NOUVEAUX PROGRAMMES 2021

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

PRÉPAS 1re ANNÉE SCIENTIFIQUES

j'intègre

EXERCICES INCONTOURNABLES

- Les exercices incontournables du programme
- Les méthodes de résolution étape par étape
- Les erreurs à éviter
- Les corrigés détaillés

DUNOD

INFORMATIQUE AVEC PYTHON

PRÉPAS 1re ANNÉE SCIENTIFIQUES

Chez le même éditeur

Python 3
2e édition
Bob Cordeau, Laurent Pointal
304 pages
Dunod, 2020

Informatique
Joëlle Delacroix et al.
480 pages
Dunod, 2017

Algorithmes
Thomas H. Cormen
240 pages
Dunod, 2013

Programmation Python avancée
Xavier Olive
368 pages
Dunod, 2021

Python pour le data scientist
2e édition
Emmanuel Jakobowicz
320 pages
Dunod, 2021

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

PRÉPAS 1re ANNÉE SCIENTIFIQUES

EXERCICES INCONTOURNABLES

j'intègre

DUNOD

Conception graphique de la couverture : Hokus Pokus Créations

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-083134-0

Table des matières

Vous pouvez télécharger à partir de la page de présentation de l'ouvrage sur le site Dunod tous les programmes Python des exercices. Des fichiers complémentaires sont également fournis afin de tester les programmes, par exemple des images pour les exercices du chapitre « Traitement d'images ».

https://dunod.com/EAN/9782100826063

Avant-propos

Cet ouvrage de la série « Exercices incontournables » traite de l'intégralité du nouveau programme d'informatique commun pour la première année des différentes filières de classes préparatoires aux grandes écoles en vigueur à partir de septembre 2021.

Il sera suivi en juin 2022 d'un deuxième ouvrage qui intégrera le nouveau programme de 2e année.

La première partie reprend la base de la programmation avec Python. Des rappels de cours et des exercices classiques vous permettront de vous familiariser avec la syntaxe Python.

Dans le premier exercice de certains chapitres (« Terminaison-correction-complexité », « Graphes »), vous trouverez un rappel de cours détaillé présentant le vocabulaire utilisé.

Pour chaque exercice classique, vous trouverez :

- La méthode de résolution expliquée et commentée étape par étape.
- Le corrigé rédigé détaillé.
- Les astuces à retenir et les pièges à éviter.

Partie 1

Prise en main de Python

Plan

1

Prise en main de Python

Exercice 1.1 : Assertion, moyenne, variance et écart-type d'une liste de nombres

On considère la liste de nombres : `L = [9, 10, 11, 20.5, 0, 12.0, -5, -8.3e1]`.

1. Écrire une fonction `rec_moy` qui admet comme argument `L` une liste non vide de nombres et retourne la moyenne de `L`.

2. Écrire une fonction `rec_variance` qui admet comme argument `L` une liste non vide de nombres et retourne la variance de la liste `L`.

3. Écrire une fonction `rec_ecart_type` qui admet comme argument `L` une liste non vide de nombres et retourne l'écart-type de la liste `L`.

4. Les en-têtes des fonctions peuvent être annotés pour préciser les types des paramètres et du résultat. Ainsi,

```
def uneFonction(n:int, X:[float], c:str, u) -> np.ndarray:
```

signifie que la fonction `uneFonction` prend quatre paramètres `n`, `X`, `c` et `u`, où `n` est un entier, `X` une liste de nombres à virgule flottante et `c` une chaîne de caractères ; le type de `u` n'est pas précisé. Cette fonction renvoie un tableau `numpy`.

Écrire une fonction `rec_moy2` qui admet comme argument `L` une liste non vide de nombres réels ou flottants et retourne la moyenne de la liste `L`. Annoter l'en-tête de la fonction pour préciser le type des données attendues en entrée et fournies en retour. Utiliser des assertions pour vérifier le type de `L`, le nombre d'éléments de `L` ainsi que le type des éléments de `L`.

Analyse du problème

On utilise une boucle `for` pour parcourir les différents éléments de la liste. On peut alors calculer la somme des éléments de la liste pour en déduire la valeur moyenne.

Une assertion est une aide de détection de bugs dans les programmes. La levée d'une assertion entraîne l'arrêt du programme.

Cours :
L'installation de Python 3 peut se faire très facilement avec la suite Anaconda 3. L'éditeur Spyder s'installe automatiquement avec Anaconda.

Un script Python est formé d'une suite d'instructions. Une instruction simple est contenue dans une seule ligne. Si une instruction est trop longue pour tenir sur une ligne ou si on souhaite améliorer la lisibilité du code, le symbole « \ » en fin de ligne permet de poursuivre l'écriture de l'instruction sur la ligne suivante (voir corrigé 4).

Une affectation se fait avec l'instruction `n = 3` : *n* prend la valeur 3 (voir exercice 2.2 « Affectation, objet immuable, copie de listes et de tableaux » dans le chapitre « Tableaux numpy – Slicing »).

On peut utiliser « _ » dans le nom des variables mais pas « - ».

Le symbole dièse permet d'ajouter des annotations dans les programmes Python : `# commentaire sur le programme`.

Le type d'une variable `n` s'obtient avec l'instruction : `type(n)`.

Les types de base des variables dans Python sont :

- Nombres entiers (positifs ou négatifs) : `int`
- Nombres à virgule flottante (ou nombres flottants) : `float`

  Exemple
  `a = -3.2e2` : $-3{,}2 \times 10^2 = -320$
- Nombres complexes : `complex`

  Exemple
  ```
  a = 3j
  b = 1j
  ```
  Les nombres complexes sont écrits avec un « `j` » pour la partie imaginaire.
- Booléens : `bool`
  Les variables booléennes sont `True` (vrai) et `False` (faux).

Opérations de base : `+`, `-`, `*`, `/`

`n // 10` : quotient de la division euclidienne de *n* par 10
`n % 10` : reste de la division euclidienne de *n* par 10
`n ** 3` : *n* puissance 3

Opérateurs :

`a == b` : cet opérateur compare *a* et *b*. Si *a* = *b*, Python retourne `True`, sinon `False`.
`a != b` : *a* différent de *b*
`a > b`, `a < b`, `a >= b`, `a <= b` : strictement supérieur, strictement inférieur, supérieur ou égal, inférieur ou égal
`or` : ou
`and` : et
`not` : non

Les types de base des conteneurs dans Python sont :

- Chaînes de caractères : `str`
  `s = "C'est une phrase"` : utilisation de guillemets `"`
  `s = 'phrase'` : utilisation d'apostrophes `'`

`s[0]` : affiche `'p'`. Le premier caractère de `s` a pour indice 0.
`s[-1]` : affiche le dernier caractère : `'e'`. On pourrait écrire également `s[5]`.
Pour extraire des caractères, voir exercice 2.4 « Slicing, tranche, `range`, `np.arange`, `np.linspace` » dans le chapitre « Tableaux numpy – Slicing ».

- Listes : `list`
  `L = []` : crée une liste vide
  `L.append(3)` : ajoute 3 à la fin de la liste `L`. On obtient `L = [3]`.
  `L.append(2)` : ajoute 2 à la fin de la liste `L`. On obtient `L = [3, 2]`.
  `x = L.pop()` : supprime le dernier élément (ici 2) de la liste `L`. On récupère cet élément dans la variable *x*.
  `x=L2.pop(i)` : supprime l'élément situé à l'indice *i* de la liste `L2`. On récupère cet élément dans la variable *x*.
  `L2.insert(i, x)` : insère l'élément *x* à l'indice *i* de la liste `L2`
  `L3 = ['r', 3, 'te']` : crée la liste `L3` contenant des chaînes de caractères et des entiers
  `len(L3)` : affiche la longueur de la liste `L3`
  Pour extraire des éléments d'une liste, voir exercice 2.4 « Slicing, tranche, `range`, `np.arange`, `np.linspace` » dans le chapitre « Tableaux numpy – Slicing ».
  `L=[2*i for i in range(5)]` : on obtient [0, 2, 4, 6, 8] (création par compréhension)
  `L=[2,3]*3` : répétition de la liste [2, 3]. On a alors `L`=[2, 3, 2, 3, 2, 3].

- Tuples : `tuple`
  Une fois le tuple créé, il ne peut pas être modifié. On peut créer un tuple avec ou sans parenthèses.
  `M = (2, 3, 8)` : crée le tuple `M`
  Ne pas confondre avec les listes, où on met des crochets.
  On crée le même tuple si on omet les parenthèses : `M = 2, 3, 9`
  `x = M[0]` : récupère dans la variable *x* l'élément d'indice 0 du tuple `M`
  `a, b, c = M` : dépaquette un tuple. On récupère dans chaque variable un élément du tuple. Il faut connaître à l'avance le nombre d'éléments du tuple.

- Deques : `deque`
  Une deque (se prononce « dèque ») permet d'ajouter et de supprimer très rapidement des éléments aux extrémités droite et gauche (voir exercice 12.8 « Utilisation de deques » dans le chapitre « Dictionnaire – Pile – File – Deque »).
  `from collections import deque` : module permettant d'utiliser les deques
  `D=deque()` : création d'une deque vide `D`
  `D.append(3)` : ajoute 3 à l'extrémité droite de `D`
  `D.appendleft(5)` : ajoute 5 à l'extrémité gauche de `D`
  `x=D.pop()` : supprime l'élément à l'extrémité droite de `D`
  `x=D.popleft()` : supprime l'élément à l'extrémité gauche de `D`

- Dictionnaires : `dict`
  Pour l'utilisation des dictionnaires, voir exercice 12.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire – Pile – File – Deque ».

- Tableaux numpy
  Voir exercice 2.1 « Manipulations de listes et de tableaux numpy » dans le chapitre « Tableaux numpy – Slicing ».

Remarque : Voir la remarque de la question 2 dans l'exercice 11.1 « Tri par insertion » dans le chapitre « Tris » pour le type `None`. On ne l'utilisera pas dans les autres exercices.

Cours :
Il existe deux types d'objets dans Python :

- les objets dont la valeur peut changer sont dits mutables (ou muables) : listes, dictionnaires, deques (voir chapitre 12 « Dictionnaire – Pile – File – Deque »), tableaux numpy… ;
- les objets dont la valeur ne peut pas changer sont dits immutables (ou immuables) : entiers, flottants, complexes, booléens, chaînes de caractères, tuples…

Voir exercices 2.2 « Affectation, objet immuable, copie de listes et de tableaux » et 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing » pour les affectations et les arguments d'entrée des fonctions.

Quelques fonctions intrinsèques :

`abs(x)` : renvoie la valeur absolue de *x*
`int(x)` : convertit *x* en entier
`float(x)` : convertit *x* en flottant
`str(x)` : convertit *x* en chaîne de caractères
`bool(x)` : convertit *x* en booléen

Première utilisation de la boucle `for` :

```
x=5                 # affecte 5 à la variable x
for i in range(n):  # boucle faisant varier i de 0 inclus à n exclu
                    # ne pas oublier ':' à la fin de la ligne
    x = x+i         # incrémente x de i à chaque passage dans la boucle
                    # attention à l'indentation
```

Deuxième utilisation de la boucle `for` :

```
for elt in L:       # elt prendra successivement les éléments de L
    print(elt)      # L chaîne de caractères, liste, tuple, deque
                    # ou dictionnaire
```

Pour l'utilisation de la boucle `while`, ne pas oublier :

```
i = 0                     # initialisation de la variable i à 0
while i<3 and rep=True:   # boucle exécutée tant que i < 3 et que rep = True
    x = x+i               # attention à l'indentation
    i = i + 1             # incrémente i de 1 à chaque étape de la boucle
```

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante (voir exercice 11.8 « Tri à bulles » dans le chapitre « Tris »).

Définition d'une fonction :

```
def f(x):     # définition de la fonction f ayant pour argument d'entrée x
              # ne pas oublier ':' à la fin de la ligne
    y = x+3   # y est une variable locale : elle est créée à l'appel
              # de la fonction et est détruite à la fin de la fonction
              # voir exercice 1.3 "Variables locales, variables globales"
    return y  # fin de la fonction et retourne la valeur y
              # attention à l'indentation
```

L'instruction `return` quitte la fonction même en cours d'exécution d'une boucle `for` ou `while`.

Structure conditionnelle :

```
if x == 3:           # teste si x = 3
    y=3*x
elif x>3 and x<=4:   # si le test précédent n'est pas vérifié,
                     # alors teste si x >3 et si x <=4
    y=x+2
elif x>4 and x <5:   # si le test précédent n'est pas vérifié,
                     # alors teste si x >4 et si x <5
    y=x-2
else:                # sinon (les tests précédents ne sont pas vérifiés)
    y=0
```

La bibliothèque math permet d'accéder à de nombreuses fonctions mathématiques.

On peut l'importer à l'aide de la commande :

```
import math   # import de la bibliothèque math
```

On pourra utiliser les fonctions trigonométriques : `math.cos`, `math.sin`, `math.tan`, `math.asin`, `math.acos`, `math.atan` ; les constantes `math.pi`, `math.e` ; les fonctions `math.exp(x)`, `math.log(x)` logarithme népérien, `math.log10(x)` logarithme en base 10…

On peut aussi importer la bibliothèque math avec la commande :

```
from math import *
```

Dans ce cas, on peut utiliser les fonctions précédentes sans ajouter `math`.

La valeur π s'obtient en tapant directement `pi` dans Python.

On peut importer une seule fonction de la bibliothèque math avec la commande :

```
from math import cos
from math import pi
print(cos(pi/2))       # Python affiche 6.123233995736766e-17
```

Les nombres flottants ne permettent pas un calcul exact à cause de la représentation des nombres à virgule avec la norme IEEE-754. Un test du type `a==b` n'a en général pas de sens si `a` et `b` sont des nombres à virgule flottante. On remplacera donc ce test par : `abs(a-b)<epsilon` où `epsilon` est une valeur proche de zéro, choisie en fonction du problème à traiter et de l'ordre de grandeur des erreurs auxquelles on peut s'attendre sur `a` et `b`.

1. On considère une liste non vide `L`. On suppose implicitement que les éléments de la liste sont des nombres entiers ou flottants et qu'elle ne contient pas de nombre complexe. On définit une variable `S` permettant de calculer la somme des éléments de la liste.

```
def rec_moy(L):
    # la fonction retourne la moyenne de la liste L
    S=0                      # initialisation de S à 0
    n=len(L)                 # longueur de la liste L
```

```
    for i in range(n):        # i varie entre 0 inclus et n exclu
        S=S+L[i]              # on pourrait écrire S+=L[i]
    moyenne=S/n               # calcul de la moyenne
    return moyenne            # retourne la moyenne de L
```

Remarques :

- La fonction `sum(L)` permet de calculer la liste des éléments d'une liste `L`.

```
print('Somme des éléments de L :', sum(L))
```

- La liste `L = [9, 10, 11, 20.5, 0, 12.0, -5, -8.3e1]` contient des entiers (`9`, `10`, `11`, `0`, `5`) ainsi que des flottants : `20.5`, `12.0`, `-8.3e1`.

  `12.0` est un nombre flottant (type `float`) alors que `12` est un nombre entier (type `int`).

  `-8.3e1` est un nombre flottant alors que `83` est un nombre entier.

