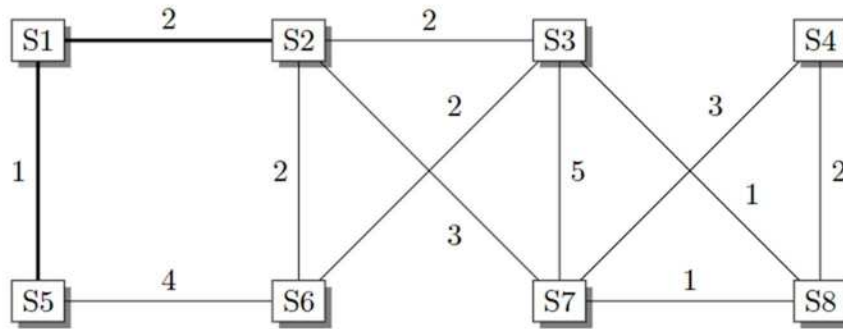
Sur l'exemple qui suit le sommet de départ est S1 et on cherche le plus court chemin entre ce sommet et chacun des autres sommets du graphe. On commence par affecter une valeur très grande à chacun des sommets, disons une valeur infinie, et la valeur 0 au sommet S1. Ces valeurs représentent les longueurs des chemins entre S1 et chacun des sommets.

- Pris : (S1, 0).
- Non pris : (S2, ∞), (S3, ∞), (S4, ∞), (S5, ∞), (S6, ∞), (S7, ∞), (S8, ∞).



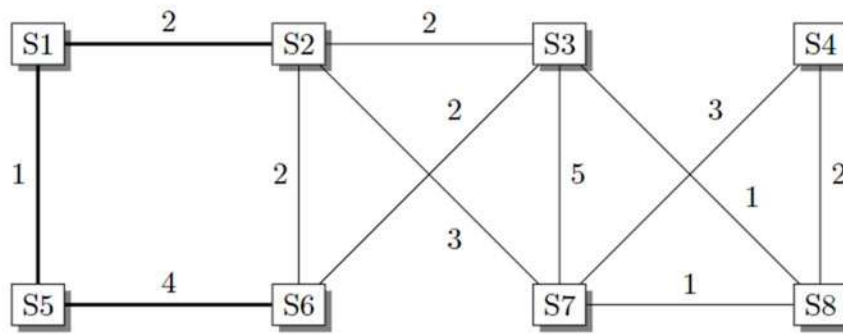
À partir du minimum, donc S1, on peut aller en S2 ou S5 avec les distances respectives 2 et 1. On prend celui correspondant à la valeur minimale S5.

- Pris : (S1, 0), (S5, 1).
- Non pris : (S2, 2), (S3, ∞), (S4, ∞), (S6, ∞), (S7, ∞), (S8, ∞).



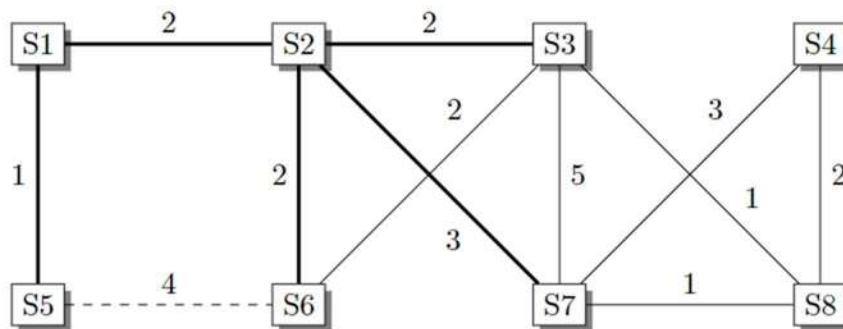
À partir de S5, on peut aller en S6 à la distance 4, donc une distance totale de 5 depuis S1. On prend le sommet non pris correspondant à la valeur minimale, soit S2.

- Pris : (S1, 0), (S5, 1), (S2, 2).
- Non pris : (S3, ∞), (S4, ∞), (S6, 5), (S7, ∞), (S8, ∞).



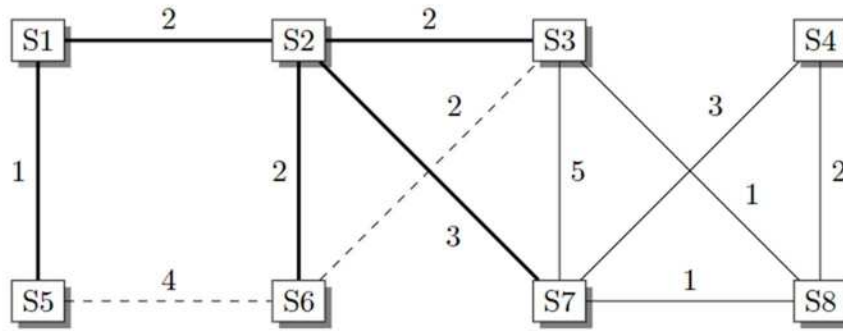
À partir de S2, on peut aller en S3, S6, S7 avec les distances respectives 2, 2, 3, donc des distances totales respectives de 4, 4, 5 depuis S1. On prend le sommet non pris correspondant à la valeur minimale, soit S6. (On pourrait aussi garder S3).

- Pris : (S1, 0), (S5, 1), (S2, 2), (S6, 4).
- Non pris : (S3, 4), (S4, ∞), (S7, 5), (S8, ∞).



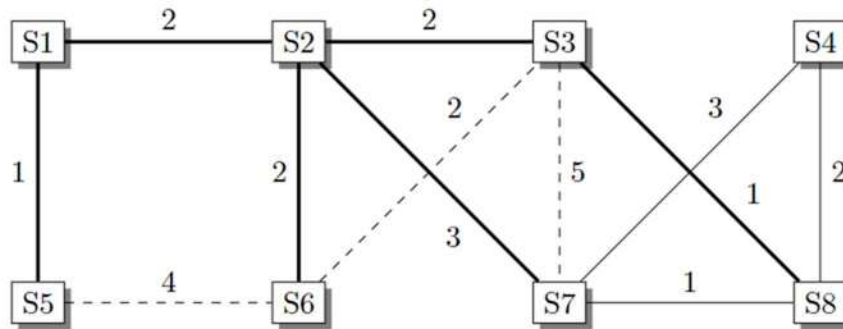
À partir de S6, on peut aller en S3 avec une distance totale de 6 depuis S1. Donc ce n'est pas intéressant. On prend le sommet non pris correspondant à la valeur minimale, soit S3.

- Pris : (S1, 0), (S5, 1), (S2, 2), (S6, 4), (S3, 4).
- Non pris : (S4, ∞), (S7, 5), (S8, ∞).



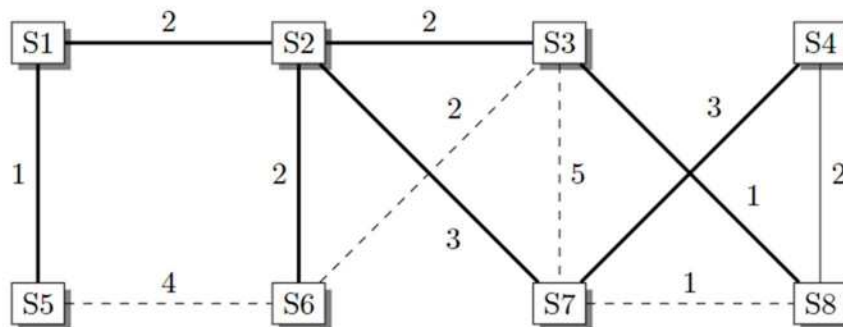
À partir de S3, on peut aller en S7 et S8 avec les distances respectives 5 et 1, donc des distances totales respectives de 9 et 5 depuis S1, ce qui n'améliore pas la distance totale pour S7. On prend le sommet non pris correspondant à la valeur minimale, soit S7. (On pourrait garder S8).

- Pris : (S1, 0), (S5, 1), (S2, 2), (S6, 4), (S3, 4), (S7, 5).
- Non pris : (S4, ∞), (S8, 5).



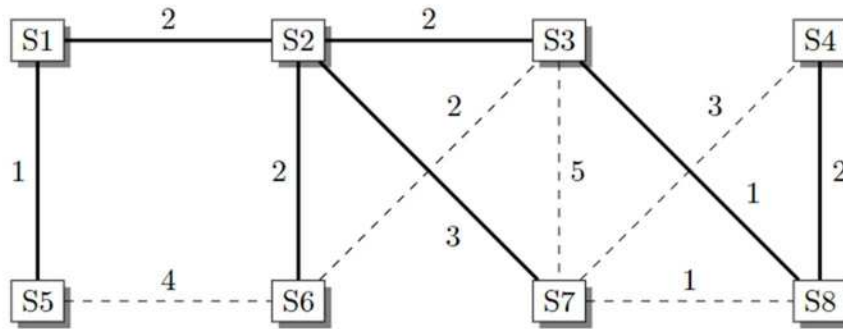
À partir de S7, on peut aller en S4 et S8 avec les distances respectives 3 et 1, donc des distances totales respectives de 8 et 6 depuis S1, ce qui n'améliore pas la distance totale pour S8. On prend le sommet non pris correspondant à la valeur minimale, soit S8.

- Pris : (S1, 0), (S5, 1), (S2, 2), (S6, 4), (S3, 4), (S7, 5), (S8, 5).
- Non pris : (S4, 8).



À partir de S8, on peut aller en S4 avec la distance 2, donc une distance totale de 7 depuis S1. On prend le sommet non pris correspondant à la valeur minimale, soit S4.

- Pris : (S1, 0), (S5, 1), (S2, 2), (S6, 4), (S3, 4), (S7, 5), (S8, 5), (S4, 7).
- Non pris :



Complexité de l'algorithme

L'étude de la complexité n'est pas simple. On peut quand même dire qu'il y a une étape par sommet, soit n étapes. Pour chaque étape, on calcule des distances avec les autres sommets, au maximum n opérations et on cherche le minimum parmi n sommets au maximum. La complexité est donc dans le pire des cas de l'ordre de n^2 .

Les résultats successifs sont rassemblés dans le tableau qui contient les valeurs des distances évaluées pour chaque sommet au fil des étapes.