Cours :

Soit une liste de valeurs $X_1, X_2 \ldots X_N$. La moyenne des valeurs est définie par : $\langle X \rangle = \frac{1}{N}\sum_{i=1}^{N} X_i$.

La variance (ou écart quadratique moyen) est définie par : $(X) = \langle X^2 \rangle - \langle X \rangle^2$ avec $\langle X^2 \rangle = \frac{1}{N}\sum_{i=1}^{N} X_i^2$. L'écart-type est défini par $\Delta X = \sqrt{\text{var}(X)}$.

2.

```
def rec_variance(L):
    # la fonction retourne la variance de la liste L
    S=0                       # initialisation de S à 0
    n=len(L)                  # longueur de la liste L
    for i in range(n):        # i varie entre 0 inclus et n exclu
        S=S+L[i]**2           # on pourrait écrire S+=L[i]**2
    variance=S/n-rec_moy(L)**2
    return variance           # retourne la variance de L
```

3.

```
def rec_ecart_type (L):
    # la fonction retourne l'écart-type de la liste L
    import math       # import de la bibliothèque math
    return(math.sqrt(rec_variance(L)))
```

Remarques :

- La fonction `np.mean(L)` de Python (voir exercice 2.1 « Manipulations de listes et de tableaux numpy » dans le chapitre « Tableaux numpy – Slicing ») retourne la moyenne d'une liste de valeurs en important la bibliothèque numpy :

```
import numpy as np    # bibliothèque numpy renommée np
```

• La fonction `np.var(L)` de Python retourne la variance d'une liste de valeurs.

```
print('moyenne :', np.mean(L))
print('variance :', np.var(L))
```

Cours :
Une assertion est une aide de détection de bugs dans les programmes.

• La fonction `rec_moy` peut être appelée si le type de `L` est `list`. On ajoute la ligne suivante dans la fonction `def rec_moy2` :

```
assert type(L)==list
```

Le programme teste si `type(L)==list`. Si la condition est vérifiée, le programme continue à s'exécuter normalement.
Si la condition `type(L)==list` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
assert type(L)==list AssertionError
```

• La fonction `rec_moy` peut être appelée si le nombre d'éléments de `L` est strictement positif. On ajoute la ligne suivante dans la fonction `def rec_moy2` :

```
assert len(L)>0
```

Le programme teste si `len(L)>0`, sinon on aurait une division par 0 dans la fonction. Si la condition est vérifiée, le programme continue à s'exécuter normalement. Si la condition `len(L)>0` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
assert len(L)>0 AssertionError
```

On supprime les assertions dans la version finale du programme Python.

4.

```
def rec_moy2(L:list)->float:
    # la fonction retourne la moyenne (float) des éléments
    # de la liste L
    # On pourrait écrire : def rec_moy2(L:[float])->float:
    # L est une liste de nombres flottants
    assert type(L)==list
    assert len(L)>0
    S=0                 # initialisation de S à 0
    n=len(L)            # longueur de la liste L
    for i in range(n):  # i varie entre 0 inclus et n exclu
        assert type(L[i])==float or type(L[i])==int
        S=S+L[i]        # on pourrait écrire S+=L[i]
    moyenne=S/n         # calcul de la moyenne
    return moyenne      # retourne la moyenne de L
```

Si on exécute l'instruction `print('moyenne :',rec_moy2(3))`, Python affiche :

```
assert type(L) == list AssertionError
```

Si on exécute l'instruction `print('moyenne :',rec_moy2([]))`, Python affiche :

```
assert len(L)>0 AssertionError
```

> Si on exécute l'instruction `print('moyenne :',rec_moy2([3, 4.0, 'a']))`, Python affiche :
>
> ```
> assert type(L[i])==float or type(L[i])==int
> AssertionError
> ```

Remarques :

On peut ajouter une chaîne de caractères dans l'instruction `assert` :

```
assert type(L)==list, 'Le type de L doit être une liste.'
```

Le programme teste si `type(L)==list`. Si la condition est vérifiée, le programme continue à s'exécuter normalement.

Si la condition `type(L)==list` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
Le type de L doit être une liste.
```

Si une instruction est trop longue pour tenir sur une ligne ou si on souhaite améliorer la lisibilité du code, on peut utiliser le symbole '\' en fin de ligne et poursuivre l'écriture de l'instruction sur la ligne suivante.

On peut écrire :

```
assert type(L[i])==float or type(L[i])==int, "Type de L[i] non correct."
```

ou

```
assert type(L[i])==float or type(L[i])==int, \
        "Type de L[i] non correct."
```

On peut ajouter des commentaires dans les programmes Python avec le symbole dièse. Pour ajouter un commentaire qui s'étend sur plusieurs lignes, on peut le commencer avec `'''` (trois apostrophes) ou `"""` (trois guillemets) et le terminer de la même façon :

```
'''
la fonction retourne la moyenne (float) des éléments de la liste L
On pourrait écrire : def rec_moy2(L:[float])->float:
'''
```

Les assertions servent à tester des conditions critiques qui ne devraient jamais arriver. Ce sont des aides au développement des programmes.

Si ces erreurs (liste vide, type de `L` incorrect, type de `L[i]` incorrect) sont susceptibles d'arriver lors de l'exécution du programme final, alors il faut utiliser le test `if len(L)==0` et gérer par programmation l'erreur.

Exercice 1.2 : Boucles, test, fonctions (banque PT 2015)

1. Soit l'entier $n = 1\,234$. Quel est le quotient, noté q, de la division euclidienne de n par 10 ? Quel est le reste ? Que se passe-t-il si on recommence la division euclidienne par 10 à partir de q ?

Écrire une fonction `calcul_base10` d'argument `n`, renvoyant une liste `L` contenant les restes des divisions euclidiennes successives.

Écrire le programme principal demandant à l'utilisateur de saisir un entier `n` strictement positif et renvoyant la décomposition en base 10 de l'entier `n`.

2. Écrire une fonction `somcube`, d'argument `n`, renvoyant la somme des cubes des chiffres du nombre entier `n`. On pourra utiliser la fonction `calcul_base10`.

3. Écrire une fonction permettant de trouver tous les nombres entiers strictement inférieurs à 1 000 égaux à la somme des cubes de leurs chiffres.

4. Écrire une fonction `somcube2` qui convertit l'entier *n* en une chaîne de caractères permettant ainsi la récupération de ses chiffres sous forme de caractères. Cette nouvelle fonction renvoie la chaîne de caractères ainsi que la somme des cubes des chiffres de l'entier *n*. On pourra utiliser la fonction `str` et manipuler les chaînes de caractères.

Analyse du problème

Cet exercice est extrait du sujet de concours 0 de la banque PT 2015. Il permet de s'entraîner à manipuler les fonctions, les boucles, les tests et les différents types rencontrés dans Python 3.

Dans Python 3, 4/3 renvoie 1.333 alors que, dans Python 2, 4/3 renvoie 1, c'est-à-dire le quotient de la division euclidienne de 4 par 3 ! Les programmes seront réalisés exclusivement avec Python 3 dans cet ouvrage.

1. $n = 1\,234 = 123 \times 10 + 4$. Le reste vaut 4.

Si on recommence la division euclidienne de 123 par 10 : $123 = 12 \times 10 + 3$. Le reste vaut 3.

Si on recommence la division euclidienne de 12 par 10 : $12 = 1 \times 10 + 2$. Le reste vaut 2.

Si on recommence la division euclidienne de 1 par 10 : $1 = 0 \times 10 + 1$. Le reste vaut 1.

On obtient la décomposition en base 10 de n : 1, 2, 3, 4.

```
def calcul_base10(n):
    # la fonction renvoie une liste contenant les restes
    # des divisions euclidiennes successives
    L=[]   # création d'une liste vide
    while n > 0: # boucle tant que n > 0
        q=n//10  # quotient de la division euclidienne de n par 10
        r=n%10   # reste de la division euclidienne de n par 10
        L.append(r)   # on ajoute le reste dans la liste L
        n=q
    L.reverse()  # inverse l'ordre des éléments de la liste L
    return L     # retourne la liste L
```

```
# programme principal
n=int(input('Taper un entier strictement positif : '))
# conversion en entier du résultat de la saisie
print('Décomposition en base 10 : ',calcul_base10(n))
```

2.

```
def somcube(n):
    # la fonction renvoie la somme des cubes des chiffres
    # de l'entier n
    somme=0   # initialisation de la variable somme
    L=calcul_base10(n)   # récupère la liste donnant
                         # la décomposition en base 10 de n
    for i in range(len(L)):   # i varie entre 0 inclus
                              # et len(L) exclu
        somme=somme+L[i]**3   # on ajoute L[i] à la puissance 3
                              # on peut écrire somme+=L[i]**3
    return somme

# programme principal
n=1234
print(somcube(n))   # affiche 100 pour n = 1234
```

3.

```
def affiche_liste_entier_cube():   # pas d'argument d'entrée
    # la fonction renvoie tous les nombres entiers strictement
    # inférieurs à 1000 égaux à la somme des cubes
    # de leurs chiffres
    L=[]   # création d'une liste vide
    for i in range(1000):   # i varie entre 0 inclus
                            # et 1000 exclu
        if i==somcube(i):   # teste si i est égal à la somme
                            # des cubes de ses chiffres
            L.append(i)     # ajoute i dans la liste L
    return L   # fin de la fonction et renvoie la liste L

# programme principal
print(affiche_liste_entier_cube())   # affiche
                                     # [0, 1, 153, 370, 371, 407]
```

4.

```
def somcube2(n):
    # la fonction convertit l'entier n en une chaîne de caractères
    # pour récupérer ses chiffres sous forme de caractères
    somme=0
    chaine=str(n)   # convertit n en une chaîne de caractères
    L=[]            # création d'une liste vide
    for elt in chaine:  # elt prend successivement
                        # les éléments de chaine
        L.append(elt)   # elt est un caractère que l'on ajoute
                        # dans L
        somme=somme+(int(elt))**3   # il faut convertir elt
                                    # en entier
    return L,somme      # on pourrait écrire return(L, somme)
```

```
# programme principal
n=int(input('Taper un entier strictement positif : '))
L1,res=somcube2(n)    # L1 contient la liste des chiffres de n
                      # res = somme des cubes des chiffres de n
```

Remarque :

La fonction `somcube2(n)` renvoie un tuple contenant deux éléments. Pour récupérer les éléments de ce tuple, on a plusieurs possibilités :

- Dépaquetage d'un tuple :

  La ligne `return L, somme` retourne un tuple : `(L, somme)`.

  Pour récupérer dans des variables séparées les éléments du tuple, on peut écrire :

```
L1, res=somcube2(25)
```

  On obtient alors : `L1=['2', '5']` et `res=133`.

- On définit un tuple *A* :

```
A=somcube2(25)
```

  Le tuple *A* vaut : `(['2', '5'], 133)`.

  Pour récupérer `['2', '5']`, le premier élément du tuple : `A[0]`.

  Pour récupérer `133`, le deuxième élément du tuple : `A[1]`.

Exercice 1.3 : Variables locales, variables globales

On considère le programme suivant :

```
def f() :
    global b
    print('d =', d)
    print('Premier print dans la fonction f : b=', b)
    a=3
    c=5
    b=b+c
    print('Deuxième print dans la fonction f : b=', b)
    print('Troisième print dans la fonction f : a=', a)
    return

a=2
b=2
d=3
print("Print avant l'appel de la fonction f : a=", a)
print("Print avant l'appel de la fonction f: b=", b)
f()
print("Print après l'appel de la fonction f : a=", a)
print("Print après l'appel de la fonction f : b=", b)
```

Qu'affiche Python lors de l'exécution du programme ? Analyser les différents affichages de `print`.

Analyse du problème

Ce programme permet de comprendre la différence entre les variables globales et les variables locales dans une fonction.

Cours :

Une variable locale est créée au début d'une fonction et est détruite lorsque la fonction est terminée. Elle existe uniquement dans le corps de la fonction.

Une variable globale est définie à l'extérieur d'une fonction. Le contenu de cette variable est visible à l'intérieur d'une fonction. L'instruction `global b` permet de définir la variable globale `b` dans la fonction `f`.

Le programme Python affiche :

```
Print avant l'appel de la fonction f : a= 2
Print avant l'appel de la fonction f : b= 2
d = 3
Premier print dans la fonction f : b= 2
Deuxième print dans la fonction f : b= 7
Troisième print dans la fonction f : a= 3
Print après l'appel de la fonction f : a= 2
Print après l'appel de la fonction f : b= 7
```

La variable `a` vaut toujours 2 après l'exécution de la fonction `f`.

Dans le corps de la fonction `f`, `a` est une variable locale qui n'a rien à voir avec la variable `a` définie dans le programme principal.

La variable `b` est modifiée par la fonction `f` car `b` est une variable globale (instruction `global b`). On retrouve 7 après l'appel de la fonction `f`.

La variable `c` est une variable locale. Elle n'est pas définie en dehors de la fonction. La syntaxe `print(c)` en dehors de la fonction entraîne un message d'erreur de Python.

La variable `d` n'est pas définie dans la fonction `f`. Python cherche alors la valeur de `d` dans le programme principal. Python affiche alors « `d = 3` ».

Remarque : L'instruction `global i,j` permet de désigner deux variables globales `i` et `j` dans une fonction.

Tableaux numpy – Slicing

2

Exercice 2.1 : Manipulations de listes et de tableaux numpy

On considère la fonction *f* définie par : $f(x) = \sin(x) - x$.

1. Définir la fonction *f* dans Python en utilisant la bibliothèque numpy.

2. Définir une liste `X1` contenant 20 valeurs régulièrement espacées entre 10 et 30. Définir une liste `Y1` contenant 20 valeurs définies par $Y1[i] = f(X1[i])$. On utilisera l'instruction `range`.

3. Définir un tableau numpy `X2` contenant 20 valeurs régulièrement espacées entre 10 et 30. Définir un tableau `Y2` contenant 20 valeurs définies par $Y[i] = f(X[i])$. On utilisera l'instruction `range`.

4. Définir un tableau numpy `X3` contenant 20 valeurs régulièrement espacées entre 10 et 30 en utilisant la fonction `np.linspace`. Définir un tableau `Y3` contenant 11 valeurs définies par $Y[i] = f(X[i])$ sans utiliser la fonction `range`.

Analyse du problème

On va étudier dans cet exercice la différence entre les listes et les tableaux numpy. Il faut savoir manipuler aussi bien les listes que les tableaux numpy.

Voir exercice 10.1 « Traitement d'images et filtrage passe-bas » du chapitre « Traitement d'images » pour l'utilisation des matrices.

Cours :
Manipulation de listes avec Python

```
L1=[]                  # création d'une liste vide
L2=[1, 2, 'a', 'b']    # création d'une liste avec 4 éléments
L2.append(3)           # on ajoute 3 dans la liste L2
                       # on a alors L2=[1, 2, 'a', 'b', 3]
len(L2)                # renvoie la longueur de la liste L2 : 5
x=L2.pop()    # efface le dernier élément de la liste et retourne sa valeur.
              # On obtient L2=[1, 2, 'a', 'b']
L3=[2, 5, 3, 1]        # création de la liste L3
L3.sort()              # cette fonction ne retourne rien.
                       # Elle trie en place la liste L3.
                       # On obtient alors L3=[1, 2, 3, 5]
2 in L3       # détermine si 2 est dans la liste L3 : renvoie True ici
```

```
8 not in L3  # détermine si 8 n'est pas dans la liste L3 : renvoie True ici
L=list(range(4))      # renvoie une liste contenant les 4 premiers entiers
                      # dans l'ordre croissant : [0, 1, 2, 3]
L=[2*i for i in range(5)]   # on obtient : [0, 2, 4, 6, 8].
                            # Création par compréhension
```

Une liste peut contenir des éléments avec des types différents.

On peut définir des listes de listes :

```
L=[[2,3,5],[1,8,7]]   # L est une liste de deux éléments.
                      # Chaque élément est une liste.
L[0]      # renvoie l'élément d'indice 0 de la liste L : [2, 3, 5]
L[0][1]   # renvoie l'élément d'indice 1 de L[0] : 3
```

La liste `L` peut représenter une matrice contenant 2 lignes et 3 colonnes. Les tableaux numpy étudiés dans le paragraphe suivant sont plus adaptés pour la manipulation de matrices.

Soit une liste de *n* éléments. Dans Python, le premier élément de la liste a pour indice 0 et le dernier élément a pour indice *n*–1. En revanche, dans le logiciel Scilab, le premier élément de la liste a pour indice 1 et le dernier élément a pour indice *n*. On utilisera uniquement Python par la suite.

Cours :
Manipulation de tableaux numpy

La bibliothèque numpy permet d'effectuer du calcul matriciel avec une structure de tableaux, de vecteurs et de matrices. Les calculs sont beaucoup plus rapides qu'avec des listes car on peut avoir des types différents dans une liste. Tous les éléments d'un tableau numpy doivent avoir le même type. Par défaut, le type des éléments est `float`.

On peut utiliser le terme *tableau* pour désigner un tableau numpy de type np.ndarray.

La bibliothèque numpy est importée dans Python avec l'instruction :

```
import numpy as np  # bibliothèque numpy renommée np
```

On peut alors utiliser les fonctions de la bibliothèque numpy :

- `np.array(liste)`
  Argument d'entrée : une liste définissant un tableau à une dimension (vecteur) ou à deux dimensions (matrice).
  Argument de sortie : un tableau (matrice).
  Description : cette fonction permet de créer une matrice (de type tableau) à partir d'une liste.