Étape	S1	S2	S3	S4	S5	S6	S7	S8
0	0	∞	∞	∞	∞	∞	∞	∞
1		2	∞	∞	1	∞	∞	∞
2		2	∞	∞		5	∞	∞
3			4	∞		4	5	∞
4			4	∞			5	∞
5				∞			5	5
6				8				5
7				7				

Écriture d'un programme

Le graphe donné en exemple est définie par les listes d'adjacence avec les poids des arêtes.

```

graphe = {"S1" : [("S2", 2), ("S5", 1)],
          "S2" : [("S1", 2), ("S3", 2), ("S6", 2), ("S7", 3)],
          "S3" : [("S2", 2), ("S6", 2), ("S7", 5), ("S8", 1)],
          "S4" : [("S7", 3), ("S8", 2)],
          "S5" : [("S1", 1), ("S6", 4)],
          "S6" : [("S2", 2), ("S5", 4), ("S3", 2)],
          "S7" : [("S2", 3), ("S3", 5), ("S4", 3), ("S8", 1)],
          "S8" : [("S3", 1), ("S4", 2), ("S7", 1)]}
```

Nous pouvons utiliser une file de priorité `PriorityQueue` implémentée dans le module `queue` (voir exercices). À la place, nous utilisons un dictionnaire dont les clés sont les sommets et les valeurs sont les distances respectives entre chaque sommet et le sommet de départ, comme dans le tableau ci-dessus.

Nous avons donc besoin d'une fonction qui calcule le minimum des valeurs contenues dans un dictionnaire et renvoie la clé correspondante. On utilise la valeur `inf` (infini) de type `float`.

```
def minimum(dico):
    """dico contient au moins une valeur différente de inf"""
    mini = float('inf')
    for k in dico:
        if dico[k] < mini:
            mini = dico[k]
            s = k
    return s
```

Pour la fonction `dijkstra`, nous avons besoin d'un tableau `D` contenant les sommets choisis avec les longueurs de chemins minimales et d'un tableau `d` contenant les sommets avec les longueurs de chemins initiales infinies sauf pour le sommet de départ.

```
def dijkstra(G, s):
    D = {} # tableau final des distances minimales
    d = {k: float('inf') for k in G} # tableau des distances initiales
    d[s] = 0 # s sommet de départ
    while len(d) > 0: # c'est fini quand d est vide
        k = minimum(d) # sommet de distance minimale pour démarrer une étape
        for j in range(len(G[k])): # on regarde les voisins de k
            v, c = G[k][j] # v un voisin de k, c la distance
            if v not in D:
                d[v] = min(d[v], d[k] + c)
        D[k] = d[k] # on copie le sommet et la distance dans D
        del d[k] # on supprime le sommet de d
    return D
```

Application :

```
>>> dijkstra(graphe, "S1")
{'S1': 0, 'S5': 1, 'S2': 2, 'S3': 4, 'S6': 4, 'S7': 5, 'S8': 5, 'S4': 7}
```

■ Exemples d'utilisation

Un réseau comme Internet ou un réseau privé peut être représenté par un graphe. Les liaisons entre deux machines sont les arêtes du graphe.

Divers protocoles existent pour transmettre des données entre des machines appartenant à un réseau. L'un de ces protocoles utilise l'algorithme de Dijkstra, un autre l'algorithme de Bellman-Ford. Avec l'algorithme de Dijkstra les sommets sont visités au plus une fois, alors qu'avec celui de Bellman et Ford ils sont visités plus d'une fois.

De manière générale, on peut utiliser les deux algorithmes tant que les poids des arêtes ne sont pas négatifs. Si c'est le cas, l'algorithme de Dijkstra ne fonctionne pas. En revanche, celui de Bellman et Ford fonctionne correctement et permet même de détecter la présence d'un cycle de poids négatif.

■ Algorithme A*

Divers algorithmes de recherche d'un chemin dans un graphe existent. Peter E. Hart, Nils John Nilsson et Bertram Raphael en ont proposé un en 1968, nommé *algorithme A**. Cet algorithme qui fournit un chemin entre deux sommets donnés est une extension de l'algorithme de Dijkstra. La solution obtenue est l'une des meilleures et elle est obtenue « rapidement ». Il est utilisé en intelligence artificielle et dans des applications comme les jeux vidéos pour lesquels le plus important est la vitesse d'obtention d'une solution même si elle n'est pas optimale.

L'algorithme utilise une évaluation heuristique sur chaque sommet afin de parvenir à trouver ce meilleur chemin. Les sommets sont visités suivant l'ordre donné par cette évaluation. L'algorithme utilise une file d'attente prioritaire.

Une méthode heuristique est une méthode de résolution utilisée pour obtenir une solution rapidement, pas forcément la meilleure, quand d'autres algorithmes ont une complexité en temps trop élevée. On n'explore pas toutes les possibilités pour trouver la solution optimale, mais on les filtre à l'aide de données supplémentaires provenant de mesures, d'expériences, de statistiques.

Pour une première version, on modifie la fonction `minimum` utilisée avec l'algorithme de Dijkstra :

```
def minim(dico):
    """dico contient au moins une valeur différente de inf"""
    mini = float('inf')
    for k in dico:
        d = dico[k][0] + dico[k][1]
        if d < mini:
            mini = d
            s = k
    return s

def algo_star(G, deb, fin, h): # h est la fonction heuristique
    D = {} # tableau final des distances minimales
    d = {k: [float('inf'), h(k)] for k in G} # tableau des distances initiales
    d[deb] = [0, h(deb)] # deb sommet de départ
    while fin in d: # on a fini quand fin a été examiné
        k = minim(d) # sommet de distance minimale pour démarrer une étape
        for j in range(len(G[k])): # on regarde les voisins de k
            v, c = G[k][j] # v un voisin de k, c le coût
            if v not in D:
                d[v][0] = min(d[v][0], d[k][0] + c)
        D[k] = d[k] # on copie la distance dans d
        del d[k] # on supprime le sommet de d
    return D
```

```
# Exemple pour un test
```

```
g = {"S" : [("A", 7), ("B", 2), ("C", 3)],
      "A" : [("S", 7), ("B", 3), ("D", 4)],
```

```

"B" : [("S", 2), ("A", 3), ("D", 4), ("H", 1)],
"C" : [("S", 3), ("L", 2)],
"D" : [("A", 4), ("B", 4), ("F", 5)],
"E" : [("G", 2), ("K", 5)],
"F" : [("D", 5), ("H", 3)],
"G" : [("H", 2), ("E", 2)],
"H" : [("B", 1), ("F", 3), ("G", 2)],
"I" : [("J", 6), ("K", 4), ("L", 4)],
"J" : [("I", 6), ("K", 4), ("L", 4)],
"K" : [("I", 4), ("J", 4), ("E", 5)],
"L" : [("C", 2), ("I", 4), ("J", 4)]]}

# les valeurs heuristiques sont par exemple des distances "à vol d'oiseau":
valeurs_heuristiques = {"A": 9, "B": 7, "C": 8, "D": 8, "E": 0,
                        "F": 6, "G": 3, "H": 6, "I": 4, "J": 4, "K": 3, "L": 6, "S": 10}

def heuristique(s):
    return valeurs_heuristiques[s]

print(algo_star(g, "S", "E", heuristique))

```

Dans cette deuxième version on utilise la même fonction minimum que pour l'algorithme de Dijkstra mais avec deux dictionnaires d et dh pour les distances réelles et heuristiques :

```

def algo_star2(G, deb, fin, h): # h est la fonction heuristique
    D = {} # tableau final des distances minimales
    d = {k: float('inf') for k in G} # tableau des distances initiales
    d[deb] = 0 # deb sommet de départ
    dh = {k: d[k]+h(k) for k in G} # distances tenant compte de l'heuristique
    while fin in d: # on a fini quand fin a été examiné
        k = minimum(dh) # sommet de distance minimale pour démarrer une étape
        for j in range(len(G[k])): # on regarde les voisins de k
            v, c = G[k][j] # v un voisin de k, c le coût
            if v not in D:
                d[v] = min (d[v], d[k] + c)
                dh[v] = d[v] + h(v)
        D[k] = d[k] # on copie le sommet et la distance dans d
        del d[k]
        del dh[k]
    return D

print(algo_star2(g, "S", "E", heuristique))

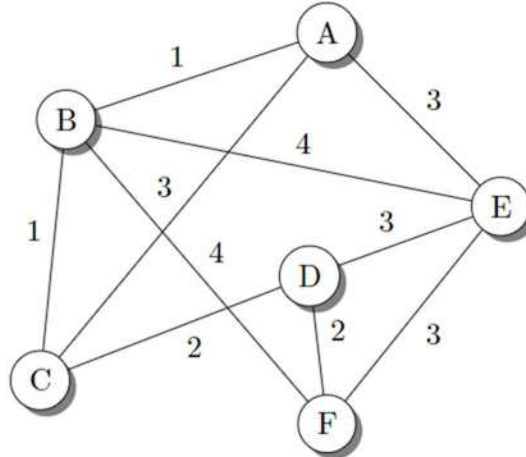
```

■ ■ Vrai/Faux

	Vrai	Faux
1. L'acronyme LILO est utilisé pour parler d'une pile.	☐	☐
2. Pour accéder au premier élément placé dans une file, il faut retirer tous les éléments qui ont été placés après.	☐	☐
3. On peut utiliser une file pour programmer avec un graphe un parcours en largeur d'abord.	☐	☐
4. Dans un parcours en profondeur, on commence par visiter tous les sommets adjacents au sommet de départ.	☐	☐
5. On peut détecter la présence d'un cycle dans un graphe non orienté avec un parcours en largeur.	☐	☐
6. On ne peut pas détecter la connexité d'un graphe non orienté avec un parcours en profondeur.	☐	☐
7. La complexité d'un parcours en profondeur dans un graphe d'ordre n est linéaire en n .	☐	☐
8. L'algorithme de Dijkstra a une complexité linéaire en fonction du nombre de sommets.	☐	☐
9. L'algorithme de Dijkstra est un algorithme glouton.	☐	☐
10. L'algorithme A* garantit de toujours obtenir le plus court chemin entre deux sommets.	☐	☐

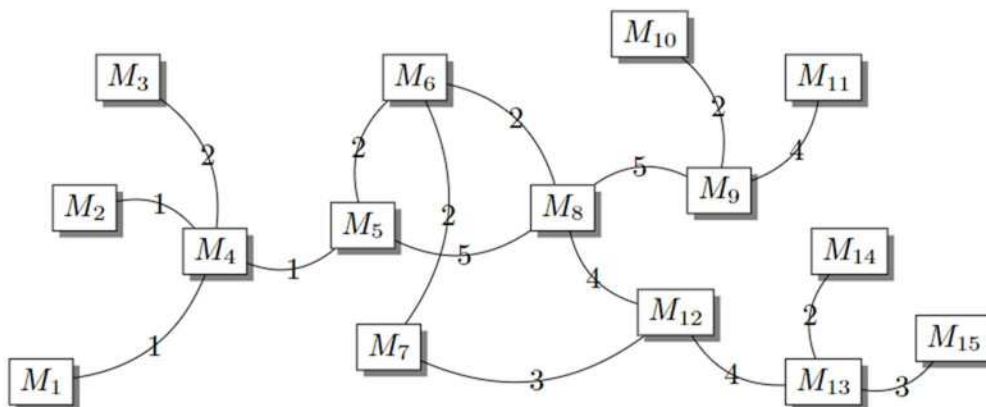
■ ■ Énoncé des exercices

Exercice 14.1 : On considère le graphe suivant, où le nombre situé sur l'arête joignant deux sommets représente une distance :



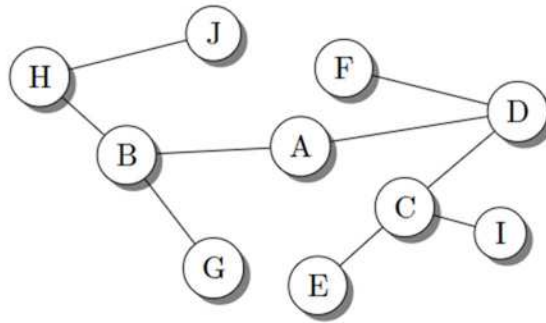
1. Citer un cycle d'origine A et de longueur 3, la longueur étant prise en nombres d'arêtes. Le chemin ne passe pas deux fois par la même arête.
Citer de même un cycle de longueur 4, puis 5, puis 6.
Dans les questions suivantes on utilise les distances (poids représentant les longueurs des arêtes).
2. Quel est le plus court chemin entre A et D ?
3. Quel est le plus court chemin entre A et F ?
4. Combien de chemins entre A et F passent par tous les sommets sans passer deux fois par le même sommet ?