```
L1=np.array([1, 2, 3])   # retourne [1, 2, 3].
                    # Cette fonction transforme la liste en tableau numpy.
                    # L1 est un tableau à une dimension contenant 3 valeurs
L1[0]     # retourne l'élément 0 du tableau, c'est-à-dire la valeur 1
len(L1)   # retourne le nombre d'éléments du tableau
          # pour un tableau à une dimension

L2=np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
     # L2 est un tableau à deux dimensions contenant 2 lignes et 4 colonnes
```

```
len(L2)           # retourne le nombre de lignes du tableau L2
np.shape(L2)      # retourne un tuple : (2,4), soit 2 lignes et 4 colonnes
np.shape(L2)[0]   # retourne le nombre de lignes du tableau
np.shape(L2)[1]   # retourne le nombre de colonnes du tableau
L2[i, j]          # retourne l'élément de la ligne i et de la colonne j
                  # du tableau L2
L2[0,0]           # retourne 1
L2[0,1]           # retourne 2
L2[1,3]           # retourne 8
L2[0,:]           # retourne la ligne : [1, 2, 3, 4]
L2[:,1]           # retourne la colonne : [2, 6]
```

- `np.zeros((n,m))`
  Argument d'entrée : un tuple de deux entiers pour créer un tableau à deux dimensions.
  Argument de sortie : un tableau (matrice) d'éléments nuls.
  Description : fonction créant une matrice (tableau) de taille $n \times m$ dont tous les éléments sont nuls.

```
np.zeros(3)        # crée le tableau [0, 0, 0]
np.zeros((2, 4))   # crée la matrice contenant 2 lignes et 4 colonnes.
                   # Toutes les valeurs sont nulles
                   #        array([[ 0.,  0.,  0.,  0.],
                   #               [ 0.,  0.,  0.,  0.]])
np.ones((2,4))     # crée la matrice contenant 2 lignes et 4 colonnes.
                   # Toutes les valeurs sont égales à 1
np.eye(3)          # crée la matrice 3×3 avec des 1 sur la diagonale
                   # et des zéros ailleurs
```

- `np.linspace(Min,Max,nbElements)`
  Argument d'entrée : un nombre flottant `Min`, un nombre flottant `Max` et un entier `nbElements`.
  Argument de sortie : un tableau (vecteur).
  Description : fonction créant un vecteur (tableau) de *nbElements* nombres espacés régulièrement entre *Min* et *Max*. Le premier élément est égal à *Min*, le dernier est égal à *Max* et les éléments sont espacés de $\frac{Max - Min}{nbElements - 1}$.

```
np.linspace(1.5, 3, 4)   # la fonction retourne : [1.5, 2, 2.5, 3]
np.arange(0, 10)   # la fonction retourne : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Voir exercice 2.4 « Slicing, tranche, `range`, `np.arange`, `np.linspace` » pour avoir plus d'explications sur la fonction `np.arange`.

```
np.mean(L)     # moyenne des éléments de L
np.var(L)      # variance des éléments de L
```

Lorsqu'on manipule des tableaux numpy, il faut connaître à l'avance le nombre d'éléments. Ce n'est pas nécessairement le cas avec les listes.

1.

```
import numpy as np   # bibliothèque numpy renommée np
def f(x):
    # la fonction retourne sin(x)-x
    return np.sin(x)-x
```

2.

```
N=20    # nombre de points
X1=[]   # création de la liste vide X1
Y1=[]   # création de la liste vide Y1
xmin=10
xmax=30
pas=(xmax-xmin)/(N-1)   # intervalle entre deux valeurs de x
for i in range(N):      # i varie entre 0 inclus et N exclu
    X1.append(xmin+i*pas)
    Y1.append(f(X1[i]))
```

3.

```
X2=np.zeros(N)   # création du tableau X2
Y2=np.zeros(N)   # création du tableau Y2
                 # comportant N zéros
for i in range(N):   # i varie entre 0 inclus et N exclu
    X2[i]=xmin+i*pas
    Y2[i]=f(X2[i])
```

4.

```
X3=np.linspace(xmin, xmax, N)   # création du tableau X3
Y3=f(X3)
```

La fonction f peut être utilisée avec des tableaux numpy puisqu'on utilise des fonctions mathématiques de la bibliothèque numpy. On verra dans l'exercice 3.1 « Tracé d'une fonction avec matplotlib » du chapitre « Graphiques » comment vectoriser une fonction, c'est-à-dire créer une fonction pouvant être utilisée avec des tableaux numpy. Le temps de calcul est dans ce cas beaucoup plus faible qu'avec des boucles `for`.

Exercice 2.2 : Affectation, objet immuable, copie de listes et de tableaux

1. Qu'affiche Python lors de l'exécution du programme ?

```
import numpy as np
import copy
i=3
j=i
print('Avant modification de i : i, j=', i, j)
i=5
print('Après modification de i : i, j=', i, j)
```

2. Qu'affiche Python lors de l'exécution du programme ?

```
L1=[1, 3, 5, 7]
L2=L1
print('Avant modification de L2 : L1, L2=', L1, L2)
L2[3]=2
L2[3]=print('Après modification de L2 : L1, L2=', L1, L2)
```

3. Qu'affiche Python lors de l'exécution du programme ?

```
L3=[1, 3, 5, 7]
L4=copy.deepcopy(L3)
print('Avant modification de L4 : L3, L4=', L3, L4)
L4[3]=2
print('Après modification de L4 : L3, L4=', L3, L4)
```

4. Qu'affiche Python lors de l'exécution du programme ?

```
A=np.array([1, 3, 5, 7])
C=copy.deepcopy(A)
B=A
B[1]=2
print('A, B=', A, B)
D=2*C+3
D[1]=2
print('C, D=', C, D)
```

5. Qu'affiche Python lors de l'exécution du programme ?

```
L1 = [1, 2, [3, 4], 5]
L2=L1
L3=copy.copy(L1)
L4=copy.deepcopy(L1)
L1[0]=12
L1[2][0]=30
print('L1 =',L1)
print('L2 =',L2)
print('L3 =', L3)
print('L4 =', L4)
```

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de copies de listes, tableaux numpy, deques et dictionnaires (voir chapitre 12 « Dictionnaire – Pile – File – Deque »). Voir également les exercices 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » (chapitre « Tableaux numpy – Slicing »), 12.4 « Manipulations de piles », 12.8 « Utilisation de deques » (chapitre « Dictionnaire – Pile – File – Deque ») ainsi que les exercices du chapitre 10 « Traitement d'images ».

Cours :

Il existe deux types d'objets dans Python :

- les objets dont la valeur peut changer sont dits mutables (ou muables) : listes, dictionnaires, deques (voir chapitre 12 « Dictionnaire – Pile – File – Deque »), tableaux numpy… ;
- les objets dont la valeur ne peut pas changer sont dits immutables (ou immuables) : entiers, flottants, complexes, booléens, chaînes de caractères, tuples…

Objets mutables (modifiables ou muables) – Partage de valeurs par plusieurs variables

```
L1=[1, 2, 3, 4]
```

Cette affectation (ou assignation) est une instruction qui réalise les opérations suivantes :

- Création d'un objet mutable (appelé `obj1`) de type `list` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj1` vaut : `[1, 2, 3, 4]`.
- Création de la variable `L1`.
- Association de la variable `L1` avec l'objet `obj1` contenant la valeur `[1, 2, 3, 4]`.

La variable `L1` ne contient pas `[1, 2, 3, 4]` mais uniquement la référence de l'objet `obj1`, c'est-à-dire l'adresse mémoire où est stocké `obj1`.

On peut modifier `[1, 2, 3, 4]` une fois que cet objet `obj1` de type `list` est créé. Les listes sont modifiables (mutables).

Cours :
Contrairement à d'autres langages de programmation (C ou Java), une affectation dans Python est une association d'une variable avec un objet contenant la valeur. C'est le choix des concepteurs du langage Python.

```
L2=L1
```

L'instruction `L2=L1` n'affecte pas `[1, 2, 3, 4]` à `L2` mais réalise les opérations suivantes :

- Création du nom de variable `L2`.
- Affectation à la variable `L2` de la référence (ou adresse mémoire) où est stocké `[1, 2, 3, 4]`.

`L1` et `L2` font donc référence au même objet `[1, 2, 3, 4]`.

La copie est très rapide puisqu'on n'occupe pas deux fois plus de place mémoire.

Si on modifie `[1, 2, 3, 4]` *via* `L1`, alors cette modification sera également visible par `L2`.

```
L1[0]=10
```

On constate que `L2[0]` vaut 10 également. C'est tout à fait normal car `L1` et `L2` font référence à la même adresse mémoire de `[10, 2, 3, 4]`.

On ajoute un élément dans `L1` avec la fonction `append` :

```
L1.append(12)
```

L'élément 12 est ajouté dans `L1` et `L2` puisque `L1` et `L2` font référence à la même liste modifiable (ou mutable) : `[10, 2, 3, 4, 12]`.

Comme dans les problèmes de concours, on utilise le langage suivant :

- Le terme « liste » appliqué à un objet Python signifie qu'il s'agit d'une variable de type `list`. Idem pour les autres types : `int`, `float`, `complex`, `bool`, `str`, `tuple`, `dict`, `deque`…
- Les termes « vecteur » et « tableau » désignent des objets numpy de type `np.ndarray`, respectivement à une dimension ou de dimension quelconque.

Cours :
Objets immutables (non modifiables ou immuables)

```
a=10
```

Cette affectation réalise les opérations suivantes :

- Création d'un objet immutable (appelé `obj2`) de type `int` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj2` vaut : `10`.
- Création de la variable `a`.
- Association de la variable `a` avec l'objet `obj2` contenant la valeur `10`.

La variable `a` ne contient pas `10` mais uniquement la référence de l'objet `obj2`, c'est-à-dire l'adresse mémoire où est stocké `obj2`.

On ne peut pas modifier `10` une fois que cet objet `obj2` de type `int` est créé. Les entiers sont non modifiables (immutables).

Cours :

```
b=a
```

Cette instruction n'affecte pas `10` à la variable `a` mais affecte la référence (ou l'adresse mémoire) où est stocké `10`.

```
a=11
```

Comme on ne peut pas modifier `10` (objet immutable), on crée un nouvel objet `11` avec une nouvelle adresse mémoire dans l'ordinateur. La variable `a` fait référence à l'adresse mémoire où est stocké `11`.

Par contre, `b` fait toujours référence à l'adresse mémoire où est stocké `10`.

```
print(b)    # b reste égal à 10
```

On retrouve le même résultat pour tous les objets immutables : entiers, flottants, booléens, chaînes de caractères, tuples…

Deux cas peuvent se présenter après l'exécution de l'instruction `L2=L1` :

- `L1` est un objet mutable ou muable (liste, dictionnaire, deque, tableau numpy…) : si on modifie `L1` dans la suite du programme, alors `L2` est également modifié puisque `L1` et `L2` font référence à la même adresse mémoire.
- `L1` est un objet immutable ou immuable (entier, nombre flottant, complexe, booléen, chaîne de caractères, tuple…) : si on modifie `L1` dans la suite du programme, alors `L2` n'est pas modifié.

Cours :
Copie profonde

```
import copy
L2=copy.deepcopy(L1)
```

Python exécute une copie profonde de `L1`, c'est-à-dire qu'il crée une nouvelle liste `L2` en copiant tous les éléments de `L1` dans `L2`. Dans ce cas, `L1` et `L2` ne font plus référence à la même adresse mémoire.

La copie profonde s'applique également aux tableaux numpy, dictionnaires et deques (voir chapitre 12 « Dictionnaire – Pile – File – Deque »).

Remarque :

Avec certains langages (langage C++, Java par exemple), le typage des variables est statique, c'est-à-dire qu'il faut d'abord déclarer (ou définir) le nom et le type des variables et ensuite leur affecter (ou assigner) une valeur compatible avec le type déclaré.

Avec le langage Python, le typage des variables est dynamique : l'interpréteur détermine le type automatiquement qui correspond au mieux à la valeur fournie lors de l'affectation.

1. Python affiche :

```
Avant modification de i : i, j= 3 3
Après modification de i : i, j= 5 3
```

Les résultats affichés dans Python sont tout à fait prévisibles. On va voir dans la question 2 que la même syntaxe appliquée aux listes donne des résultats surprenants !

2. Python affiche :

```
Avant modification de L2 : L1,L2= [1, 3, 5, 7]
                                  [1, 3, 5, 7]
Après modification de L2 : L1,L2= [1, 3, 5, 2]
                                  [1, 3, 5, 2]
```

Le programme de la question 2 est exactement le même que celui de la question 1 sauf qu'on manipule des listes au lieu de manipuler des entiers. Le comportement est complétement différent : la liste `L1` a été modifiée !

L'instruction `L2=L1` n'a pas effectué une copie de `L1` dans `L2` mais a copié uniquement la référence de la liste, c'est-à-dire l'adresse mémoire de la liste. `L1` et `L2` font donc référence à la même adresse mémoire de l'ordinateur. Si on modifie un élément de `L2` alors `L1` est également modifié.