Exercice 14.2 : Le schéma ci-dessous représente un réseau d'appareils connectés qui peuvent être des ordinateurs, des smartphones, des boîtiers internet, des routeurs. Les arêtes représentent les connexions filaires ou sans fil. Les nombres sont les temps de transmission en unité de temps.



1. Combien de routes différentes peut prendre un message entre les machines M_1 et M_{15} ? Une route ne peut passer qu'une seule fois par un appareil donné.
2. Parmi ces routes, quelle est la plus rapide ?

Exercice 14.3 : On considère le graphe représenté ci-dessous :



Expliciter comment on parcourt ce graphe à partir du sommet initial A avec un parcours en largeur d'abord puis avec un parcours en profondeur d'abord.

Exercice 14.4* : On effectue un tri par insertion d'une liste de nombres $[x_1, x_2, \dots, x_n]$.

On suppose que le tri est effectué suivant l'ordre croissant. Une liste vide est triée.

1. Écrire une fonction récursive `insertion` qui insère un élément x à la bonne place dans une pile où les éléments sont empilés du plus grand au plus petit. La pile représente la partie triée de la liste dans l'ordre décroissant. L'élément x et la pile sont les paramètres de la fonction.
2. Écrire une fonction `tri` qui insère les éléments de la liste à trier dans une pile à l'aide de la fonction `insertion`. Lorsque tous les éléments ont été insérés, ils sont retirés de la pile et placés un à un à la bonne place dans la liste qui est alors triée dans l'ordre croissant.

Pour représenter une pile on utilise simplement une liste avec uniquement les méthodes `append`, `pop` et la fonction `len`.

Exercice 14.5* : Notation polonaise inverse

On considère une expression écrite en notation polonaise inverse ou notation postfixe.

Cette expression est représentée par une liste. Par exemple, l'expression « $8\ 3\ +\ 5\ \times$ » est représentée par la liste `[8, 3, '+', 5, '*']`. En mathématiques, la forme d'écriture habituelle est $(8 + 3) \times 5$ et la valeur est 55.

On utilise une pile pour l'algorithme suivant :

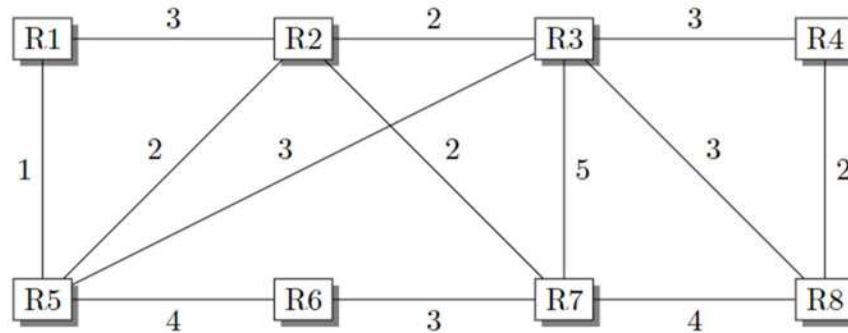
```
Pour e dans expression
    si e est un nombre
        alors on l'empile
    sinon
        on dépile deux opérandes
        on effectue l'opération
        on empile le résultat
On renvoie le résultat
```

1. Écrire une fonction `calcule` qui prend en paramètre une liste représentant une expression comme ci-dessus, notée `exp` et renvoie la valeur de l'expression. Les opérateurs peuvent être "+", "*", "-", ou "/". Une pile est représentée par le conteneur `deque` du module `collections`.
2. Tester la fonction en vérifiant les résultats qui suivent :

<code>calcule([7, 2, '+', 3, '*'])</code>	(réponse : 27);
<code>calcule([2, 5, '*', 4, '+'])</code>	(réponse : 14);
<code>calcule([8, 2, '/', 3, '-'])</code>	(réponse : 1).

Exercice 14.6* : On considère qu'une pile est implémentée par une simple liste avec les méthodes `append` et `pop` respectivement pour empiler et dépiler un élément. Implémenter une file à l'aide de deux piles en écrivant des fonctions `creer_file`, `file_vide`, `enfiler` et `defiler`.

Exercice 14.7 : Voici un réseau. On cherche les plus courtes distances entre chaque sommet et R8. Appliquer l'algorithme de Dijkstra « à la main » en construisant un tableau des distances.



Exercice 14.8* : File de priorité

Considérer dans le cours le programme de parcours en largeur d'abord avec une file. Nommer la fonction `dijkstra_file` et renommer la liste `visite` en liste `distances`. Remplacer la file par une file de priorité : pour cela, écrire au début `from queue import PriorityQueue`, puis dans le corps de la fonction `attente = PriorityQueue()`. Utiliser les méthodes `put`, `get` et `empty` dont la signification est évidente. On ne place pas dans la file un sommet mais un couple (valeur, sommet), par exemple `(0, sommet)` pour le sommet initial. Dans la liste `distances`, on ajoute les couples `[sommet, valeur]`.

Tester la fonction avec l'exemple de l'exercice précédent et comparer le résultat avec celui obtenu par la fonction `dijkstra` du cours.

Exercice 14.9** : Algorithme de Bellman et Ford

L'algorithme de Bellman et Ford suit le principe de la programmation dynamique. Les chemins les plus courts sont déterminés à partir d'un sommet donné `s` jusqu'à tous les autres sommets.

Il s'agit de résoudre des sous-problèmes qui sont les calculs des distances entre le sommet `s` et les autres sommets en au plus `p` arêtes, `p` allant de 1 à `n` où `n` est le nombre de sommets. Ensuite, on combine les solutions.

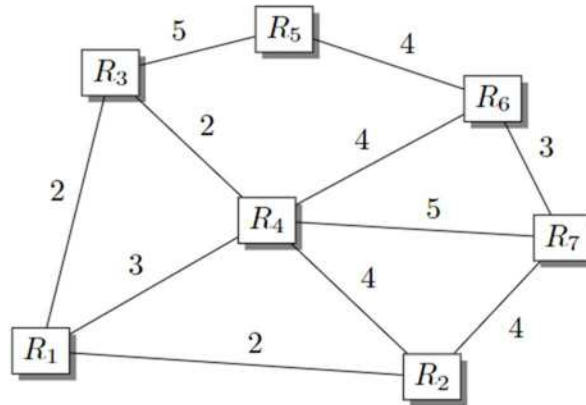
Le principe est le suivant. Toutes les distances initiales d'un sommet au sommet `s` ont une valeur infinie. On choisit un ordre de parcours des sommets. Pour chaque sommet, on visite tous ses voisins qui lui sont directement reliés par une arête et on met à jour les distances. On recommence cette étape `n - 1` fois en suivant le même parcours des sommets. À chaque étape, toutes les arêtes sont visitées au moins une fois. Le graphe peut être orienté ou pas.

Écrire en suivant l'algorithme une fonction `bellman_ford` dont les paramètres sont un graphe `g` et un sommet initial `s`.

Tester le programme avec le graphe de l'exercice 7 et comparer les résultats avec ceux obtenus à l'exercice 8.

Exercice 14.10 : On considère un graphe qui représente une partie d'un réseau constituée de sept appareils. On cherche la route la plus courte entre l'appareil R1 et l'appareil R6. Décrire les différentes étapes lors de l'exécution d'un programme utilisant l'algorithme de Dijkstra. Comparer

avec les étapes d'un programme utilisant l'algorithme A*. L'heuristique choisie est le nombre d'arêtes reliant les sommets multiplié par 3, 3 étant la longueur moyenne supposée d'une arête.



Ce réseau est défini par le dictionnaire `reseau`.

```
reseau = {"R1" : [("R2", 2), ("R3", 2), ("R4", 3)],
          "R2" : [("R1", 1), ("R4", 4), ("R7", 4)],
          "R3" : [("R1", 2), ("R5", 5), ("R4", 2)],
          "R4" : [("R1", 3), ("R2", 4), ("R3", 2), ("R6", 4), ("R7", 5)],
          "R5" : [("R3", 5), ("R6", 4)],
          "R6" : [("R4", 4), ("R5", 4), ("R7", 3)],
          "R7" : [("R2", 4), ("R4", 5), ("R6", 3)]}
```

L'heuristique est définie par :

`heur = "R1": 6, "R2": 6, "R3": 6, "R4": 3, "R5": 3, "R6": 0, "R7": 3`

■ ■ Indications

Ex. 14.5

Traiter en premier les opérateurs.

Ex. 14.6

Faire des dessins pour comprendre l'utilité des deux piles dans la simulation d'une file. Ou bien, prendre un paquet de cartes à jouer, le présenter face visible et retirer les cartes une à une pour les faire défiler. C'est une file. Ensuite, poser une partie des cartes une à une, face non visible, sur une table pour former une première pile. Prendre le paquet déposé et le retourner à côté. C'est une deuxième pile. Retirer alors les cartes une à une, les faces sont visibles.

Ex. 14.8

Écrire la fonction du cours `iteratif_bfs` et tester après chaque modification.

Ex. 14.9

Le programme nécessite trois boucles imbriquées : une boucle pour répéter $n - 1$ fois, une boucle sur les sommets et une boucle sur les voisins de chaque sommet.

■ ■ Corrigé du vrai/faux

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

Quelques explications

1. LIFO pour une pile, FIFO pour une file.
6. Quel que soit le type de parcours, on passe par tous les sommets appartenant à la classe de connexité du sommet de départ.
8. Le calcul de la complexité n'est pas facile, mais on peut vérifier assez simplement qu'il est au pire quadratique.
10. C'est vrai seulement sous certaines conditions.

❑ Erreurs classiques et conseils.—

- Ne pas confondre les structures de pile et de file. Il est recommandé de se créer des images mentales correspondant à l'une et à l'autre.
- Les principes des parcours en profondeur d'abord et en largeur d'abord doivent être compris. L'utilisation d'une structure particulière peut spécifier le type de parcours.
- Le principe de l'algorithme de Dijkstra doit être connu. Il n'est utilisable qu'avec des poids positifs.

■ ■ Corrigé des exercices

Exercice 14.1

1. A-B-E-A constitue un cycle de longueur 3. A-B-F-E-A constitue un cycle de longueur 4. A-B-C-D-E-A constitue un cycle de longueur 5. A-B-C-D-F-E-A constitue un cycle de longueur 6.
2. Le plus court chemin entre A et D est le chemin A-B-C-D (distance totale 4).
3. Le plus court chemin entre A et F est le chemin A-B-F (distance totale 5).
4. Cinq chemins satisfont les conditions énoncées : ce sont les chemins A-B-C-D-E-F, A-C-B-E-D-F, A-C-D-E-B-F, A-E-B-C-D-F, A-E-D-C-B-F.

Exercice 14.2

1. Le message passe obligatoirement par M_4 et M_{12} . Entre M_4 et M_{12} , il peut passer soit uniquement par M_8 , soit par M_8 puis M_6 puis M_7 , soit par M_6 puis M_8 , soit par M_6 puis M_7 .

Nous avons donc 4 routes possibles : M_4 - M_5 - M_8 - M_{12} - M_{13} ou M_4 - M_5 - M_8 - M_6 - M_7 - M_{12} - M_{13} ou M_4 - M_5 - M_6 - M_8 - M_{12} - M_{13} ou M_4 - M_5 - M_6 - M_7 - M_{12} - M_{13} .

2. On calcule le temps de transmission pour chacune des quatre routes. La plus rapide est la route qui passe par M_4 - M_5 - M_6 - M_7 - M_{12} - M_{13} . Le temps total de transmission est 16 unités de temps.

Exercice 14.3

Avec un parcours en largeur d'abord, on commence par les voisins de A, soit B et D. L'ordre B puis D ou D puis B n'a pas d'importance. Ensuite on passe aux voisins de B, soit H et C et aux voisins de D soit C et F. Il reste le voisin de H, soit J, et les voisins de C, soit E et I. On obtient par exemple le parcours : A - B - D - H - G - C - F - J - E - I.

Avec un parcours en profondeur d'abord, on commence par un voisin de A, soit B. Puis on passe à un voisin de B soit H, puis un voisin de H soit J. H n'a pas d'autre voisin, on retourne à B qui a pour autre voisin G. B n'a pas d'autre voisin donc on retourne à A qui a pour voisin D. On passe à un voisin de D, soit F qui n'a pas d'autre voisin. Un autre voisin de D est C qui a pour voisins E puis I. On obtient par exemple le parcours : A - B - H - J - G - D - F - C - E - I.

Exercice 14.4

1. On définit la fonction `insertion` de manière récursive :

```
def insertion(x, pile):
    if len(pile) == 0 or x < pile[len(pile)-1]:
        pile.append(x)
    else:
        y = pile.pop()
        insertion(x, pile)
        pile.append(y)
```

2. La fonction `tri` :

```
def tri(liste):
    n = len(liste)
    pile = []
```



```
# utilisation de la fonction insertion pour ordonner les éléments
for i in range(n):
    insertion(liste[i], pile)
for i in range(n):
    liste[i] = pile.pop()
```

Exercice 14.5

1. Fonction calcule :

```
from collections import deque

def calcule(exp):
    p = deque()
    for e in exp:
        if e in ['+', '*', '-', '/']:
            y = p.pop()
            x = p.pop()
            if e == '+':
                p.append(x + y)
            elif e == '*':
                p.append(x * y)
            elif e == '-':
                p.append(x - y)
            else:
                p.append(x / y)
        else:
            p.append(e)
    return p.pop()
```

2. On teste cette fonction avec l'instruction `assert`.

```
assert calcule([7, 2, '+', 3, '*']) == 27
assert calcule([2, 5, '*', 4, '+']) == 14
assert calcule([8, 2, '/', 3, '-']) == 1
```

Exercice 14.6

```
def creer_file():
    return ([], [])

def file_vide(f):
    return f == ([], [])
```

```
def enfiler(f, x):
    f[0].append(x)

def defiler(f):
    assert not file_vide(f)
    if f[1] == []:
        while f[0] != []:
            f[1].append(f[0].pop())
    return f[1].pop()
```

Test :

```
>>> f = creer_file()
>>> for i in range(5, 12):
    enfiler(f, i)

>>> f
([5, 6, 7, 8, 9, 10, 11], [])
>>> defiler(f)
5
>>> defiler(f)
6
>>> f
([], [11, 10, 9, 8, 7])
>>> for i in range(15, 20):
    enfiler(f, i)

>>> f
([15, 16, 17, 18, 19], [11, 10, 9, 8, 7])
```

Avec cette implémentation, il apparaît que la complexité pour le retrait d'un élément de la file n'est pas la meilleure possible.

Exercice 14.7

On commence en R8 avec une distance nulle. On complète les distances des trois voisins de R8. La plus petite est celle de R4 que l'on garde. On met à jour avec les voisins de R4, rien ne change. La distance la plus petite est celle de R3 que l'on garde et on met à jour avec les voisins de R3. Et ainsi de suite.

R1	∞	∞	∞	∞	∞	8	7	7
R2	∞	∞	∞	5	5			
R3	∞	3	3					
R4	∞	2						
R5	∞	∞	∞	6	6	6		
R6	∞	∞	∞	∞	7	7	7	
R7	∞	4	4	4				
R8	0							

Exercice 14.8

On copie la fonction du cours et on effectue les modifications :

```
from queue import PriorityQueue

def dijkstra_file(graphe, sommet):
    distances = []
    marque = {}
    attente = PriorityQueue() # une file de priorité (distance minimale)
    attente.put((0, sommet))
    while not attente.empty():
        valeur, sommet = attente.get()
        if sommet not in marque:
            distances.append((sommet, valeur))
            marque[sommet] = True
            #print(sommet, valeur, end=" ")
            for s, v in graphe[sommet]:
                if s not in marque:
                    attente.put((valeur + v, s))
    return distances
```

Pour tester le programme, on définit le graphe représentant l'exemple de l'exercice 7 avec un dictionnaire des listes d'adjacence :

```
graphe = {"R1" : [("R2", 3), ("R5", 1)],
          "R2" : [("R1", 3), ("R3", 2), ("R5", 2), ("R7", 2)],
          "R3" : [("R2", 2), ("R4", 3), ("R5", 3), ("R7", 5), ("R8", 3)],
          "R4" : [("R3", 3), ("R8", 2)],
          "R5" : [("R1", 1), ("R2", 2), ("R3", 3), ("R6", 4)],
          "R6" : [("R5", 4), ("R7", 3)],
          "R7" : [("R2", 2), ("R3", 5), ("R6", 3), ("R8", 4)],
          "R8" : [("R3", 3), ("R4", 2), ("R7", 4)]}
```

On écrit les fonctions `minimum` et `dijkstra` du cours pour comparer les résultats.
On exécute le programme :

```
>>> dijkstra_file(graphe, 'R8')
[('R8', 0), ('R4', 2), ('R3', 3), ('R7', 4), ('R2', 5), ('R5', 6), ('R1', 7), ('R6', 7)]
>>> dijkstra(graphe, "R8")
{'R8': 0, 'R4': 2, 'R3': 3, 'R7': 4, 'R2': 5, 'R5': 6, 'R1': 7, 'R6': 7}
```

On constate qu'on obtient évidemment des résultats identiques.

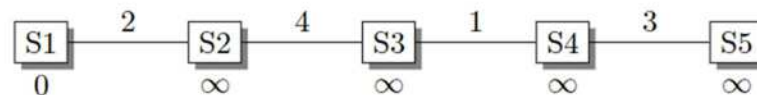

```
def bellman_ford(g, s):
    n = len(g)
    d = {k: float('inf') for k in g}
    d[s] = 0
    for i in range(n-1): # on répète n-1 fois
        for k in g: # puis deux boucles for pour explorer toutes les arêtes
            for j in range(len(g[k])):
                v, c = g[k][j]
                d[v] = min (d[v], d[k] + c)
    return d
```

Test avec le graphe défini dans l'exercice précédent :

```
>>> bellman_ford(graphe, 'R8')
{'R1': 7, 'R2': 5, 'R3': 3, 'R4': 2, 'R5': 6, 'R6': 7, 'R7': 4, 'R8': 0}
```

Les résultats sont les mêmes que ceux obtenus avec la fonction `dijkstra`.

Examinons le déroulement des étapes avec le graphe ci-dessous qui est défini de manière à parcourir les sommets suivant l'ordre S5, S4, S3, S2, S1 :



```
g = {"S5" : [("S4", 3)],
     "S4" : [("S5", 3), ("S3", 1)],
     "S3" : [("S4", 1), ("S2", 4)],
     "S2" : [("S3", 4), ("S1", 2)],
     "S1" : [("S2", 2)]}
```

étape 1 : {'S5': inf, 'S4': inf, 'S3': inf, 'S2': 2, 'S1': 0}

étape 2 : {'S5': inf, 'S4': inf, 'S3': 6, 'S2': 2, 'S1': 0}

étape 3 : {'S5': inf, 'S4': 7, 'S3': 6, 'S2': 2, 'S1': 0}

étape 4 : {'S5': 10, 'S4': 7, 'S3': 6, 'S2': 2, 'S1': 0}

Explication de l'étape 1 : toutes les distances à S1 sont infinies à part celle de S1 à lui-même qui vaut 0. Donc la distance de S5 à S1 vaut la distance de S5 à S4 plus la distance de S4 à S1, $3 + \infty$, soit l'infini. La distance de S4 à S1 vaut la distance de S4 à S3 plus la distance de S3 à S1, $1 + \infty$, soit l'infini. La distance de S3 à S1 vaut la distance de S3 à S2 plus la distance de S2 à S1, $4 + \infty$, soit l'infini. La distance de S2 à S1 vaut la distance de S2 à S1 plus la distance de S1 à S1, $2 + 0$, soit 2. C'est la fin de l'étape 1.