3. Python affiche :

```
Avant modification de L4 : L3,L4= [1, 3, 5, 7]
                                  [1, 3, 5, 7]
Après modification de L4 : L3,L4= [1, 3, 5, 7]
                                  [1, 3, 5, 2]
```

L'instruction `L4=copy.deepcopy(L3)` permet de réaliser une copie profonde de `L3`. Les listes `L3` et `L4` ont des adresses mémoire différentes. La modification d'un élément de `L4` n'a donc aucune conséquence sur `L3`.

4. Python affiche :

```
A, B= [1 2 5 7] [1 2 5 7]
C, D= [1 3 5 7] [ 5  2 13 17]
```

On retrouve le même problème avec les tableaux numpy et les dictionnaires (voir exercice 12.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire – Pile – File – Deque »).

L'instruction `B=A` ne réalise pas une copie profonde mais copie uniquement la référence du tableau.

La modification d'une valeur du tableau `B` se répercute immédiatement sur `A` puisque `A` et `B` désignent le même tableau.

L'instruction `C=copy.deepcopy(A)` permet de réaliser une copie profonde du tableau `A`. Le tableau `C` n'est donc pas affecté par les modifications sur `A`.

La commande `D=2*C+3` est une opération qui nécessite la création d'un tableau `D`. Chaque élément du tableau `C` est multiplié par 2 et on ajoute la valeur 3. Le résultat de l'opération est affecté dans le tableau `D`.

Les modifications sur `D` n'ont aucune influence sur `C` puisqu'ils ont des adresses mémoire différentes.

5. Python affiche :

```
L1 = [12, 2, [30, 4], 5]
L2 = [12, 2, [30, 4], 5]
L3 = [1, 2, [30, 4], 5]
L4 = [1, 2, [3, 4], 5]
```

L'affectation `L2=L1` ne réalise pas une copie de `L1`. Si on modifie un élément de `L1`, alors cet élément est modifié dans `L2` puisque `L1` et `L2` pointent vers la même adresse mémoire.

L'instruction `L3=copy.copy(L1)` permet de réaliser une copie superficielle de `L1`. Si on modifie un élément de `L1` qui est une liste (exemple `[30, 4]`) alors cet élément est également modifié dans `L3`.

L'instruction `L4=copy.deepcopy(L1)` permet de réaliser une copie profonde de `L3`. Si on modifie un élément de `L1` qui est une liste (exemple `[30, 4]`) alors cet élément n'est pas modifié dans `L3`.

Exercice 2.3 : Passage par référence pour les listes et les tableaux numpy, effet de bord

On considère le programme suivant :

```
import numpy as np  # bibliothèque numpy renommée np

def f(a, b, L, tab):
    a+=1
```

```
    print('Print dans la fonction : a=', a)
    b=b+1
    print('Print dans la fonction : b=', b)
    L.append(4)
    print('Print dans la fonction : L=', L)
    tab2=tab
    tab2[0]=2
    print('Print dans la fonction : tab=', tab)
    return a
a=3
b=5
L=[1, 2, 3]
tab=np.array([1, 2, 3])
c=f(a, b, L, tab)
print('Print après la fonction : a=', a)
print('Print après la fonction : b=', b)
print('Print après la fonction : c=', c)
print('Print après la fonction f : L=', L)
print('Print après la fonction : tab=', tab)
```

Qu'affiche Python lors de l'exécution du programme ? Analyser les différents affichages de `print`.

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de l'utilisation de listes et de tableaux numpy dans les arguments d'entrée des fonctions. Dans l'exercice 12.4 « Manipulations de piles » dans le chapitre « Dictionnaire – Pile – File – Deque », on fera attention à ne pas modifier la pile initiale après l'appel d'une fonction.

Voir exercice 11.5 « Transformation d'un pseudo-code en programme Python » dans le chapitre « Tris » pour les problèmes rencontrés par la modification de la liste initiale après l'appel de la fonction.

Cours :

Les arguments d'entrée des fonctions sont tous passés par référence. On considère deux cas :

- objet mutable (liste, dictionnaire, deque, tableau numpy…) : toute modification de cet objet dans la fonction est visible en dehors de la fonction. C'est « l'effet de bord » puisque la fonction modifie des données définies hors de sa portée locale ;
- objet immutable (entier, nombre flottant, complexe, booléen, chaîne de caractères, tuple…) : toute modification de cet objet dans la fonction n'est pas visible en dehors de la fonction.

Si on modifie un argument d'entrée mutable (liste, dictionnaire, deque, tableau numpy…) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction.

Python affiche :

```
Print dans la fonction : a= 4
Print dans la fonction : b= 6
Print dans la fonction : L= [1, 2, 3, 4]
Print dans la fonction : tab= [2 2 3]
Print après la fonction : a= 3
Print après la fonction : b= 5
Print après la fonction : c= 4
Print après la fonction f : L= [1, 2, 3, 4]
Print après la fonction : tab= [2 2 3]
```

Avant l'appel de la fonction `f`, la variable `b` fait référence à `5`. La variable `b` est passée par référence dans la fonction `f` et fait toujours référence à `6`.

L'instruction `b=b+1` dans la fonction `f` ne peut affecter `5` qui est immutable. La variable `b` dans la fonction `f` fait donc référence à une nouvelle valeur `6`.

La variable `b` du programme principal garde donc la même valeur `6`.

La fonction `f` retourne la valeur `4` qui est affectée dans la variable `c`.

La variable `L` fait référence à la liste `[1, 2, 3]` qui est un objet mutable. La liste `L` est passée par référence. Lorsqu'on modifie `L` dans la fonction `f`, on modifie la liste `[1, 2, 3]`.

La variable `L` dans la fonction `f` fait référence à la même adresse mémoire que la variable `L` dans le programme principal.

L'instruction `tab2=tab` n'effectue pas une copie profonde du tableau. `tab` et `tab2` font référence à la même adresse mémoire. Cela permet de ne pas occuper deux fois plus de place mémoire. Si on modifie `tab2`, `tab` est affecté également puisqu'ils font référence à la même adresse mémoire. La copie `tab2=tab` est donc très rapide.

Remarque :

Pourquoi un passage par référence avec des objets mutables (modifiables) ?

Dans l'exercice 10.1 « Traitement d'images et filtrage passe-bas » du chapitre « Traitement d'images », on utilise des tableaux pour traiter des images. Ces tableaux peuvent contenir plusieurs millions d'éléments. Le passage par référence permet de gagner énormément de temps de calcul puisqu'on passe uniquement l'adresse mémoire du tableau et non les millions d'éléments de la matrice !

Si on modifie un argument d'entrée mutable (liste, dictionnaire, deque, tableau numpy) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction.

Remarque : Voir exercice précédent « Affectation, objet immuable, copie de listes et de tableaux » pour avoir une copie profonde avec la fonction `deepcopy`.

Exercice 2.4 : Slicing, tranche, `range`, `np.arange`, `np.linspace`

On considère le programme suivant :

```
import numpy as np
L1=[]
for i in range(2, 7, 2):
    L1.append(i)
print('L1 =', L1)
print(L1[:-1])

T1=np.arange(2, 7, 2)
print('T1 =', T1)
print(T1[1:3])

x1=np.linspace(0, np.pi, 5)
print('x1 =', x1)
print('x1[1:3] = ',x1[1:3])
```

Qu'affiche Python lors de l'exécution du programme ?

Analyse du problème

La technique du slicing, ou découpage en tranches, permet d'extraire des éléments d'une liste ou d'un tableau numpy de type `ndarray`. On l'utilise également pour les fonctions `range`, `np.arange` et `np.linspace`.

Cours :

Lorsqu'on veut extraire des éléments d'une liste `L` (par exemple `L=[3, 5, 1, 8, 10]`), on utilise la technique du slicing, ou découpage en tranches. Il suffit de mettre entre crochets les indices correspondant au début et à la fin de la « tranche ». On utilise l'instruction :

```
L[start:stop]
```

`start` : indice de départ, inclus
`stop` – `start` : longueur de la liste extraite (avec un pas de 1 par défaut)
`stop` : indice final, exclu

Exemple : `L[1:4]` renvoie `[5, 1, 8]`.

L'indice de départ vaut 1 avec `L[1]` = 5.
L'indice final vaut 4 – 1 = 3 avec `L[3]` = 8.
On récupère bien 3 éléments : `[5, 1, 8]`.

On peut utiliser des indices négatifs :

`L[-1]` renvoie le dernier élément de la liste : 10.
`L[-2]` renvoie 8.

Il faut bien retenir l'instruction : `L[start:stop]`
Les indices sont compris entre `start` inclus et `stop` exclu.
On peut retenir facilement que le nombre d'éléments vaut `stop` – `start` avec un pas de 1 par défaut.

Cours :
On peut utiliser le slicing avec un pas différent de 1. On utilise alors l'instruction suivante :

```
L[start:stop:step]
```

`start` : indice de départ, inclus
`stop` : indice final, exclu
`step` : cette variable désigne le pas.

Exemple : `L[1:4:2]` renvoie `[5, 8]`.

L'indice de départ vaut 1 avec `L[1]` = 5.
L'indice final 4 est exclu.
On ne prend qu'un élément sur 2.

On peut omettre n'importe lequel des arguments dans `L[start:stop:step]`.

Par défaut : `start` = 0, `stop` = indice du dernier élément de la liste + 1, `step` = 1.

`L[:3]` renvoie tous les éléments dont les indices sont compris entre 0 inclus et 3 exclu. Python retourne `[3, 5, 1]`.

`range` permet de générer une suite de valeurs.

Pour créer une boucle `for`, on utilise l'instruction suivante :

```
L1=[]   # création d'une liste vide
for i in range(3):   # i varie entre 0 inclus et 3 exclu
    L1.append(i):    # ajoute la valeur i à la liste L1
```

On obtient : `L1 = [0, 1, 2]`.

On peut utiliser également le slicing dans `range` :

```
for i in range(start,stop,step):
    # i varie entre start inclus et stop exclu avec un pas égal à step
    print(i)
```

Attention aux séparateurs dans l'extraction de liste et dans la fonction `range` :

```
L[start:stop:step]       # mettre deux points entre start et stop
range(start,stop,step)   # mettre une virgule entre start et stop
```

Cours :
On peut générer une liste de valeurs avec `range` :

```
L2=[i for i in range(3)]   # on obtient L2=[0, 1, 2].
                           # Création par compréhension
```

La fonction `np.arange` permet de générer des tableaux numpy :

```
import numpy as np    # bibliothèque numpy renommée np
tab1=np.arange(0,4)   # renvoie un tableau numpy [0, 1, 2, 3]
```

On peut utiliser également le slicing dans `np.arange` :

```
tab2=np.arange(3,5,0.5)   # renvoie un tableau numpy[3., 3.5, 4., 4.5]
```

`np.arange(start,stop,step)` permet de générer un tableau numpy : la première valeur est `start`. La dernière valeur, `stop`, est exclue. Le pas est `step`.

Pour générer des tableaux numpy, on peut utiliser la fonction `np.linspace` :

```
tab3=np.linspace(start, stop, N)
```

On obtient un tableau numpy dont la première valeur est `start`, la dernière valeur est `stop`. Cette valeur `stop` est comprise dans le tableau. `N` désigne le nombre de points (répartis d'une façon régulière) entre `start` et `stop`.

```
tab4=np.linspace(3, 4.5, 4)    # tableau numpy [3., 3.5, 4., 4.5]
                               # avec 4 points
```

Ne pas confondre `np.arange` et `np.linspace` :

`np.arange(start, stop, step)` : la dernière valeur `stop` est exclue. `step` désigne le pas.

`np.linspace(start, stop, N)` : la dernière valeur `stop` est incluse. `N` désigne le nombre de points.

Python affiche :

```
L1 = [2, 4, 6]
[2, 4]
```

Liste `L1` : *i* varie entre 2 inclus et 7 exclu avec un pas de 2.

`L1[:-1]` extrait les éléments de `L1` : le dernier élément de `L1` est exclu.

Python affiche :

```
T1 = [2, 4, 6]
[4  6]
```

Tableau numpy `T1` : *i* varie entre 2 inclus et 7 exclu avec un pas de 2.

`T1[1:3]` extrait les éléments d'indices 1 à 2 du tableau `T1`. On obtient 3 – 1 = 2 éléments.

Pour `x1`, on a 5 valeurs entre 0 et π compris.

Python affiche :

```
x1 = [0., 0.78539816, 1.57079633, 2.35619449,
      3.14159265]
x1[1:3] = [0.78539816, 1.57079633]
```

Graphiques

3

Exercice 3.1 : Tracé d'une fonction avec matplotlib

On considère les fonctions f_1 et f_2 définies sur [0, 2] par :

$$f_1(x) = \begin{cases} x & \text{pour } 0 \leq x < 1 \\ 1 & \text{pour } 1 \leq x \leq 2 \end{cases} \quad \text{et } f_2(x) = \sin(x) + 0{,}1.$$

Les fonctions suivantes permettent le tracé de fonctions :

```
import matplotlib.pyplot as plt
plt.figure()   # nouvelle fenêtre graphique
plt.plot(x,y,color='r', linewidth =3,marker='o')
   # color : choix de la couleur
   # ('r' : red, 'g' : green, 'b' : blue, 'black' : black)
   # linewidth : épaisseur du trait
   # marker : différents symboles '+', '.', 'o', 'v'
   # linestyle : style de la ligne ('-' ligne continue,
   # '--' ligne discontinue, ':' ligne pointillée)
plt.plot(x, y, '*')   # points non reliés représentés par '*'
plt.grid()   # affichage de la grille
plt.title('Titre')   # ajout d'un titre
plt.xlabel('axe x')  # affiche 'axe x' en abscisse d'un graphique
plt.ylabel('axe y')  # affiche 'axe y' en ordonnée d'un graphique
plt.axis ([xmin, xmax, ymin, ymax])   # précise les bornes pour les
                                      # abscisses et les ordonnées
plt.legend(['courbe 1', 'courbe 2'])  # permet de légender les courbes
plt.show()   # affiche la figure à l'écran

import numpy as np
f_vect=np.vectorize(f)  # vectorisation de la fonction f
         # transforme la fonction scalaire en une fonction pouvant être
         # utilisée avec des tableaux numpy
```

1. Définir les deux fonctions f_1 et f_2 dans Python en utilisant la bibliothèque math. Tracer les courbes représentatives des deux fonctions sur l'intervalle [0, 2] avec un pas de 0,05. Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de listes.
- Courbe représentative de f_1 : épaisseur du trait égale à 3, couleur bleue.
- Courbe représentative de f_2 : points non reliés représentés par *, couleur rouge.
- Légender les deux courbes.
- Afficher `'x'` pour l'axe des abscisses et `'y'` pour l'axe des ordonnées.

- Afficher le titre : `'Tracé de fonctions'`.
- Axe des x compris entre 0 et 2, axe des y compris entre 0 et 1,5.

2. Définir les deux fonctions f_1 et f_2 dans Python en utilisant la bibliothèque numpy. On pourra vectoriser la fonction f_1. Tracer les courbes représentatives des deux fonctions sur l'intervalle [0, 2]. Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de `np.linspace` avec 21 points et de tableaux numpy.
- Courbe représentative de f_1 : épaisseur du trait égale à 3, couleur noire, ligne discontinue.
- Courbe représentative de f_2 : couleur bleue, symbole `'o'` et ligne continue.
- Légender les deux courbes.
- Afficher `'x'` pour l'axe des abscisses et `'y'` pour l'axe des ordonnées.
- Afficher la grille.
- Afficher le titre : `'Tracé de fonctions'`.
- Même échelle sur les axes.