Exercice 14.10

Avec l'algorithme de Dijkstra, on obtient une nouvelle valeur à chaque étape en utilisant le sommet à distance minimale. Le sommet R6 est obtenu à la dernière étape :

- étape 1 : { 'R1' : 0, 'R2' : inf, 'R3' : inf, 'R4' : inf, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 2 : { 'R1' : 0, 'R2' : 2, 'R3' : inf, 'R4' : inf, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 3 : { 'R1' : 0, 'R2' : 2, 'R3' : 2, 'R4' : inf, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 4 : { 'R1' : 0, 'R2' : 2, 'R3' : 2, 'R4' : 3, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 5 : { 'R1' : 0, 'R2' : 2, 'R3' : 2, 'R4' : 3, 'R5' : inf, 'R6' : inf, 'R7' : 6 }
- étape 6 : { 'R1' : 0, 'R2' : 2, 'R3' : 2, 'R4' : 3, 'R5' : 7, 'R6' : inf, 'R7' : 6 }
- étape 7 : { 'R1' : 0, 'R2' : 2, 'R3' : 2, 'R4' : 3, 'R5' : 7, 'R6' : 7, 'R7' : 6 }

Avec l'algorithme A*, trois étapes seulement sont utilisées. Grâce à l'heuristique, le premier sommet examiné est R4 qui amène à l'étape suivante à R6 :

- étape 1 : { 'R1' : 0, 'R2' : inf, 'R3' : inf, 'R4' : inf, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 2 : { 'R1' : 0, 'R2' : inf, 'R3' : inf, 'R4' : 3, 'R5' : inf, 'R6' : inf, 'R7' : inf }
- étape 3 : { 'R1' : 0, 'R2' : inf, 'R3' : inf, 'R4' : 3, 'R5' : inf, 'R6' : 7, 'R7' : inf }

■ Annexe

Chapitre **15**

Langage Python

UN SCIENTIFIQUE



Guido van Rossum, le créateur du langage de programmation Python, est né en 1956 à Amsterdam. Il effectue des études de mathématiques dans sa ville natale et obtient un master en 1982. Il travaille alors en tant que développeur dans la conception du langage ABC. Il s'en inspire pour élaborer le langage Python en 1991. Passionné de l'émission télévisuelle *Monty Python's Flying Circus* qui était programmée à la BBC entre 1969 et 1974, il a nommé PYTHON le langage qu'il venait de concevoir.

■ Un peu d'histoire

Les premiers langages apparus dans les années 1950 comme FORTRAN ou COBOL s'adressaient uniquement à des spécialistes. Depuis l'apparition des ordinateurs personnels, il s'est avéré nécessaire de créer des langages plus conviviaux et adaptés à un plus grand nombre. Surtout, l'entrée dans l'enseignement de l'informatique au niveau des lycées a fait naître de nouveaux besoins.

Le Basic, apparu en 1964 et très inspiré du FORTRAN, ne convenant pas à ce type de tâche, il a laissé place au Turbo Pascal qui connaît une grande vogue dans les années 1980.

Depuis une dizaine d'années, PYTHON est devenu le langage le plus utilisé dans l'enseignement français. Conçu par l'informaticien Guido van Rossum, il est facile à apprendre, avec des lignes de programme claires et concises. En outre, sa portée est générale et on l'utilise pour de nombreux logiciels en *open source*.

■ ■ Langage Python

■ Traits généraux

Le langage Python utilise des objets. Un objet possède une valeur et un ensemble de propriétés. On confond souvent l'objet avec la valeur lorsque cela peut simplifier le propos. Par exemple on parle de la valeur 3 qui désigne en fait un objet de type entier, qui a pour valeur 3, avec lequel diverses opérations sont permises et qui est placé en mémoire à une adresse précise.

■ Variables et affectation

De manière générale, les données utilisées par un programme sont stockées dans la machine à l'aide de *variables*. Ceci se fait par une *affectation* qui associe un *nom* à une donnée, représentée par une valeur ou une expression qui a une valeur.

Quand on dit « variable », on peut penser à une boîte, située dans la mémoire de la machine, dans laquelle on place des informations diverses définissant un objet et à laquelle on lie un nom. Plusieurs noms peuvent être liés à la même boîte. Le nom peut être n'importe quelle chaîne alphanumérique, exceptés certains mots réservés, sans commencer par un chiffre.

L'opérateur d'affectation est noté "=". L'*instruction* `x = 5+3` place dans une boîte en mémoire la valeur 8 de l'expression `5+3`, ainsi que diverses informations (définissant l'objet 8) et lie le nom `x` à cette boîte. Lorsqu'on parle de la valeur de la variable `x`, il s'agit de la valeur de l'objet auquel est lié le nom `x`.

Le vocabulaire est important et la signification des termes *expression*, *instruction* et *affectation* doit être parfaitement comprise.

- Une **expression** a une valeur qui est le résultat calculé d'opérations sur les valeurs d'objets.
- Une **instruction** est une commande qui doit être exécutée par la machine.
- Une **affectation** est une instruction qui lie un nom, existant ou nouveau, à une valeur (la valeur d'un objet).

Une expression est placée à droite du signe `=` dans une affectation.

Note : la pratique en Python est de ne pas utiliser les lettres capitales dans les noms de variables ou de fonctions ; nous écrivons `variable`, `ma_variable`, `ma_fonction` mais pas `Variable` ni `MaVariable` ou `MaFonction`.

■ Fonctions

La syntaxe pour définir une fonction est :

```
def nom_de_la_fonction(arguments):  
    """ aide sur la fonction, non obligatoire  
        mais recommandée """  
    corps de la fonction
```


Le mot `def` est un mot clé du langage. Les arguments sont séparés par des virgules. Il peut ne pas y avoir d'argument. La première ligne, ligne de définition, se termine par le signe deux-points.

La partie entre les triples guillemets est une aide, une indication sur ce qu'on attend des paramètres et sur ce qu'on attend du résultat. Il s'agit d'une spécification, importante pour tout utilisateur de la fonction.

Le corps de la fonction est un bloc de code indenté par rapport à la ligne de définition.

■ Principe d'indentation

L'indentation est un élément de syntaxe. Un groupe d'instructions est indenté à la suite du signe deux-points, donc dans la définition d'une fonction, dans un test, une alternative, une boucle.

Considérons une fonction f de paramètre x .

L'exécution de $f(e)$ consiste à évaluer la valeur de e et à l'affecter à la variable locale x , puis à exécuter le code constituant le corps de la fonction.

La fonction peut renvoyer un résultat si le corps de la fonction contient l'instruction `return`. Dans ce cas l'appel de la fonction est une expression qui a une valeur. Il peut y avoir plusieurs instructions `return` mais chacune interrompt l'exécution de la fonction. S'il n'y a pas d'instruction `return` dans le corps de la fonction, alors l'appel de la fonction a la valeur `None`. C'est le cas par exemple de la fonction `print`. Ce type de fonction s'appelle une procédure.

Remarque : le mot `return` est un mot clé du langage comme le mot `def`. Ce n'est pas le nom d'une fonction mais d'une instruction et il n'est pas nécessaire de le faire suivre de parenthèses.

■ Espace de nom et portée des variables

Les noms sont enregistrés dans différents *espaces de noms* : un espace *global* dans lequel le programme et en particulier des variables globales sont définis et pour chaque fonction un espace *local*. L'espace de noms d'une fonction est initialisé lors de l'appel avec les noms des arguments qui sont liés aux objets passés en arguments. Il contient aussi les variables définies dans son corps. Ces variables locales n'existent que pendant l'utilisation de la fonction. Pour qu'une variable puisse être utilisée dans le corps d'une fonction, elle doit appartenir à l'espace local de la fonction, ou à l'espace dans lequel la fonction a été appelée.

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche, pour obtenir sa valeur, si le nom de la variable est dans son espace local et à défaut cherche dans l'espace global du module. La *portée d'une variable* ou *portée lexicale* est une région du programme, une partie de texte, où un espace de noms est accessible à la variable.

Une fonction ne peut pas modifier par affectation la valeur d'une variable extérieure à son espace local puisqu'une affectation crée une nouvelle variable locale qui est détruite après l'utilisation de la fonction. Dans le code suivant, la variable locale x utilisée dans la fonction est distincte de la variable globale x définie au début du programme par l'instruction $x=1$ et n'existe plus après l'appel de la fonction.

```
x = 1
def f(x):
    x = x + 1
    return x

print(f(x)) # affiche 2
print(x)    # affiche 1
```


Il est cependant possible de modifier avec une fonction une variable extérieure à celle-ci en la déclarant comme variable globale avec le mot-clé `global`. Mais ce procédé est plutôt à éviter.

■ Exécution d'une fonction

La définition d'une fonction permet de grouper un ensemble d'instructions, ensemble auquel on donne un nom, afin de pouvoir utiliser cet ensemble plusieurs fois en l'appelant par son nom.

```
def f(x, y):  
    instructions  
    return res  
  
r = f(a, b)
```

L'instruction `r = f(a, b)` permet de résumer le code suivant :

```
x_f = a  
y_f = b  
instructions  
r = res
```

Les écritures `x_f` et `y_f` sont utilisées pour différencier les variables `x` et `y` locales à la fonction. C'est important visuellement en particulier lorsqu'on écrit `f(x, y)`.

Note : si on écrit `var = v1`, `v1` est la valeur d'un objet auquel le nom `var` est lié. Si on écrit ensuite `var = v2`, le nom `var` est lié à un nouvel objet qui a une valeur `v2`. Le premier objet est détruit en mémoire s'il n'est plus utilisé.

Après un appel `f(x, y)`, les noms `x` et `y` restent liés aux mêmes objets auxquels ils étaient liés avant l'appel. Deux nouveaux noms `x_f` et `y_f` sont liés respectivement à chacun de ces objets. Les valeurs des variables `x` et `y` ne peuvent donc pas être modifiées si ces objets sont non mutables. Elles peuvent l'être si ces objets sont mutables comme des listes.

■ Structures de contrôle

■ Instruction conditionnelle

La structure la plus simple est :

```
if condition:  
    instructions
```

Le mot `condition` désigne une expression qui a la valeur `True` ou `False` et le mot `instructions` désigne une instruction ou un bloc d'instructions écrites sur plusieurs lignes. Le bloc `instructions` qui suit le signe deux-points est indenté. Si la condition est vérifiée, si l'expression a la valeur `True`, alors les instructions sont exécutées.

On peut mentionner aussi la structure `if-else` qui présente une alternative ou la structure `if-elif-else` qui présente plusieurs alternatives.

Note : les mots `True` et `False` sont avec `None` les rares mots dont l'écriture en Python commence par une lettre capitale.

■ Boucles conditionnelles

La structure est :

```
while condition:
    instructions
```

La structure est identique à celle de l'instruction conditionnelle `if`. La condition qui suit le mot `while` est une expression dont la valeur interprétée peut être `True` ou `False`. Le bloc d'instructions qui suit, indenté, est exécuté si la condition est vérifiée. Si la condition est encore vérifiée après l'exécution des instructions, le bloc est à nouveau exécuté, et ainsi de suite. Ce bloc d'instructions doit donc modifier la valeur de l'expression à tester afin que la condition ne soit plus vérifiée après un nombre fini d'exécutions du bloc. Si le programme a une suite, elle pourra alors être exécutée.

■ Boucles non conditionnelles

Exemples :

```
for i in range(10):
    bloc d'instructions
for x in [1, 2, 3, 4]: # parcours d'une liste
    bloc d'instructions
for car in 'test': # parcours d'une chaîne de caractères
    bloc d'instructions
```

La dernière boucle par exemple revient à effectuer :

```
car = 't'
bloc d'instructions
car = 'e'
bloc d'instructions
car = 's'
bloc d'instructions
car = 't'
bloc d'instructions
```

Une variable `car` est créée. Elle a pour valeur `'t'` à la fin de la boucle et n'est pas détruite.

Avec une chaîne ou une liste, les éléments de la séquence parcourue ont un indice prenant les valeurs de 0 à `len(sequence)-1`, et si on a besoin des valeurs de ces indices, on utilise l'objet `range(n)` qui produit les entiers de 0 à `n-1`. On pourrait d'ailleurs s'en passer puisque les écritures `for i in range(4)` ou `for i in [0, 1, 2, 3]` produisent le même effet.

Suivant les cas, avec des boucles sur des chaînes ou des listes, il peut être nécessaire d'utiliser les indices ou pas.

`range(d, f)` renvoie la séquence des entiers successifs de `d` inclus à `f` exclu si `f > d` sinon rien.
`range(f)` renvoie la séquence des entiers successifs de 0 inclus à `f` exclu si `f > 0` sinon rien.

L'instruction `break` permet d'interrompre une boucle. Si la boucle est écrite dans le corps d'une fonction, une instruction `return` permet aussi d'arrêter la boucle (puisque cette instruction arrête immédiatement l'exécution de la fonction).

■ Types

Le *type* d'une variable est le type de l'objet auquel peut être lié le nom de la variable. Il définit l'ensemble des valeurs autorisées, les opérations et les fonctions utilisables.

■ Types de base

Les types de base sont les types numériques `int`, `bool` et `float`.

Le type `int` est utilisé pour représenter les nombres entiers. Leur taille n'est limitée que par la capacité de la machine et le temps nécessaire à leur utilisation. Un calcul avec de très grands entiers est possible en Python mais peut prendre un temps important.

Le type `bool` permet de représenter les valeurs booléennes `True` (vrai) et `False` (faux). Ces valeurs peuvent être considérées comme les entiers 1 pour `True` et 0 pour `False`.

Le type `float` est utilisé pour représenter les nombres réels. Les nombres du type `float`, forcément en nombre fini, sont des décimaux qui servent à approximer les réels. Ils sont stockés dans la machine sous la forme de « nombres en virgule flottante » avec des valeurs comprises entre environ $-1,7 \times 10^{308}$ et $1,7 \times 10^{308}$.

Citons aussi le type `None` qui n'a qu'une seule valeur, la valeur `None`. C'est par exemple la valeur que renvoie une fonction sans instruction `return`.

■ Opérations sur les types numériques

Nous avons les opérations mathématiques habituelles : addition, soustraction, multiplication, exponentiation et division notées respectivement `a + b`, `a - b`, `a * b`, `a ** b` et `a / b`.

La division entière `//` et l'opération modulo `%` utilisées avec des entiers naturels, donnent respectivement le quotient et le reste, de type `int`, dans la division euclidienne.

■ Opérateurs de comparaison et opérateurs booléens

Les opérateurs mathématiques classiques de comparaisons `=`, `≠`, `<`, `≤`, `>`, `≥` s'écrivent en Python respectivement `==`, `!=`, `<`, `<=`, `>` et `>=`.

`x == y` a pour valeur `True` si `x` et `y` ont la même valeur, sinon a pour valeur `False`.

`x != y` a pour valeur `True` si `x` et `y` n'ont pas la même valeur, sinon a pour valeur `False`.

Les opérateurs logiques sont `and`, `or`, `not` :

`a and b` a pour valeur `True` si `a` et `b` ont la valeur `True` et sinon a pour valeur `False`;

`a or b` a pour valeur `False` si `a` et `b` ont la valeur `False` et sinon a pour valeur `True`;

`not a` a pour valeur `True` si `a` a la valeur `False` et sinon a pour valeur `False`.

Ces opérateurs fonctionnent de manière séquentielle, on parle de « caractère paresseux ». Avec `a and b`, `b` n'est évalué que si `a` vaut `True`. Avec `a or b`, `b` n'est évalué que si `a` vaut `False`.

■ Le type `str`

C'est un type simple mais structuré. Le mot `str` est une abréviation de `string`. Ce type est utilisé pour représenter des chaînes de caractères, par exemple ce qui est obtenu dans la machine lorsqu'on appuie sur les touches du clavier. On utilise des guillemets ou des apostrophes comme `ch = "bonjour"` ou `ch = 'bonjour'`. Un caractère est une chaîne de longueur 1.

On obtient la longueur d'une chaîne, le nombre de caractères contenus dans la chaîne, avec la fonction `len`. Par exemple, `len("bonjour")` a pour valeur l'entier 7.

Chaque caractère d'une chaîne a un indice qui va de 0 à $n - 1$ si n est la longueur de la chaîne et il est possible d'accéder au caractère d'indice `i` d'une chaîne `ch` avec la notation `ch[i]`.

Nous pouvons aussi accéder à une suite de caractères d'une chaîne `ch` avec la notation `ch[i:j]` qui est une chaîne de caractères contenant dans l'ordre les caractères de `ch` d'indices `i` inclus à `j` exclu. On parle aussi d'une *tranche*.

■ Types structurés composés

Ce sont par exemple les types `tuple`, `list`, `dict`.

Les dictionnaires de type `dict` sont décrits plus loin dans une section particulière.

Un objet de type `list` ou `tuple` est composé de zéro, un ou plusieurs objets, à priori de n'importe quel type. Ce sont des types *itérables* comme le type `str`.

Un objet de type `list` ou `tuple` représentent un ensemble ordonné. Les éléments sont indexés par un entier commençant à 0 et se terminant à $n - 1$ si n est le nombre d'éléments.

Ce nombre d'éléments est obtenu avec la fonction `len`.

L'accès à un élément particulier ou à une suite d'éléments s'obtient comme pour les chaînes de caractères. On utilise la syntaxe avec crochets : par exemple `x = liste[i]` permet d'affecter à `x` l'élément d'indice `i`.

Un point très important est qu'un élément d'une liste peut être modifié avec une instruction d'affectation alors que c'est interdit avec le type `tuple` (il n'est pas mutable). On peut écrire pour une liste `liste[0] = val`, mais avec un `tuple` ce type d'instruction provoque une erreur.

De nombreuses méthodes permettent de travailler avec des listes. La syntaxe doit être bien connue : le nom de la méthode s'écrit après le nom de l'objet auquel elle s'applique, les deux noms étant séparés par un point sans espace.

La méthode `append` est très souvent employée. L'instruction `liste.append(x)` ajoute, à la fin de la liste `liste`, l'objet `x`. Cette méthode sert en particulier à construire une liste en ajoutant les éléments un par un à la fin de la liste.

■ Opérations communes aux types `list`, `tuple` et `str`

► Appartenance

On utilise le mot `in`. L'expression `x in obj` a pour valeur `True` si `x` est un élément de `obj` et sinon a pour valeur `False`.

► Concaténation

Si `obj1` et `obj2` sont du même type, `obj1 + obj2` est la concaténation de `obj1` et `obj2`.

`n * obj` est la concaténation de n copies de `obj` si n est un entier naturel.

La fonction `type` permet de déterminer le type d'un objet. Les fonctions `int`, `float`, `str` permettent de convertir si c'est possible un objet d'un autre type respectivement en un objet de type `int`, `float`, `str`.

■ Typage dynamique

On dit que l'interpréteur détermine le type à la volée lors de l'exécution du code.

Ceci signifie qu'après une instruction comme `x = 3`, la variable `x` est du type `int`, alors qu'après une instruction comme `y = 3.0`, la variable `y` est du type `float`. Le type n'est pas déclaré, c'est l'interpréteur qui le détermine en fonction de l'écriture. Après une instruction comme `x = x + y`, la variable `x` est du type `float` (sa valeur est 6.0).

■ Les listes

En langage Python, une liste est un objet de type `list` qui est un ensemble ordonné d'éléments hétérogènes de types quelconques.

Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

■ Génération

- En extension : `liste = [1, 3, 5, 7, 9]`
- En compréhension : `liste = [2*i+1 for i in range(5)]`
- Par conversion : `liste = list(range(1, 10))`
- Par ajouts successifs :

```
liste = []  
for i in range(1, 10):  
    liste.append(i)
```

De manière générale, on crée en compréhension une liste avec une instruction comme :
`liste = [f(u) for u in objet]`, où `f` est une fonction et `objet` est de type `str` ou `list`.

■ Manipulation

La fonction `len` : l'expression `len(liste)` a pour valeur la longueur de la liste, c'est-à-dire le nombre d'éléments de la liste.

On accède à chaque élément par des indices qui prennent les valeurs de 0 à `len(liste)-1`.

Avec `liste = [1, 3, 5]`, `liste[0]` vaut 1, `liste[1]` vaut 3, `liste[2]` vaut 5 et une écriture comme `liste[3]` provoque une erreur et l'arrêt d'un programme.

Soit : `liste = [1, 3, 5, 7, 9]`.
`liste[1:4]` vaut `[3, 5, 7]`.
`liste[:4]` vaut `[1, 3, 5, 7]` (par défaut l'indice de début `d` vaut 0).
`liste[1:]` vaut `[3, 5, 7, 9]` (par défaut l'indice de fin `f` vaut `len(liste)`).
Et donc : `liste[:]` vaut `[1, 3, 5, 7, 9]`.

Une liste vide : `liste = list()` ou `liste = []`.

Les opérateurs `+` et `*` concatènent des listes pour créer une nouvelle liste :

l'expression `[1, 2] + [3, 4]` vaut `[1, 2, 3, 4]` ;

l'expression `3 * [1, 2]` vaut `[1, 2, 1, 2, 1, 2]`.

Les opérateurs `+` et `*` n'ont pas d'effet de bord, de nouvelles listes sont créées.

L'expression `x in liste` vaut `True` si `x` est un élément de `liste`, sinon vaut `False`.

Le type `list` est mutable : on peut modifier la valeur d'un élément par une affectation.

Attention, en écrivant `liste = [1]` puis `liste = [3]`, on ne change pas la valeur du premier objet `[1]` auquel est lié `liste`, mais on crée un nouvel objet `[3]` auquel on lie `liste`. Par contre avec `liste = [2, 4]` puis `liste[1] = 6`, on change la valeur de `[2, 4]` qui devient `[2, 6]`.