Analyse du problème

Il faut bien connaître les fonctions suivantes pour tracer une fonction :

```
import matplotlib.pyplot as plt
plt.figure()    # nouvelle fenêtre graphique
plt.plot(x,y)   # représentation graphique de y en fonction de x
plt.show()      # affiche la figure à l'écran
```

Cours :

La bibliothèque matplotlib de Python permet de tracer des graphiques. On l'importe à l'aide de la commande :

```
import matplotlib.pyplot as plt
```

Il faut connaître quelques fonctions de la bibliothèque matplotlib :

- `plt.plot(x, y)`
  Arguments d'entrée : un vecteur d'abscisses x (tableau de dimension n) et un vecteur d'ordonnées y (tableau de dimension n). On peut utiliser également une liste d'abscisses x de longueur n et une liste d'ordonnées y de longueur n.
  Description : fonction permettant de tracer un graphique de n points dont les abscisses sont contenues dans le vecteur **x** et les ordonnées dans le vecteur **y**. Cette fonction doit être suivie de la fonction `plt.show()` pour que le graphique soit affiché.
- `plt.xlabel(nom)`
  Argument d'entrée : une chaîne de caractères.
  Description : fonction permettant d'afficher le contenu de `nom` en abscisse d'un graphique.

- `plt.ylabel(nom)`
  Argument d'entrée : une chaîne de caractères.
  Description : fonction permettant d'afficher le contenu de `nom` en ordonnée d'un graphique.
- `plt.title(nom)`
  Argument d'entrée : une chaîne de caractères.
  Description : fonction permettant d'afficher le contenu de `nom` en titre d'un graphique.
- `plt.show()`
  Description : fonction réalisant l'affichage d'un graphe préalablement créé par la commande `plt.plot(x,y)`. Elle doit être appelée après la fonction `plt.plot` et après les fonctions `plt.xlabel`, `plt.ylabel` et `plt.title`.
- `plt.axis([xmin, xmax, ymin, ymax])`
  Description : fonction permettant de définir les valeurs limites pour l'axe des abscisses et celui des ordonnées.
- `plt.xlim([xmin, xmax])`
  Description : fonction permettant de définir les valeurs limites pour l'axe des abscisses.
- `plt.ylim([ymin, ymax])`
  Description : fonction permettant de définir les valeurs limites pour l'axe des ordonnées.

Voir exercice 11.7 « Tri par comptage, histogramme » dans le chapitre « Tris » pour le tracé d'histogrammes.

1.

```
import matplotlib.pyplot as plt
import math

def f1(x):   # définition de la fonction f1
    if x>=0 and x<1:
        return x
    elif x>=1 and x<=2:
        return 1
    else:
        return 0

def f2(x):   # définition de la fonction f2
    return math.sin(x)+0.1


xmin=0   # valeur minimale de x
xmax=2   # valeur maximale de x
pas=0.05
N=int(((xmax-xmin)/pas)+1)   # pas=(xmax-xmin)/(N-1)
  # le nombre de points doit être un entier
  # N=nombre de points et N-1=nombre d'intervalles

x=[]    # initialisation de la liste x
y1=[]   # initialisation de la liste y1
y2=[]   # initialisation de la liste y2
```

```
for i in range(N):    # i varie entre 0 inclus et N exclu
    x.append(i*pas)   # ajout de l'élément x[i]
    y1.append(f1(x[i]))   # ajout de l'élément f1(x[i])
    y2.append(f2(x[i]))   # ajout de l'élément f2(x[i])


plt.figure()   # nouvelle fenêtre graphique
plt.plot(x, y1, linewidth=3, color='b')   # création de la
                                          # première courbe
plt.plot(x, y2, '*', color='r')   # création de la deuxième courbe
plt.legend(['f1(x)', 'f2(x)'])    # légende des deux courbes
plt.xlabel('x')   # affichage de 'x' en abscisse du graphique
plt.ylabel('y')   # affichage de 'y' en ordonnée du graphique
plt.title('Tracé de fonctions')  # affichage du titre du graphique
plt.axis([0,2,0,1.5])   # [xmin,xmax,ymin,ymax]
plt.show()   # affiche la figure à l'écran
```

2.

```
import numpy as np   # bibliothèque numpy renommée np
import matplotlib.pyplot as plt

def f1(x):   # définition de la fonction f1
    if x>=0 and x<1:
        return x
    elif x>=1 and x<=2:
        return 1
    else:
        return 0

def f2(x):   # définition de la fonction f2
    return np.sin(x)+0.1

N=21   # nombre de points
X=np.linspace(xmin, xmax, N)   # valeurs comprises entre xmin
                               # et xmax

f1_vect=np.vectorize(f1)
  # transforme la fonction scalaire en une fonction
  # pouvant être utilisée avec des tableaux numpy

Y1=f1_vect(X)   # applique f1 à chaque élément de X
    # temps de calcul moins long qu'avec des boucles
Y2=f2(X)   # fonction déjà vectorisée avec np.sin

plt.figure()   # nouvelle fenêtre graphique
plt.plot(X, Y1, linewidth=3, color='black', linestyle='--')
plt.plot(X, Y2, color='b', marker='o', linestyle='-')
plt.legend(['f1(x)', 'f2(x)'])
plt.xlabel('x')
plt.ylabel('y')
plt.grid()   # affichage de la grille
plt.title('Tracé de fonctions')
plt.show()   # affiche la figure à l'écran
```

Exercice 3.2 : Tracé d'un histogramme avec matplotlib

On considère la liste `L=[2, 2, 6, 6, 3, 14, 14, 6, 8, 9, 9]`.

Les fonctions suivantes permettent le tracé d'histogrammes :

```
import matplotlib.pyplot as plt
plt.figure() # nouvelle fenêtre graphique
plt.hist(L, range=(5, 25), bins=10, color='red')
 # range=(5, 25) permet de préciser le minimum et le maximum des
 # valeurs représentées sur l'histogramme.
 # Par défaut : range=(min(L), max(L)), bins=10=nombre d'intervalles
 # ou bins=[5, 15, 20, 25, 40] permettant de préciser les limites
 # des intervalles. Les barres de l'histogramme ont dans cet
 # exemple des largeurs différentes.
 # color='red' permet de préciser la couleur des barres
plt.show()   # affiche la figure à l'écran
```

Tracer l'histogramme de la liste `L` avec les caractéristiques suivantes :

- Afficher « Valeurs de L » pour l'axe des abscisses et « Nombre d'occurrences » pour l'axe des ordonnées.
- Afficher le titre : « Histogramme de la liste L ».
- 12 intervalles.

Analyse du problème

Voir exercice précédent « Tracé d'une fonction avec matplotlib » pour l'affichage du titre, de l'axe des abscisses et de l'axe des ordonnées.

Il faut bien connaître les fonctions suivantes pour tracer un histogramme :

```
import matplotlib.pyplot as plt
plt.figure()  # nouvelle fenêtre graphique
plt.hist(L)   # tracé de l'histogramme
plt.show()    # affiche la figure à l'écran
```

```
import matplotlib.pyplot as plt
L=[2, 2, 6, 6, 3, 14, 14, 6, 8, 9, 9]
plt.figure()          # nouvelle fenêtre graphique
plt.hist(L, bins=12)  # tracé de l'histogramme de la liste L
                      # avec 12 intervalles
plt.title('Histogramme de la liste L') # titre de l'histogramme
plt.xlabel('Valeurs de L')
plt.ylabel("Nombre d'occurrences")
plt.show()            # affiche la figure à l'écran
```

Partie 2

Terminaison – Correction – Complexité

Plan

Terminaison – Correction – complexité

4

Exercice 4.1 : Comparaison de deux listes (Mines Ponts 2017)

1. Proposer une fonction `egal(L1, L2)` retournant un booléen permettant de savoir si deux listes `L1` et `L2` sont égales.

2. Que peut-on dire de la complexité de cette fonction ?

3. Préciser le type de retour de cette fonction.

Analyse du problème

On se place dans le pire des cas avec deux listes égales. On calcule le nombre total d'opérations élémentaires pour déterminer la complexité de cette fonction.

Cours :
Terminaison d'un programme

Un programme itératif est constitué d'une boucle `for` ou `while`. On cherche à démontrer que cette boucle se termine.

On considère un **variant de boucle**, par exemple un entier naturel qui décroît à chaque itération de la boucle et qui atteindra 0 à un moment, ce qui permet de montrer que la boucle se termine.

On considère le programme suivant qui cherche un élément dans une liste non triée.

```
L=[3, 1, 6, 9, 19, -1,20]
def rec_pos(L, x):
    # la fonction renvoie True et i l'indice si x est dans la liste
    # ou le tableau L, False sinon
    i=0                   # initialisation de l'indice i
    n=len(L)
    while i<n:
        if x==L[i]:
            return True,i # return provoque un arrêt de boucle
                          # si x est dans L
        i=i+1
    return False, -1      # l'élément n'est pas trouvé
print(rec_pos(L,6))       # le programme affiche : (True, 2)
```

On appelle n le nombre d'éléments de la liste L. On considère le variant de boucle : $n - i$.

- Le variant de boucle vaut n à l'entrée de la boucle `while`.
- Il décroît de 1 à chaque passage dans la boucle si l'élément x n'est pas dans L. Si l'élément est trouvé, on quitte la boucle.
- En sortie de boucle, si l'élément n'est pas trouvé, le variant de boucle vaut $n - n = 0$.

Dans tous les cas, on a démontré la terminaison du programme.

Voir le chapitre 7 « Récursivité » pour des exemples avec des fonctions récursives.

Correction d'un programme

Pour démontrer la correction d'un programme, il faut montrer que l'algorithme effectue bien la tâche souhaitée. On utilise souvent une démarche proche du raisonnement par récurrence.

Rédaction pour un programme itératif

La propriété P_n (appelée invariant de boucle) doit avoir trois caractéristiques :

- La propriété doit être vérifiée avant d'entrer dans la boucle.
- Si la propriété est vérifiée avant une itération, elle doit être vérifiée après cette itération.
- Si la propriété est vérifiée en fin de boucle, alors le programme est correct.

On considère le programme suivant qui calcule la factorielle d'un entier naturel.

```
def fact(n):
    # la fonction renvoie la factorielle de n (int)
    res=1                       # initialisation de res à 1
    for i in range(1,n+1):      # i varie entre 1 inclus et n+1 exclu
        res=res*i
    return res

print('Factorielle :',fact(4)) # affiche 24
```

On considère la propriété (appelée **invariant de boucle**) P_i : $res = i!$.

- Avant d'entrer dans la boucle, $i = 1$ et on a bien $res = 1!$.
- Si P_i est vraie, alors $res = i!$. Dans la boucle, le programme fait le calcul : $\text{res} \times (i+1) = i! \times (i+1) = (i+1)!$.
  res prend alors la valeur $(i+1)!$. La propriété P_{i+1} est donc vraie.
- En fin de boucle, i vaut n et on a bien $res = n!$.

On a démontré que le programme est correct.

Rédaction pour un programme récursif

Voir le chapitre 7 « Récursivité » pour la rédaction avec des fonctions récursives.

On distingue parfois correction partielle et correction totale :

- La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête.
- La correction est totale si elle est partielle et si l'algorithme termine.

Complexité

On distingue complexité en temps et complexité en espace. La complexité en temps mesure la durée nécessaire à l'exécution du programme, alors que la complexité en espace mesure la taille mémoire nécessaire à l'exécution du programme. On étudiera par la suite la complexité en temps.

La complexité est une mesure du nombre d'opérations élémentaires que l'algorithme effectue. Si on se place dans les conditions les plus favorables, on calculera la complexité dans le meilleur des cas. Si on se place dans les conditions les plus défavorables, on calculera la complexité dans le pire des cas.

On considère le programme suivant, qui cherche le maximum d'une liste non triée.

```
def rec_max(L):
    # la fonction renvoie le maximum des éléments de la liste
    # ou du tableau L
    maxi=L[0]
    n=len(L)               # nombre d'éléments de L
    for i in range(1, n):  # i varie entre 1 inclus et n exclu
        if L[i]>maxi:
            maxi=L[i]
    return maxi
```

On cherche à évaluer la complexité de cette fonction dans le cas le moins favorable.

On appelle n le nombre d'éléments de la liste L. On note $O(n)$ la complexité de la fonction `rec_max(L)`.

On cherche à calculer le nombre d'opérations élémentaires :

- 2 opérations élémentaires : appel de l'élément $L[0]$ et affectation de *maxi*.
- 2 opérations élémentaires : appel du nombre d'éléments de L et affectation de n.
- Pour chaque étape de la boucle `for`, on a 3 opérations élémentaires : appel de l'élément $L[i]$, comparaison, affectation.

La boucle `for` est exécutée $n - 1$ fois dans le pire des cas (si la liste est triée dans l'ordre croissant).

Le nombre d'opérations élémentaires vaut : $4 + 3(n - 1) = 1 + 3n$.

La complexité est linéaire en $O(n)$ pour la fonction `rec_min`.

On rencontre les types suivants de complexité :

- constante en $O(1)$ (ne dépend pas de n),
- logarithmique en $O(\log n)$,
- linéaire en $O(n)$,
- quasi linéaire en $O(n \log n)$,
- quadratique en $O(n^2)$,
- polynomiale en $O(n^p)$,
- exponentielle en $O(2^n)$.

1.

```
def egal(L1, L2):
    # la fonction renvoie True si les deux listes L1 et L2
    # sont égales, False sinon
    if len(L1)!=len(L2):        # listes de longueurs différentes
        return False
    else:
        for i in range(len(L1)):# parcourt tous les éléments de L1
            if L1[i]!=L2[i]:
                return False
```

```
    return True

L1=[3, 2, 5, 8]
L2=[3, 2, 5, 8]
print(egal(L1, L2)                # retourne True
```

2. On se place dans le pire des cas avec deux listes égales. On appelle *n* la longueur de la liste `L1`.

On calcule le nombre d'opérations élémentaires de la fonction `egal(L1, L2)` :

- 3 opérations élémentaires : appel de la longueur de `L1`, appel de la longueur de `L2`, test.
- À chaque appel de la boucle `for` : 3 opérations élémentaires (test et appel de `L1[i]`, `L2[i]`).

On a donc 3 + 3*n* opérations élémentaires.

La complexité est linéaire en $O(n)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

3. Le type de retour de la fonction `egal(L1, L2)` est `bool` (booléen).

Exercice 4.2 : Amélioration de la complexité (Mines Ponts 2018)

On s'intéresse à des mesures de niveau de la surface libre de la mer. La distribution des hauteurs de vague lors de l'analyse vague par vague est réputée être gaussienne. On peut contrôler ceci par des tests de *skewness* (variable désignée par *S*) et de *kurtosis* (variable désignée par *K*) définis ci-après. Ces deux tests permettent de quantifier respectivement l'asymétrie et l'aplatissement de la distribution.

On appelle $\overline{H}$ et σ^2 les estimateurs non biaisés de l'espérance et de la variance, *n* le nombre d'éléments $H_1, H_2, ..., H_n$. On définit alors :

$$S = \frac{n}{(n-1)(n-2)} \times \left(\frac{1}{\sigma^3}\right) \times \sum_{i=1}^{n} \left(H_i - \overline{H}\right)^3 \text{ et}$$

$$K = \frac{n}{(n-1)(n-2)(n-3)} \times \left(\frac{1}{\sigma^4}\right) \times \sum_{i=1}^{n} \left(H_i - \overline{H}\right)^4 - \frac{3(n-1)^2}{(n-2)(n-3)}$$

On suppose disposer de la fonction `ecartType` qui permet de retourner la valeur de l'écart-type non biaisé σ.