Il existe plusieurs méthodes spécifiques aux listes. Les méthodes `append` et `pop`, modifient une liste respectivement en ajoutant un élément en fin de liste ou en supprimant et renvoyant l'élément en fin de liste. Par exemple :

```
liste = [1, 2, 3]  
liste.append(4)  (liste vaut [1, 2, 3, 4])  
liste.pop()     (a pour valeur 4, l'élément est retiré de la liste, liste vaut [1, 2, 3]).
```

On peut créer une liste de listes : `liste = [[1, 2], [3, 4], [5, 6]]`.

`liste[i][j]` est l'élément d'indice `j` de la liste d'indice `i`. Par exemple, `liste[1][0]` est l'élément d'indice 0 de la liste `[3, 4]`, soit le nombre 3.

■ Parcours et itération

Nous avons deux manières de parcourir les éléments d'une liste.

Si nous n'avons pas besoin de la valeur de l'indice :

```
liste = [1, 3, 5, 7, 9]
```

```
for u in liste:
    print(u)
```

Si nous avons besoin de la valeur de l'indice :

```
for i in range(len(liste)):
    print(i, liste[i])
```

■ Copie d'une liste

La souplesse d'utilisation d'une liste est pratique mais demande de la vigilance. Considérons les instructions `liste1 = [1, 3, 5]`, puis `liste2 = liste1`. Nous avons ici deux noms liés à un même objet, une liste. Donc après l'instruction `liste2[1] = 4` censée modifier `liste2`, l'objet auquel est lié `liste1` est la liste `[1, 4, 5]`. On dit que `liste1` a été modifiée.

Pour obtenir une véritable copie d'une liste simple, c'est-à-dire créer un nouvel objet distinct de l'objet initial, on peut utiliser le code `liste2 = list(liste1)`.

La méthode `copy()` renvoie une nouvelle liste. Après l'instruction `liste2 = liste1.copy()`, `liste1` et `liste2` sont liés à deux objets distincts dans le sens où ils sont enregistrés en mémoire à des adresses différentes. Mais attention, les adresses de leurs éléments sont identiques.

Le module `copy` propose des fonctions de copie, en particulier dans le cas de listes emboîtées. Pour copier en profondeur une liste de listes, nous utilisons la fonction `deepcopy` avec les instructions : `from copy import deepcopy`, puis `liste2 = deepcopy(liste1)`.

■ Dictionnaires

Dans une liste, un n-uplet ou une chaîne de caractères, les éléments sont ordonnés. On les repère par leur indice. Si n est la longueur, à chaque entier de 0 à $n - 1$ correspond un élément.

■ Définition

Un dictionnaire, objet de type `dict`, est une association entre des clés et des valeurs. Les clés sont des objets non mutables, les valeurs des objets quelconques. Ces clés ne sont pas ordonnées. On accède à une clé en temps constant selon un processus qu'il n'est pas nécessaire de décrire ici. On accède à une valeur par sa clé.

Pour créer un dictionnaire, on écrit entre des accolades les couples `clé: valeur`, une clé et la valeur sont séparés par le signe deux-points, les couples sont séparés par des virgules.

```
Exemple : jours = {"dimanche": 1, "lundi": 2, "mardi": 3, "mercredi": 4,
                  "jeudi": 5, "vendredi": 6, "samedi": 7}.
```

■ Manipulation

La construction par compréhension existe pour les dictionnaires.

```
>>> d = {x: x**2 for x in range(1, 5)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}
```


■ Accès aux clés

La méthode `keys` permet d'obtenir les différentes clés.

```
>>> jours.keys()
dict_keys(['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi',
           'samedi'])
```

La méthode `items` permet d'obtenir l'ensemble des couples (clé, valeur).

```
>>> jours.items()
dict_items([('dimanche', 1), ('lundi', 2), ('mardi', 3), ('mercredi', 4),
           ('jeudi', 5), ('vendredi', 6), ('samedi', 7)])
```

Le mot `in` permet de tester l'appartenance d'une clé à un dictionnaire, pas d'une valeur.

```
>>> 4 in jours
False
>>> "dimanche" in jours
True
>>>
```

Donc, avec `for elt in dic`, où `dic` est un dictionnaire, la variable d'itération `elt` est une clé.

```
>>> cle = []
>>> for elt in jours:
        cle.append(elt)
```

La variable `cle` a pour valeur `['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']`.

Mais il est aussi possible d'itérer sur les couples clés-valeurs en utilisant la méthode `items`.

```
>>> for cle, val in jours.items():
        print(cle, val)

dimanche 1
lundi 2
mardi 3
mercredi 4
jeudi 5
vendredi 6
samedi 7
```

L'accès à une valeur s'obtient comme avec les listes, les tuples ou les chaînes. Mais il faut préciser la clé à la place de l'indice. Par exemple, `jours['dimanche']` nous donne la valeur 1.

■ Insertion, modification

Comme pour les listes, il est possible de modifier une valeur par affectation, par exemple avec une instruction comme `jours['dimanche'] = 0`. Pour insérer un élément, on écrit une instruction comme `jours['dimanche2'] = 8`. On ajoute un deuxième dimanche à la semaine.

Si la clé `c` existe, l'instruction `d[c] = val` modifie la valeur associée à la clé `c`. Si la clé `c` n'existe pas, elle est créée et la valeur `val` lui est associée.

■ **Nombre d'éléments** : comme avec les listes, les n-uplets et les chaînes, la fonction `len` renvoie la longueur d'un dictionnaire qui est à la fois le nombre de clés et le nombre de couples.

■ Copie

Les comportements sont similaires à ceux rencontrés avec les listes.

```
>>> d1 = {'one': 1, 'two': 2, 'three': 3}
>>> d2 = d1
>>> d2['one'] = 0
>>> d2['four'] = 4
>>> d1
{'one': 0, 'two': 2, 'three': 3, 'four': 4}
```

Dans le code précédent, `d1` et `d2` sont deux noms différents liés au même dictionnaire. Avec le code qui suit, le nom `d2` est lié à un nouveau dictionnaire.

```
>>> d1 = {'one': 1, 'two': 2, 'three': 3}
>>> d2 = dict(d1)
>>> d2['one'] = 0
>>> d1
{'one': 1, 'two': 2, 'three': 3}
```

Nous pouvons aussi utiliser la méthode `copy` en écrivant `d2 = d1.copy()`. Mais attention, si les valeurs sont des listes, les questions posées sur les copies de listes sont à nouveau présentes.

Dans le doute, pour obtenir une vraie copie, il est plus sûr d'utiliser la fonction `deepcopy` du module `copy` qui se comporte comme avec les listes.

■ Résumé

Itérable : objet qui peut renvoyer ses éléments un par un.

Séquence : itérable avec une méthode `len` et un accès aux éléments par un indice. Les indices sont des entiers allant de 0 à $n - 1$ si n est la longueur de la séquence. Les éléments sont ordonnés suivant leur indice.

Un dictionnaire est un itérable mais n'est pas une séquence : l'accès aux éléments s'obtient par une clé arbitraire immuable. Un dictionnaire est aussi appelé *tableau associatif*.

Type	Valeur	Itérable	Séquence	Mutable
int	entier	non	non	non
bool	0 ou 1	non	non	non
float	décimal	non	non	non
str	tableau de caractères	oui	oui	non
tuple	tableau d'adresses	oui	oui	non
list	tableau d'adresses	oui	oui	oui
dict	tableau d'adresses	oui	non	oui

■ Mots, signes, caractères spéciaux

Le langage Python utilise des mots clés, des signes, des caractères spéciaux. Une partie seulement doit être parfaitement maîtrisée. Elle est rassemblée ci-dessous.

```
=
def return
if elif else for while break
int bool float str list dict range
False True None
len copy
append pop
split
keys items
print assert
from import as
open close read readline readlines write
in
and or not
+ - * / ** // %
== != < <= > >=
: , .
' " #
( ) [ ] { }
```

■ Complément

En mémoire les objets sont stockés sur un nombre d'octets à une place où est inscrite leur valeur mais aussi leur type (plutôt l'adresse en mémoire où se trouve le type). Pour les listes et les tuples, le nombre d'éléments est enregistré, (pour la fonction `len`), ainsi qu'un tableau d'adresses contenant les adresses des différents éléments. Les éléments sont enregistrés à une autre place.

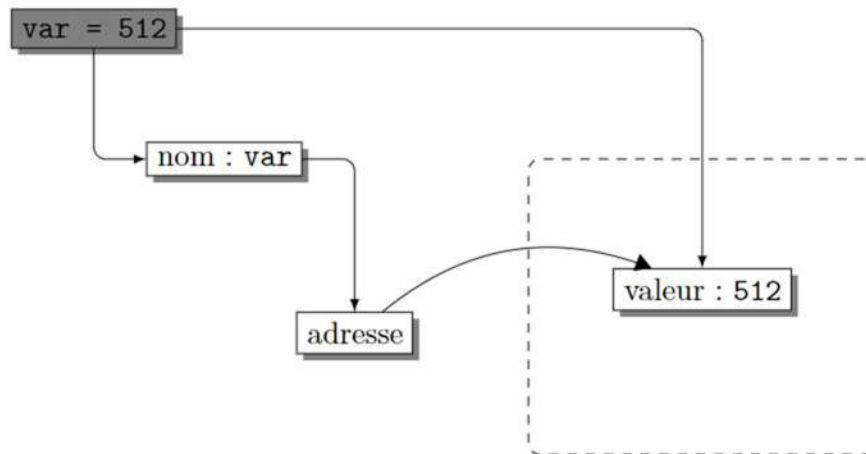
Le signe `=` traduit une *affectation*. Ceci signifie qu'avec l'instruction `var = val`, la variable qui a pour nom `var` est associée à la valeur `val` qui est d'un certain type et qui est stockée en mémoire à une certaine adresse. On dit pour simplifier que « la variable `var` prend la valeur `val` ».

Le nom est créé, la valeur écrite en mémoire. Le nom est lié à la valeur par l'adresse en mémoire où se situe la valeur. Cette adresse permet au nom d'accéder à la valeur mais ce n'est pas réciproque.

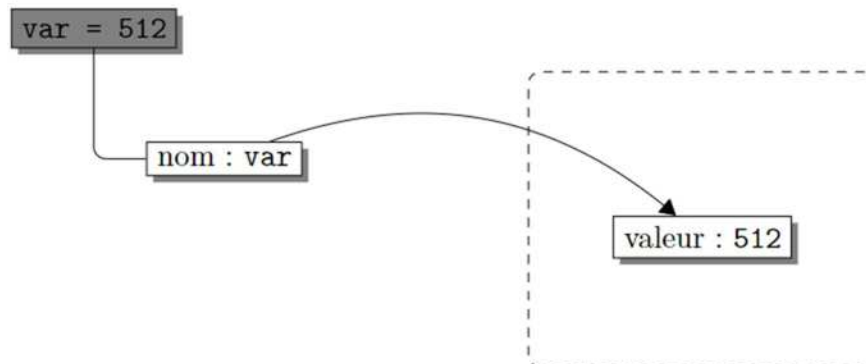
Une affectation permet aussi de modifier un item d'un objet mutable.