1. Proposer une fonction `moyenne` prenant en argument une liste non vide `L` et retournant sa valeur moyenne.

2. Un codage de la fonction `skewness` pour une liste ayant au moins 3 éléments est donné ci-dessous. Le temps d'exécution est anormalement long. Proposer une modification simple de la fonction pour diminuer le temps d'exécution (sans remettre en cause le codage des fonctions `ecartType` et `moyenne`).

```
def skewness (liste_hauteurs):
    n=len(liste_hauteurs)
    et3=(ecartType(liste_hauteurs))**3
    S=0
    for i in range(n):
        S+=(liste_hauteurs[i]-moyenne(liste_hauteurs))**3
        S=n/(n-1)/(n-2)*S/et3
    return S
```

3. Doit-on s'attendre à une différence de type de la complexité entre une fonction évaluant S et une fonction évaluant K ?

Analyse du problème

On calcule le nombre total d'opérations élémentaires pour déterminer la complexité de cette fonction.

1.

```
def moyenne(L):
    # la fonction renvoie la valeur moyenne de la liste L
    som=0                # initialisation de som à 0
    n=len(L)             # nombre d'éléments de L
    for i in range(n):   # i varie entre 0 inclus et n exclu
        som=som+L[i]
    return(som/n)
```

2. La boucle `for` de la fonction `skewness` est exécutée n fois. À chaque étape de cette boucle `for`, on calcule `moyenne(liste_hauteurs)` qui fait intervenir une autre boucle exécutée n fois.

On a donc une complexité quadratique en $O(n^2)$.

Pour améliorer la complexité de cette fonction, on calcule `moyenne(liste_hauteurs)` avant la boucle `for`. Dans ce cas, on a une complexité linéaire en $O(n)$:

```
def ecartType(L):
    # la fonction renvoie l'écart-type de la liste L
    return np.sqrt(np.var(L))

def skewness (liste_hauteurs):
    # la fonction renvoie S pour la liste liste_hauteurs
    n=len(liste_hauteurs)
    et3=(ecartType(liste_hauteurs))**3
    moy=moyenne(liste_hauteurs)
```

```
    S=0
    for i in range(n):
        S+=(liste_hauteurs[i]-moy)**3
        S=n/(n-1)/(n-2)*S/et3
    return S
```

3. On a une seule boucle pour calculer *S* et *K*. Il n'y a pas de différence de type de la complexité.

Exercice 4.3 : Décomposition en base *b* d'un entier (CCP MP Maths 2015)

1. Donner la décomposition binaire (en base 2) de l'entier 21.

2. On considère la fonction `mystere` suivante :

```
def mystere(n, b):
    """Données: n > 0 un entier et b > 0 un entier
    Résultat: ......."""
    t = []   # liste vide
    while n > 0:
        c = n % b
        t.append(c)
        n = n // b
    return t
```

Pour $k \in \mathbb{N}^*$, on note c_k, t_k et n_k les valeurs prises par les variables `c`, `t` et `n` à la sortie de la k-ième itération de la boucle `while`. Quelle liste est renvoyée lorsque l'on exécute `mystere(256,10)` ?

On recopiera et complétera le tableau suivant, en ajoutant les éventuelles colonnes nécessaires pour tracer entièrement l'exécution.

k	1	2	…
c_k			…
t_k			…
n_k			…

3. Soit $n > 0$ un entier. On exécute `mystere(n,10)`. On pose $n_0 = n$.

a. Justifier la terminaison de la boucle `while`.

b. On note p le nombre d'itérations lors de l'exécution de `mystere(n,10)`. Justifier que, pour tout $k \in [\![0, p]\!]$, on a $n_k \leq \dfrac{n}{10^k}$. En déduire une majoration de p en fonction de n.

4. En s'aidant du script de la fonction `mystere`, écrire une fonction `somme_chiffres` qui prend en argument un entier naturel et renvoie la somme de ses chiffres. Par exemple, `somme_chiffres(256)` devra renvoyer 13.

5. Écrire une version récursive de la fonction `somme_chiffres`, on la nommera `somme_chiffres_rec`.

Analyse du problème

On étudie dans cet exercice, extrait du concours CCP MP Maths 2015, la décomposition en base b d'un entier. La méthode est d'effectuer des divisions euclidiennes successives pour obtenir la liste des chiffres de la décomposition en base b d'un entier.

1. La décomposition binaire de l'entier 21 s'écrit : $21 = 10101_b$.
On a effet : $21 = 24 + 22 + 1 = \boxed{1} + 2 \times (\boxed{0} + 2 \times (\boxed{1} + 2 \times (\boxed{0} + 2 \times (\boxed{1}))))$.

Remarque :

On peut effectuer les divisions euclidiennes suivantes :

a) Division euclidienne de l'entier 21 par 2 :

$21//2 = 10$ = quotient

$21\%2 = \boxed{1}$ = reste

Le premier reste fournit le premier chiffre du code binaire, c'est-à-dire $1010\boxed{1}_b$.

b) Division euclidienne du quotient précédent par 2 :

$10//2 = 5$

$10\%2 = \boxed{0}$

Le deuxième reste fournit le deuxième chiffre du code binaire, c'est-à-dire $101\boxed{0}1_b$.

c) Division euclidienne du quotient précédent par 2 :

$5//2 = 2$

$5\%2 = \boxed{1}$

d) Division euclidienne du quotient précédent par 2 :

$2//2 = 1$

$2\%2 = \boxed{0}$

e) Division euclidienne du quotient précédent par 2 :

$1//2 = 0$

$1\%2 = \boxed{1}$

On en déduit que : $21 = 10101_b = 2^4 + 2^2 + 1 = \boxed{1} + 2 \times (\boxed{0} + 2 \times (\boxed{1} + 2 \times (\boxed{0} + 2 \times (\boxed{1}))))$.

On peut envisager une boucle où on effectue les divisions euclidiennes successives. On termine la boucle dès que le quotient est nul.

Cet algorithme permet d'effectuer la division euclidienne de l'entier n par deux jusqu'à ce que le résultat soit nul. La suite des restes donne le code binaire en ordre inverse.

2.

k	1	2	3
c_k	$256\%10 = 6$	$25\%10 = 5$	$2\%10 = 2$
t_k	[6]	[6, 5]	[6, 5, 2]
n_k	25	2	0

La fonction `mystere(256,10)` renvoie la liste `[6,5,2]`.
Cette procédure donne la décomposition en base b de l'entier n.

Cours :
Un variant de boucle sert à démontrer qu'une boucle se termine.

On peut utiliser par exemple un entier naturel qui décroît strictement à chaque itération. Il finit par atteindre 0 à un moment, on est alors à la fin de la boucle.

3.

a. Le variant de boucle est n. On notera n_k la liste des valeurs successives prises par n au cours des différentes itérations.

- Initialisation : $n_0 = n$.
- À l'étape k, le variant de boucle est n_k. À l'étape k+1, on a $n_{k+1} = n_k//b$. n_{k+1} est le quotient de la division euclidienne de n_k par b. Il existe r tel que : $n_k = b\ n_{k+1} + r$ avec $0 \leq r < b$. On a $n_{k+1} < n_k$. La suite est donc strictement décroissante.
- En sortie de boucle, n_p est nul car b est un entier strictement positif et le quotient est nul dès que n est plus petit que b.

b. Démonstration par récurrence : propriété P_k : $n_k \leq \dfrac{n}{10^k}$

- La propriété est vraie pour $k = 0$ puisque $10^0 = 1$.
- Supposons la propriété vraie pour le rang k. $n_{k+1} = n_k//10$: c'est le quotient de la division euclidienne de n_k par 10. Il existe r tel que : $n_k = 10\ n_{k+1} + r$ avec $0 \leq r < 10$. D'où $\dfrac{nk}{10} = n_{k+1} + \dfrac{r}{10}$, soit $n_{k+1} \leq \dfrac{n_k}{10}$. Or $n_k \leq \dfrac{n}{10^k}$. On en déduit immédiatement que $n_{k+1} \leq \dfrac{n}{10^{k+1}}$. La propriété est donc vraie au rang k+1.

Pour la dernière itération de la boucle, on a $n_p = 0$. Pour l'itération précédente, on a : $1 \leq n_{p-1} \leq 9$. On a donc $n_{p-1} \geq 1$.

On a vu que $n_{p-1} \leq \dfrac{n}{10^{p-1}}$, soit $10^{p-1} \leq \dfrac{n}{n_{p-1}}$. Comme $n_{p-1} \geq 1$, alors $10^{p-1} \leq n$, d'où $p-1 \leq \log_{10}(n)$. Finalement, on a :

$$p \leq \log_{10}(n+1)$$

4. On reprend le programme de la question 2. Les différents chiffres de l'entier *n* sont obtenus par les restes des divisions euclidiennes successives.

```
def somme_chiffres(n, b):
    """Données: n > 0 un entier et b > 0 un entier
    Résultat: ......."""
    somme=0   # initialisation de la somme
    while n > 0:
        r = n % b   # calcul de reste de la division
                    # euclidienne de n par b
        somme+=r    # on ajoute le reste r à somme
        n = n // b
    return somme
```

Cours :
Dans toute procédure récursive, l'instruction `return` doit être présente au moins deux fois : une fois pour la condition d'arrêt (premier `return` dans le programme) et une autre fois pour l'appel récursif (dernier `return` dans le programme)

5.

```
def somme_chiffres_rec(n, b):
    """Données: n > 0 un entier et b > 0 un entier
    Résultat: ......."""
    if n==0:
        return 0   # condition d'arrêt
    else:
        return (n%b + somme_chiffres_rec(n//b,b))    # appel
                                                     # récursif
```

Remarque :

L'entier 37 s'écrit 100101_b en binaire. On utilise une liste contenant les valeurs des différents bits. Il existe deux conventions pour repérer les bits dans une liste :

- $L = [1,0,0,1,0,1]$: $L[0]$ représente le bit de poids le plus fort et $L[5]$ représente le bit de poids le plus faible.
- $L = [1,0,1,0,0,1]$: $L[0]$ représente le bit de poids le plus faible et $L[5]$ représente le bit de poids le plus fort.

Il faut bien regarder si l'énoncé impose une convention. Sinon il faut bien la définir dans le programme pour éviter toute ambiguïté. La fonction `mystere(37,2)` retourne alors : `[1, 0, 1, 0, 0, 1]`. On utilise donc la deuxième convention dans tout l'exercice.

Exercice 4.4 : Recherche du nombre de zéros (banque PT 2015)

Soit un entier naturel *n* non nul et une liste `T` de longueur *n* dont les termes valent 0 ou 1. On cherche le nombre maximal de 0 contigus dans `T` (c'est-à-

dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste `T1` suivante vaut 4 :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T1[i]	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

1. Écrire une fonction `nombreZeros(T,i)` qui admet pour argument une liste `T` de longueur n, un indice `i` compris entre 0 et n–1, et qui retourne :

- 0, si `T[i]` = 1 ;
- le nombre de zéros consécutifs dans `T` à partir de `T[i]` inclus, si `T[i]` = 0.

Par exemple, les appels `nombreZeros(T1,4)`, `nombreZeros(T1,1)` et `nombreZeros(T1,8)` renvoient respectivement les valeurs 3, 0 et 1.

2. Comment obtenir le nombre maximal de zéros contigus d'une liste `T` connaissant la liste des `nombreZeros(T,i)` pour $0 \leq i \leq n-1$?

En déduire une fonction `nombreZerosMax(T)`, de paramètre `T`, renvoyant le nombre maximal de 0 contigus d'une liste `T` non vide. On utilisera la fonction `nombreZeros`.

3. Évaluer la complexité de la fonction `nombreZerosMax`.

4. Trouver un moyen simple, toujours en utilisant la fonction `nombreZeros`, d'obtenir un algorithme plus performant.

Analyse du problème

Dans cet exercice, on parcourt une liste pour déterminer le nombre maximal de zéros contigus dans celle-ci. On verra comment améliorer l'algorithme. Cet exercice est extrait du sujet de concours 0 de la banque PT 2015.

1.

```
def nombreZeros(T,i):
    if T[i]==1:
        return 0
    else:
        nbr_zeros=0
        j=i
        while j<len(T) and T[j]==0:
            # il faut mettre j<len(t) avant T[j]
            # sinon message d'erreur pour j=len(T)
            nbr_zeros+=1   # on incrémente de 1
                           # le nombre de zéros
            j+=1
    return nbr_zeros
```

2. On utilise la fonction `nombreZeros` pour définir une liste `L` : chaque élément `L[i]` de la liste `L` contient le nombre de zéros consécutifs dans `T` à partir de `T[i]`.

Il suffit ensuite de déterminer le maximum de la liste `L`.

```
def nombreZerosMax(T):
    L=[]   # initialisation de la liste
    for i in range(len(T)):
        L.append(nombreZeros(T,i))    # on stocke dans la liste
                                      # nombreZeros(T,i)

    max_T=L[0]   # recherche du maximum de la liste L
    for i in range(1,len(L)):   # i varie entre 1 et len(L)-1
        if L[i]>max_T:
            max_T=L[i]
    return max_T
```

3. On appelle n la longueur de la liste `T`. On se place dans le pire des cas.

- Première boucle `for` : i varie entre 0 et $n-1$. Pour chaque valeur de i on appelle la fonction `nombreZeros` (pour chaque valeur de j, il y a deux comparaisons, une incrémentation de la variable `nbr_zeros` et une incrémentation de la variable j qui varie entre i et $n-1$), soit $4(n-i)$ opérations élémentaires.

  On a donc $\sum_{i=0}^{n-1} 4(n-i) = (4n)\times n - 4\sum_{i=0}^{n-1} i = 4n^2 - 4\frac{n(n-1)}{2} = 2n^2 + 2n$ opérations élémentaires pour cette première boucle.
- Deuxième boucle `for` : i varie entre 1 et $n-1$. Pour chaque valeur de i, on a 2 opérations élémentaires (une comparaison et une affectation).

  On a $2(n-1)$ opérations élémentaires pour la deuxième boucle.

La complexité de la fonction `nombreZerosMax` est quadratique en $O(n^2)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

4. Pour un indice i de la liste `T`, on appelle `valeur` le nombre de zéros consécutifs dans `T` à partir de i en utilisant la fonction `nombreZeros`.

Si `valeur` est non nul, il est alors inutile d'appeler la fonction `nombreZeros(T, i+1)` mais on appelle `nombreZeros(T, i+valeur+1)`. Entre i et i + `valeur`, la liste `T` ne contient que des zéros et `T[i+valeur+1]` vaut nécessairement 1 puisque la fonction `nombreZeros` donne le nombre de zéros consécutifs, sauf si on est en fin de liste.

```
def nombreZerosMax2(T):
    i=0
    valeur_max=0
    while i<len(T):   # i ne doit pas dépasser len(T)-1
        valeur=nombreZeros(T,i)
        if valeur==0:
            i+=1
```

```
        else:
            i+=valeur+1
            if valeur>valeur_max:
                valeur_max=valeur
    return valeur_max
```

Remarque : On utilisera cette même technique d'optimisation dans l'exercice 5.3 « Recherche d'un mot dans un texte » du chapitre « Recherche séquentielle ».