L'affectation `a = b` est une instruction. Ce n'est pas une expression. Cela signifie par exemple qu'un test comme `if (a=b)` provoque une erreur.

Le graphique qui suit représente une affectation `var = 512`. La valeur est stockée en mémoire.



La représentation est simplifiée :



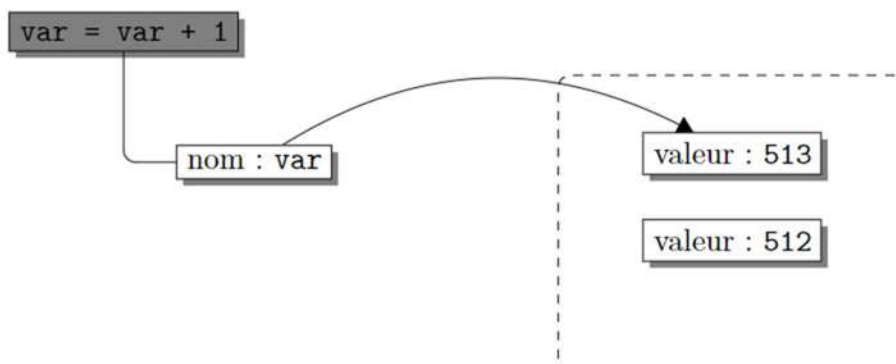
Considérons maintenant l'instruction `var = var + 1`.

Le nom `var` permet d'accéder à la valeur 512, puis d'évaluer le résultat de l'addition qui constitue la valeur de l'expression à droite du signe `=`. Cela revient donc à écrire `var = 513`.

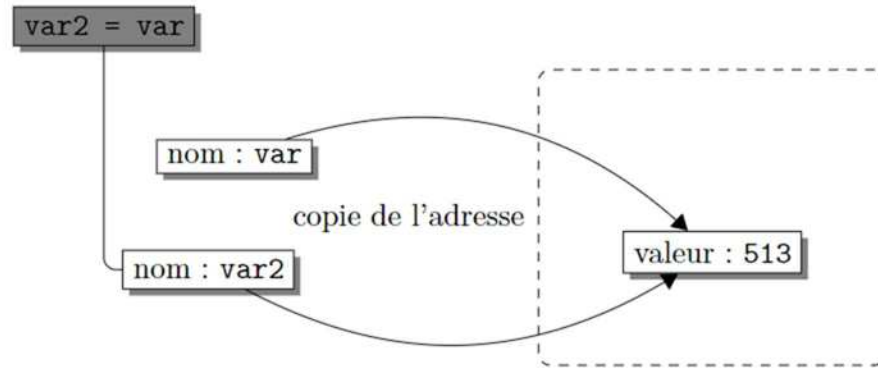
Que se passe-t-il dans la machine? Nous pouvons envisager deux manières de procéder.

Première possibilité : la valeur 513 est écrite en mémoire à la place de la valeur 512. C'est ce qui se passe avec certains langages de programmation comme le langage C.

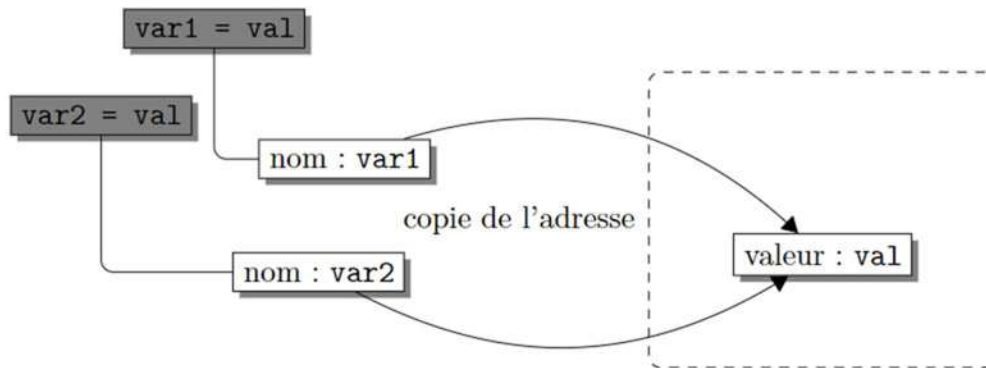
Deuxième possibilité : la valeur 513 est écrite à un autre emplacement de la mémoire. Le nom est alors délié de la valeur 512 et relié à la valeur 513. C'est ce qui se passe en Python.



Considérons la création d'une deuxième variable avec l'instruction `var2 = var`. La valeur n'est pas copiée à un autre endroit en mémoire comme c'est le cas avec certains langages.

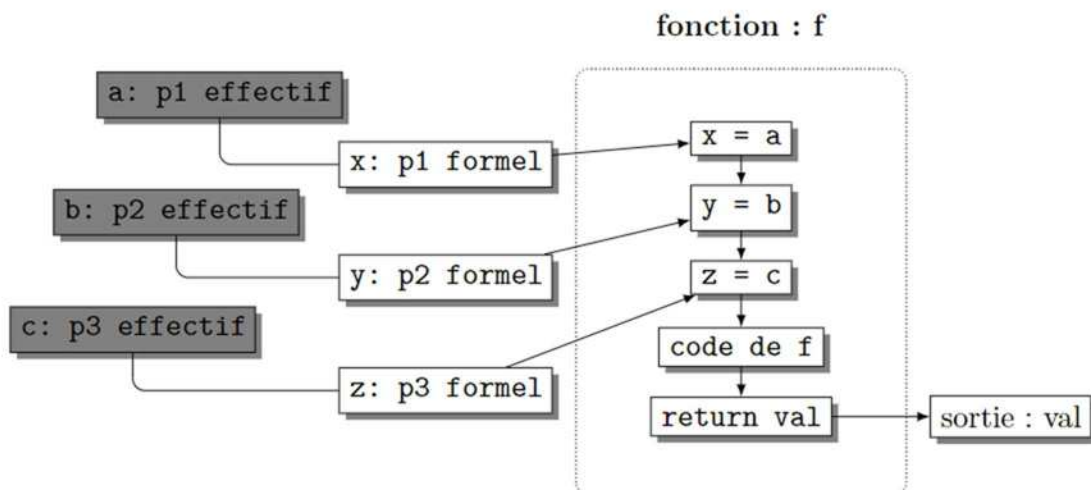


Ceci économise la copie d'une valeur qui peut être très grande comme une chaîne de caractères. Ce procédé est aussi utilisé avec les deux instructions `var1 = val` et `var2 = val`. Si `val` est la valeur d'un objet non mutable, autant que possible, une seule valeur est écrite en mémoire.



Une fonction est un morceau de programme qui a un nom et peut être utilisé autant de fois qu'on le souhaite. La fonction `f` est définie avec des paramètres formels `x`, `y`, `z`. Elle est appelée avec des paramètres effectifs `a`, `b`, `c`. L'exécution de `f` est représentée ci-dessous.

```
def f(x, y, z):
    code
    return val
f(a, b, c)
```



■ Index

- accumulateur, 6
- affectation, 187, 326
- algorithme A*, 308
- algorithme de Hörner, 222
- alphabet, 234
- arc, 273
- assertion, 193
- bibliothèque, 45
- binaire, 233
- Boole George, 234
- boucle
 - conditionnelle, 5
 - non conditionnelle, 5
- Boyer S. Robert, 31
- complément à deux, 237
- complexité, 214
- compteur, 6
- connexe, 274
- correction, 207
- coût, 214
- degré, 273
- dichotomie, 71
- dictionnaire, 11
- Dijkstra Edsger, 303
- diviser pour régner, 163
- docstring, 189
- effet de bord, 188
- entier non signé, 236
- entier signé, 237
- exposant, 256
- expression, 187, 326
- factorielle, 92
- FIFO, 295
- file, 295

- flottant, 254
- fonction, 326
- fonction récursive, 89
- glouton, 111
- graphe, 273
- heuristique, 308
- Hoare C.A.R., 166
- Holberton Betty, 159
- Horspool Nigel, 31
- instruction, 187, 326
- invariant de boucle, 208
- jeu de tests, 210
- Karatsuba Anatolii Alexevich, 241
- Knuth Donald, 67
- Lehmer D. H., 67
- lettre, 234
- LIFO, 295
- mantisse, 256
- médiane, 159
- module, 45
- Moore Strother, 31
- mot, 234
- nombres de Fibonacci, 94
- occurrence, 7
- octet, 234
- ordre d'un graphe, 273
- package, 45
- parcours en largeur d'abord, 301
- parcours en profondeur d'abord, 299
- Peters Tim, 159
- pile, 295

récurtivité, 87
recherche
 d'un extrémum, 10
 d'une occurrence, 7
 dichotomique, 67

signature, 191
sommet, 273
spécification, 189

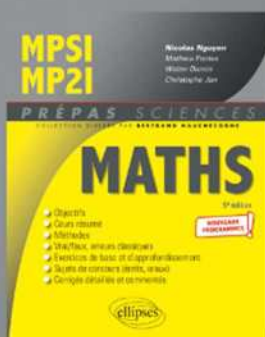
terminaison, 207
test, 209

tri
 à bulles, 28
 fusion, 163
 insertion, 161
 rapide, 166
 sélection, 159
type, 330

validité d'un algorithme, 207
variant de boucle, 207
von Neumann John, 163

Dans la même collection

NOUVEAUX
PROGRAMMES



PRÉPAS SCIENCES

Les ouvrages de la collection **PRÉPAS SCIENCES** sont le complément indispensable à la réussite en CPGE. Ils ont été conçus et rédigés par des professeurs enseignant en classes préparatoires dans différents lycées de notre pays. Leur contenu a été discuté et pensé avec soin pour permettre la meilleure adéquation avec les attentes et les besoins des étudiants.

Pour intégrer, il faut assimiler le cours, en connaître les points fondamentaux et savoir le mettre en application dans des exercices ou des problèmes souvent ardu. Cette collection est conçue pour répondre à ces difficultés. Dans chaque chapitre vous trouverez :

- **Le cours**

Il vous permettra d'accéder à une connaissance synthétique des notions et par de nombreux exemples de vous initier aux techniques usuelles qu'il faut savoir mettre en place.

- **Le vrai/faux**

Il testera votre compréhension du cours et vous évitera de tomber dans les erreurs classiques.

- **Les exercices, avec des indications**

Souvent tirés d'annales de concours, ils vous entraîneront aux écrits comme aux oraux.

- **Les corrigés**

Toujours rédigés avec soin, ils vous aideront à progresser dans la résolution d'exercices.

Ainsi, avec un seul livre par année et par matière, la collection **PRÉPAS SCIENCES** vous permettra de surmonter les colles, d'affronter les devoirs, et elle vous guidera, jour après jour, dans votre cheminement vers la **réussite aux concours**.

www.editions-ellipses.fr