Partie 3

Algorithmes

Plan

Recherche séquentielle 5

Exercice 5.1 : Recherche du minimum et du maximum d'une liste de nombres

On considère la liste de nombres : `L = [9, 10, 11, 56, 15, 16, 12, 18, 20, 12, -5, -8]`.

1. Écrire une fonction `rec_min` qui admet comme argument une liste non vide de nombres. Cette fonction retourne le minimum de cette liste. On n'utilisera pas la fonction `min`.

2. Écrire une fonction `rec_max` qui admet comme argument une liste non vide de nombres. Cette fonction retourne le maximum de cette liste. On n'utilisera pas la fonction `max`.

3. Évaluer la complexité de ces deux fonctions dans le cas le moins favorable.

Analyse du problème

On considère une liste non vide. On définit une variable `mini` définie par le premier élément de la liste. On parcourt la liste en comparant chaque élément de la liste à la variable `mini`.

1.

```
def rec_min(L):
    mini=L[0]   # ne pas utiliser la variable min
       # c'est une syntaxe Python pour avoir le minimum : min(L)
    for i in range(len(L)):
        if L[i]<mini:
            mini=L[i]
    return mini
```

Remarques :

- Ne pas utiliser « – » mais « _ » dans les noms de variables et de fonctions.
- La fonction `min(L)` de Python retourne le minimum d'une liste de valeurs.

2.

```
def rec_max(L):
    maxi=L[0]   # ne pas utiliser la variable max
                # c'est une syntaxe Python pour avoir
                # le maximum : max(L)
    for i in range(len(L)):
        if L[i]>maxi:
```

```
            maxi=L[i]
    return maxi
```

Remarque : La fonction `max(L)` de Python retourne le maximum d'une liste de valeurs.

3. On considère la fonction `rec_min`. On se place dans le pire des cas, c'est-à-dire une liste triée par ordre décroissant.

a. On cherche à calculer le nombre d'opérations élémentaires à chaque itération :

- Un test : 1 opération élémentaire.
- Une affectation : 1 opération élémentaire.

Le nombre d'opérations élémentaires vaut 2.

b. On a n itérations. Le nombre d'opérations élémentaires vaut $2n$.

La complexité est linéaire en $O(n)$ pour la fonction `rec_min`.

On obtient la même complexité pour la fonction `rec_max`.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

Exercice 5.2 : Assertion. Recherche du maximum, du second maximum d'un tableau

On considère le tableau de nombres :

```
import numpy as np
L=np.array([9, 10, 56, 28, -8])
```

1. Écrire une fonction `rec_max2` qui admet comme argument un tableau. Cette fonction retourne le maximum et le second maximum de ce tableau. On n'utilisera pas la fonction `max`. Utiliser une assertion pour vérifier le nombre d'éléments de `L`.

2. Écrire le programme principal permettant d'afficher le maximum et le second maximum du tableau `L`.

Analyse du problème

On considère un tableau non vide contenant au moins 2 éléments. On définit une variable `max1` définie par le premier élément de la liste ainsi que `indmax1` l'indice du premier maximum. On parcourt une fois le tableau en comparant chaque élément de celui-ci à la variable `max1`.

On parcourt une seconde fois le tableau pour déterminer le second maximum en s'assurant que l'indice du second maximum est différent de `indmax1`.

1.

```
def rec_max2(L):
    # la fonction retourne le maximum et le second maximum
    # du tableau ou de la liste L
    n=len(L)              # nombre d'éléments du tableau L
    assert n>=2
    # recherche du premier maximum
    indmax1=0
    max1=L[0]
    for i in range(1, n): # i varie entre 1 inclus et n exclu
        if L[i]>max1:
            max1=L[i]     # valeur du premier maximum
            indmax1=i     # indice du premier maximum
    # recherche du second maximum
    # l'indice du 2nd maximum est différent du 1er maximum
    if indmax1==0:        # indice du premier maximum = 0
        max2=L[1]
    else:               # indice du premier maximum différent de 0
        max2=L[0]
    for i in range(1, n): # i varie entre 1 inclus et n exclu
        if L[i]>max2 and i!=indmax1:
            # l'indice i doit être différent de indmax1
            max2=L[i]     # valeur du second maximum
    return max1, max2
```

Cette fonction convient aussi bien pour les listes que pour les tableaux.

Voir exercice 1.1 « Assertion, moyenne, variance et écart-type d'une liste de nombres » dans le chapitre « Prise en main de Python » pour l'utilisation de `assert`. Le nombre d'éléments du tableau doit être supérieur ou égal à 2.

2.

```
import numpy as np                # bibliothèque numpy renommée np
L=np.array([9, 10, 56, 28, -8])  # tableau de type ndarray
max1, max2=rec_max2(L)
print('Maximum de L :', max1)
print('Second maximum de L :', max2)
```

Exercice 5.3 : Recherche d'un mot dans un texte

1. Écrire une fonction `recherche_mot` qui admet comme arguments `texte` une chaîne de caractères et `mot` une chaîne de caractères. Cette fonction retourne `True` si `mot` est présent dans `texte`, `False` sinon, ainsi que l'indice de la première lettre de `mot` s'il est présent dans `texte`.

2. Écrire une fonction `recherche_mot_occurence` qui admet comme arguments `texte` une chaîne de caractères et `mot` une chaîne de caractères. Cette fonction retourne le nombre d'occurrences où `mot` est présent dans `texte` ainsi que la liste des indices de la première lettre des occurrences de `mot`.

3. Proposer une amélioration de la fonction précédente en optimisant la première boucle.

Analyse du problème

On parcourt la chaîne de caractères `texte` jusqu'à ce qu'on trouve le premier caractère de `mot`. On parcourt ensuite successivement tous les caractères de `mot` pour savoir s'ils sont présents les uns à la suite des autres. On utilise deux boucles imbriquées.

1. On considère la chaîne de caractères `texte = 'mots chaîne de caractères'` et `mot = 'caractères'`.

La longueur de `texte` vaut `len(texte) = 25`. La longueur de mot vaut `len(mot) = 10`.

On définit un indice *i* qui correspond à l'indice de la première lettre du mot s'il est présent dans `texte`.

Il est inutile de faire varier *i* entre 0 et 24 mais uniquement entre 0 et `len(texte) - len(mot) = 15`.

Lorsque *i* atteint 15, on a `texte[i] = 'c'` : c'est bien la première lettre du mot `'caractères'`.

Ensuite, il faut une deuxième boucle en faisant varier `j` entre les valeurs 0 et `len(mot) - 1` pour tester tous les caractères du mot.

```
def recherche_mot(texte, mot):
    # la fonction retourne True si mot(str) est présent dans
    # texte(str), False sinon, ainsi que l'indice de la première
    # lettre de mot s'il est présent dans la chaîne
    rep=False
    position=0
    i=0
    while i<=(len(texte)-len(mot)) and rep==False:
        j=0
        while j<=(len(mot)-1) and mot[j]==texte[i+j]:
            j+=1
        if j==len(mot):   # le mot est bien présent
            position=i
            rep=True
        i+=1
    return rep, position
```

2.

```
def recherche_mot_occurrence(texte, mot):
    # la fonction retourne le nombre d'occurrences où mot(str)
    # est présent dans texte(str) ainsi que la liste des indices
    # de la première lettre des occurrences de mot
    nbr_rep=0
    i=0
    liste=[]
    while i<=(len(texte)-len(mot)):
        j=0
        while j<=(len(mot)-1) and mot[j]==texte[i+j]:
            j+=1
        if j==len(mot):   # le mot est bien présent
            nbr_rep+=1
            liste.append(i)
```

```
        i+=1
    return nbr_rep, liste
```

3. On considère la chaîne de caractères `texte = 'mots chaîne de caractères'` et `mot = 'caractères'`. La longueur de `texte` vaut `len(texte)` = 25. La longueur de `mot` vaut `len(mot)` = 10. On définit un indice *i* qui correspond à l'indice de la première lettre de `mot` s'il est présent dans `texte`. On fait varier *i* entre 0 inclus et `len(texte) – len(mot)` = 15 inclus.

Ensuite, il faut une deuxième boucle en faisant varier *j* entre 0 inclus et `len(mot) – 1` inclus pour tester tous les caractères de `mot`. Lorsque la boucle *j* est terminée, on a fait le test `mot[j]==texte[i+j]`. *j* a été incrémenté de 1. Si `mot` a été trouvé, on peut incrémenter *i* de *j*+1.

```
def recherche_mot_occurrence2(texte, mot):
    # la fonction retourne le nombre d'occurrences où mot(str)
    # est présent dans texte(str) ainsi que la liste des indices
    # de la première lettre des occurrences de mot
    nbr_rep=0
    i=0
    liste=[]
    while i<=(len(texte)-len(mot)):
        j=0
        while j<=(len(mot)-1) and mot[j]==texte[i+j]:
            j+=1
        if j==len(mot):   # le mot est bien présent
            nbr_rep+=1
            liste.append(i)
        if j>0:
            j-=j
        i=i+j+1
    return nbr_rep, liste
```

Algorithmes de dichotomie

6

Exercice 6.1 : Recherche d'un élément dans un tableau non trié, algorithme naïf

On considère le tableau :

```
import numpy as np
L = np.array([16, 9, 11, 32, 15, 17, 18, 10, 25])
```

1. Écrire une fonction `rec_elt1` qui admet comme arguments un tableau `L` non trié et un élément `x`. Cette fonction retourne `True` si `x` est dans le tableau, `False` sinon.

2. Évaluer la complexité de cet algorithme pour un tableau de longueur *n* dans le cas le moins favorable.

3. Écrire une fonction `rec_elt2` qui admet comme arguments un tableau `L` et un élément `x`. Cette fonction retourne `True` si `x` est dans le tableau, `False` sinon ainsi que l'indice de l'élément recherché s'il est présent dans le tableau.

Analyse du problème

On parcourt le tableau numpy en partant du premier élément jusqu'à ce qu'on trouve `x`. On étudiera dans l'exercice suivant la méthode dichotomique, qui ne s'applique qu'aux tableaux triés.

1. Dans Python, les booléens vrai et faux s'écrivent `True` et `False`.

```
def rec_elt1(L, x):
    # la fonction retourne True si x est dans
    # le tableau ou la liste L, False sinon
    n=len(L)              # nombre d'éléments du tableau
    for i in range(n):    # i varie entre 0 inclus et n exclu
        if x==L[i]:       # teste si x=L[i]
            return True
    return False
```

Cette fonction convient aussi pour les listes et les chaînes de caractères.

2. On appelle *n* le nombre d'éléments du tableau *L*. On note $O(n)$ la complexité de la fonction `rec_elt1(L, x)`.

On cherche à calculer le nombre d'opérations élémentaires :

- 2 opérations élémentaires : appel du nombre d'éléments de *L* et affectation de *n*.
- Pour chaque étape de la boucle `for`, on a 2 opérations élémentaires : appel de l'élément L[i], comparaison.

La boucle `for` est exécutée *n* fois dans le pire des cas (si l'élément n'est pas présent dans le tableau par exemple).

Le nombre d'opérations élémentaires vaut : $2 + 2n$.

La complexité est linéaire en $O(n)$ pour la fonction `rec_elt1`.

3. On pourrait utiliser une boucle `for` au lieu d'une boucle `while`.

On utilise un indice *i* pour repérer la position dans le tableau. Les indices des tableaux commencent en 0 avec Python.

```
def rec_elt2(L, x):
    # la fonction retourne True si x est dans
    # le tableau (ou liste ou chaîne de caractères), False sinon
    # et l'indice de l'élément recherché s'il est présent
    # dans le tableau
    n=len(L)    # nombre d'éléments du tableau ou liste
                # ou chaîne de caractères
    i=0
    while i<n:
        if L[i]==x:
            return True, i
        i+=1
    return False
```

Exercice 6.2 : Recherche dichotomique dans un tableau trié

On considère le tableau trié :

```
import numpy as np
L=np.array([9, 10, 11, 15, 16, 17, 18, 25, 32])
```

1. Écrire une fonction `rec_dicho` qui admet comme arguments un tableau trié `L` et un élément `x`. Cette fonction retourne `True` si `x` est dans le tableau, `False` sinon ainsi que l'indice de l'élément recherché s'il est présent dans le tableau. On utilise la méthode dichotomique.

2. Évaluer la complexité de cet algorithme pour un tableau de longueur $n > 1$ dans le cas le moins favorable. On pourra considérer que l'entier *n* est une puissance de 2.

3. Comparer la complexité de l'algorithme naïf (voir exercice précédent « Recherche d'un élément dans un tableau non trié, algorithme naïf ») et de l'algorithme dichotomique.

Analyse du problème

La méthode dichotomique utilise le fait que le tableau numpy est trié. Elle consiste à comparer l'élément recherché à l'élément se trouvant au milieu d'un tableau trié. Comme le tableau est trié, cela permet d'éliminer une moitié du tableau comme emplacement possible de l'élément, sauf si on l'a déjà trouvé. Ensuite, on prend la moitié du tableau qui reste et on recommence…

Voir exercice 7.6 « Recherche dichotomique dans une liste triée – Version récursive » dans le chapitre « Récursivité » pour une version récursive de ce programme.

Cours :
La méthode dichotomique divise le problème initial et élimine une partie des données.

On verra la méthode générale « diviser pour régner » qui profite de la subdivision pour effectuer moins de calculs : exercice 7.2 « Exponentiation naïve, exponentiation rapide » dans le chapitre « Récursivité », et exercices 11.3 « Tri rapide » et 11.4 « Tri par partition-fusion » dans le chapitre «Tris ».

1.

```
def rec_dicho(L, x):
    # la fonction retourne True si x est dans
    # le tableau ou la liste L, False sinon, et l'indice
    # de l'élément recherché s'il est présent dans le tableau
    deb, fin=0, len(L)-1
    rep=False
    while fin>=deb and rep==False:
        milieu=(deb+fin)//2
        if x==L[milieu]:
            rep=True
        elif x>L[milieu]:   # x est dans la deuxième moitié
            deb=milieu+1
        else:               # x est dans la première moitié
            fin=milieu-1
    return rep, milieu
```

Exemple de fonctionnement de l'algorithme avec `x` = 9 :

- 1re itération : `deb` = 0, `fin` = 8 et `milieu` = 4 = `int`(8/2). On compare `x` avec `L[4]` = 16.
- 2e itération : `deb` = 0 et `fin` = `milieu` – 1 = 3. Le milieu vaut `int`(3/2) = 1. On compare `x` avec `L[1]` = 10.
- 3e itération : `deb` = 0 et `fin` = `milieu` – 1 = 0. On a trouvé l'élément `x`.

Exemple de fonctionnement de l'algorithme avec `x` = 25,5 :

- 1re itération : `deb` = 0, `fin` = 8 et `milieu` = 4 = `int`(8/2). On compare `x` avec `L[4]` = 16.
- 2e itération : `deb` = 5 = `milieu` +1 et `fin` = 8. Le milieu vaut `int`((5+8)/2) = 6. On compare `x` avec `L[6]` = 18.
- 3e itération : `deb` = `milieu` – 1 = 7 et `fin` = 8. Le milieu vaut `int`((7+8)/2) = 7. On compare `x` avec `L[7]` = 25.

- 4e itération : `deb` = 8 et `fin` = 8. Le milieu vaut 8. On compare `x` avec `L[8]` = 32. On a alors `fin` = 7.
- On n'a plus d'itération car `fin` < `deb`.

2. On se place dans le cas le moins favorable pour évaluer la complexité, c'est-à-dire dans le cas où l'élément n'est pas présent dans le tableau comportant n éléments.

a. On cherche à calculer le nombre d'opérations élémentaires à chaque itération :

- Calcul du milieu et une affectation : 2 opérations élémentaires.
- 1 test `if x==L[milieu]` : 1 opération élémentaire.
- 1 test `x>L[milieu]` : 1 opération élémentaire.
- Une affectation : 1 opération élémentaire.

Le nombre d'opérations élémentaires vaut 5.

b. On cherche à calculer le nombre d'itérations dans le pire des cas. Pour simplifier les calculs, on considère que l'entier n est une puissance de 2.

Pour chaque itération, on divise par deux la longueur du tableau.

Après une itération, la longueur du tableau est $n/2$. Après une deuxième itération, la longueur du tableau est $\frac{n}{2^2}$.

Après k itérations, la longueur du tableau est $\frac{n}{2^k}$. On arrive alors à un tableau de longueur 1.

On a donc $\frac{n}{2^k} = 1$, soit $\ln(n) = k\ln(2)$. Finalement, on obtient :

$$k = \frac{\ln(n)}{\ln(2)} = \log_2(n)$$

c. Le nombre d'opérations élémentaires vaut donc $5k$ dans le cas le moins favorable. Il est donc proportionnel à $\log_2(n)$.

La complexité est logarithmique en $O\left(\log_2(n)\right)$.

Remarque :

On peut noter également la complexité : $O\left(\log n\right)$.

On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

3. La complexité de l'algorithme naïf est linéaire alors que la complexité de l'algorithme dichotomique est logarithmique. On a une recherche d'un élément dans un tableau beaucoup plus rapide avec l'algorithme dichotomique pour des tableaux comportant un grand nombre d'éléments.

Exercice 6.3 : Recherche dichotomique du zéro d'une fonction

On considère une fonction f continue et strictement monotone sur l'intervalle $[a, b]$ telle que $f(a)f(b) < 0$. On cherche le zéro de cette fonction en utilisant la méthode de dichotomie.

La fonction `round(number, digits)` renvoie `number` arrondi avec une précision de `digits` chiffres après la virgule.

1. Écrire une fonction `rec_zero_dic` qui admet comme arguments la fonction `f`, `a`, `b` et `epsilon`. Cette fonction retourne l'abscisse `x` à `epsilon` près tel que $f(x) = 0$.

2. Écrire le programme principal permettant d'appliquer la recherche du zéro à la fonction sinus dans l'intervalle [3, 4] avec $\varepsilon = 10^{-7}$ et d'afficher le résultat avec une précision de 6 chiffres.

3. Évaluer la complexité de cet algorithme en fonction de `a`, `b` et `epsilon`.

Analyse du problème

On considère une fonction continue sur le segment $[a, b]$ telle que $f(a)f(b) < 0$: par le théorème des valeurs intermédiaires, l'équation $f(x) = 0$ admet au moins une solution sur l'intervalle $[a, b]$. Comme la fonction est strictement monotone sur le segment $[a, b]$, alors cette solution est unique sur le segment $[a, b]$. On utilise la méthode dichotomique pour rechercher cette solution.

On a déjà rencontré la méthode dichotomique dans l'exercice de recherche d'un élément dans un tableau trié.

On divise l'intervalle en deux et on calcule la valeur de la fonction au milieu de l'intervalle. On garde alors une moitié de cet intervalle. On réitère la procédure sur ce sous-intervalle.

On verra dans le chapitre 7 « Récursivité » une version récursive de ce programme.

Cours :

```
import numpy as np        # bibliothèque numpy renommée np
print(np.cos(np.pi/2))    # Python affiche 6.123233995736766e-17
```

Le nombre flottant ne permet pas un calcul exact à cause de la représentation des nombres à virgule avec la norme IEEE-754. Un test du type `a==b` n'a en général pas de sens si `a` et `b` sont des nombres à virgule flottante. On remplacera donc un tel test par une condition de la forme : `abs(a-b)<epsilon` où `epsilon` est une valeur proche de zéro, choisie en fonction du problème à traiter et de l'ordre de grandeur des erreurs auxquelles on peut s'attendre sur `a` et `b`.

1. On cherche une abscisse à ε (variable notée `epsilon`) près et non une ordonnée à ε près.

On ne fait donc pas le test `abs(f(x))<=epsilon` mais le test `(abs(b-a))<=epsilon`.

Dès qu'on a trouvé `a` et `b` tels que `(abs(b-a))<=epsilon`, la boucle `while` est terminée.

Il ne faut pas oublier la valeur absolue `np.abs` car on ne connaît pas le signe de `b-a`.

Le test `if f(milieu)*f(a)>0` permet de bien identifier l'intervalle à éliminer :

- si `f(milieu)` et `f(a)` ont le même signe, il faut éliminer l'intervalle `[a, milieu]` et ne garder que l'intervalle `[milieu, b]` ;
- sinon il faut éliminer l'intervalle `[milieu, b]` et ne garder que l'intervalle `[a, milieu]`.

```
import numpy as np       # bibliothèque numpy renommée np
def rec_zero_dic(f, a, b, epsilon):
    while np.abs(b-a)>epsilon:   # le test est à faire sur
                         # l'intervalle [a, b] et non sur f(milieu)
        milieu=(a+b)/2   # attention /2 et non pas //2
                         # comme avec les listes triées !
        if f(milieu)*f(a)>0:     # on peut avoir une fonction
                         # croissante ou décroissante
            a=milieu     # recherche dans l'intervalle [milieu, b]
        else:
            b=milieu     # recherche dans l'intervalle [a, milieu]
    return (a+b)/2       # on pourrait retourner a ou b
```

2. Si on applique la recherche du zéro par dichotomie à la fonction sin(x) dans l'intervalle [3, 4] avec $\varepsilon = 10^{-7}$, le programme suivant retourne `3.141593`.

```
import numpy as np
def f(x):  # fonction sin(x)
    return np.sin(x)
a=3
b=4
epsilon=1e-7
abszero=round(rec_zero_dic(f, a, b, epsilon), 6)
           # résultat arrondi avec une précision de 6 chiffres
print(abszero)
```

3.

a. On cherche à calculer le nombre d'opérations élémentaires à chaque itération :

- Un calcul du milieu et une affectation : 2 opérations élémentaires.
- Un test `if f(milieu)*f(a)>0` : 4 opérations élémentaires (deux calculs de la fonction, une multiplication et un test).
- Une affectation : 1 opération élémentaire.

Le nombre d'opérations élémentaires vaut 7.

b. On cherche à calculer le nombre d'itérations dans le pire des cas.
Pour chaque itération, on divise par deux la longueur de la liste.

Après une itération, la longueur de l'intervalle est $\frac{(b-a)}{2}$. Après une deuxième itération, $\frac{(b-a)}{2^2}$.

Après k itérations, la longueur de l'intervalle est $\frac{(b-a)}{2^k}$.

On quitte les itérations lorsque $\frac{(b-a)}{2^k} < \varepsilon$. On a donc $2^k > \frac{b-a}{\varepsilon}$, soit $k \ln 2 > \ln\left(\frac{b-a}{\varepsilon}\right)$.

Finalement, on obtient : $k > \log_2\left(\frac{b-a}{\varepsilon}\right)$.

c. Le nombre d'opérations élémentaires vaut donc $7k$. Il est donc proportionnel à $\log_2\left(\frac{b-a}{\varepsilon}\right)$.

La complexité est en $O\left(\log_2\left(\frac{b-a}{\varepsilon}\right)\right)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

Partie 4

Récursivité

Plan

Récursivité

7

Exercice 7.1 : Factorielle d'un entier naturel

On souhaite calculer la factorielle d'un entier naturel *n*.

1. Écrire une fonction `fact` qui admet comme argument un entier naturel `n` et qui retourne la valeur de la factorielle de `n` en utilisant un programme itératif.

2. Écrire une fonction `fact_rec` qui admet comme argument un entier naturel `n` et qui retourne la valeur de la factorielle de `n` en utilisant un programme récursif.

3. Démontrer la terminaison pour la fonction `fact_rec`.

4. Représenter les différentes activations de la fonction récursive `fact_rec(3)` sous la forme d'un arbre.

5. Démontrer la correction de la fonction `fact_rec`.

6. Évaluer la complexité de la fonction `fact_rec`.

7. Que se passe-t-il si on exécute `fact_rec(-3)` ?

Analyse du problème

On va étudier la différence entre une fonction itérative et une fonction récursive. Les méthodes mises en place dans cet exercice seront utilisées dans de très nombreux exercices concernant les fonctions récursives.

Cours :

Une fonction est récursive lorsque le corps de cette fonction fait appel à cette même fonction (c'est une fonction qui s'appelle elle-même). Sinon, on dit que cette fonction est itérative (elle peut être constituée de boucles `while` ou `for`).

Il est essentiel de prévoir qu'une procédure récursive se termine ! L'instruction `return` doit être présente au moins deux fois :

- une fois pour la condition d'arrêt (premier `return` dans le programme) ;
- une autre fois pour l'appel récursif (dernier `return` dans le programme).

1.

```
def fact (n):
    # la fonction renvoie la factorielle d'un entier naturel n
    # programme itératif
    res=1
    for i in range(1, n+1):  # i varie entre 1 inclus et n+1 exclu
```

```
        res=res*i
    return res
```

2.

```
def fact_rec(n):
    # la fonction renvoie la factorielle d'un entier naturel n
    # programme récursif
    if n==0:
        return (1)                  # condition d'arrêt
    else:
        return (n*fact_rec(n-1))    # rappel récursif
```

Cours :

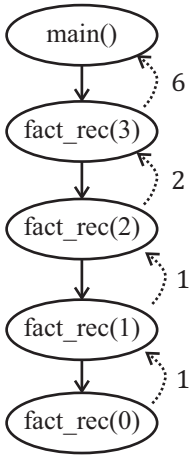
Pour démontrer la terminaison d'un programme, on cherche une grandeur positive, que l'on appelle variant de boucle, qui décroît entre deux appels de la fonction récursive et qui converge vers une valeur d'un cas correspondant à un appel de la condition d'arrêt.

De façon générale, il faut montrer que l'on arrivera en un nombre fini d'étapes à un appel de la condition d'arrêt.

3. On considère le variant de boucle n. À chaque appel de la fonction récursive, il décroît d'une unité et finit par atteindre la valeur 0 correspondant à une condition d'arrêt. Le programme se termine donc dans tous les cas si $n \geq 0$.

4. L'arbre ci-dessous représente les différents appels de la fonction `fact_rec(3)`.

Les différents appels de la fonction récursive sont stockés dans une pile : c'est la phase de descente. Quand on atteint la condition d'arrêt, on passe à la phase de remontée et les appels sont désempilés jusqu'à retourner à l'appel initial.



Au dernier appel de la fonction récursive, $n = 0$. La condition d'arrêt est vérifiée. On passe à la phase de remontée.

Remarque : Dans l'exercice 11.4 « Tri par partition-fusion » dans le chapitre « Tris », l'arbre des appels de la fonction récursive ressemble encore plus à un « arbre » !

Pour calculer la factorielle de 3, on a deux phases :

- Phase de descente : On a des appels successifs de la fonction `fact_rec` jusqu'à ce que l'on arrive au cas $n = 0$.
- Phase de montée : Le programme retourne les valeurs des appels successifs précédents. On remonte jusqu'à l'appel initial et le programme retourne le résultat.

Remarque :

Dans la phase de descente, les appels successifs sont stockés dans une pile.

Une fois que la condition d'arrêt est obtenue, les appels sont ensuite désempilés jusqu'à arriver à l'appel initial dans la phase de montée.

Le nombre d'appels récursifs est limité à 1 000 avec Python. Tout dépassement provoquera une erreur.

Pour calculer la factorielle de n, on applique la fonction `fact_rec` à plusieurs sous-problèmes. Cette méthode de décomposition/recomposition est appelée **diviser pour régner**. On utilisera cette méthode dans les algorithmes de tri.

Cours :
Pour démontrer la correction d'un programme, il faut montrer que l'algorithme effectue bien la tâche souhaitée. On utilise souvent une démarche proche du raisonnement par récurrence.

On établit une propriété (appelée invariant de boucle) P_n :

- la propriété P_n doit être vraie pour $n = 0$;
- si P_n est vraie, alors P_{n+1} doit être vraie.

5. On considère la propriété (appelée invariant de boucle) P_n : « La fonction `fact_rec(n)` retourne $n!$ ».

- P_0 est vrai puisque c'est la condition d'arrêt.
- Supposons que la propriété P_n est vraie. La fonction `fact_rec(n+1)` réalise l'opération : $(n+1) \times \text{fact\_rec}(n)$. Comme P_n est vrai, alors le programme retourne : $(n+1) \times \text{fact\_rec}(n) = (n+1)n! = (n+1)!$. La propriété P_{n+1} est donc vraie.

On a démontré par récurrence que la propriété P_n est vraie pour tout entier naturel n. La correction de la fonction `fact_rec` est donc démontrée.

Remarque :

On distingue parfois correction partielle et correction totale :

- La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête.
- La correction est totale si elle est partielle et si l'algorithme termine.

Cours :
La complexité est une mesure du nombre d'opérations élémentaires que l'algorithme effectue. On évalue la complexité d'une fonction récursive à partir d'une relation de récurrence.

6. On définit $T(n)$ la complexité de la fonction `fact_rec(n)`.
À chaque appel de la fonction récursive, on a 2 opérations élémentaires :
- un test pour savoir si $n = 0$;
- un calcul : `n*fact_rec(n-1)`.

On en déduit la relation de récurrence : $T(n) = T(n-1) + 2$ avec $T(0) = 1$.
On a donc : $T(0) = 1$, $T(1) = 1 + 2$, $T(2) = 1 + 2 + 2$, soit $T(n) = 1 + 2n$.
La complexité de la fonction `fact_rec(n)` est linéaire en $O(n)$.

Remarque :

On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

La complexité d'une fonction récursive est souvent beaucoup plus importante (voir exercice 8.1 « Suite des nombres de Fibonacci » dans le chapitre « Algorithmes gloutons ») que la complexité de la fonction itérative.

On peut donc être amené à dérécursiver un programme, en faisant le contraire de ce que fait le programme récursif. Le programme itératif `fact` commence par la plus petite valeur 1 alors que le programme récursif `fact_rec` commence par la plus grande valeur n.

7. Si on exécute le programme suivant : `print(fact_rec(-3))`, Python affiche le message d'erreur : `#RecursionError: maximum recursion depth exceeded in comparison`.
On a vu dans la question 4 que les différents appels de la fonction récursive sont stockés dans une pile. Comme $n < 0$, on n'arrête pas d'appeler la fonction récursive. On arrive alors à un dépassement de la taille de la pile et le programme Python s'arrête et renvoie un message d'erreur.

Exercice 7.2 : Exponentiation naïve, exponentiation rapide

On souhaite calculer la puissance entière d'un nombre réel.

1. Écrire une fonction `puiss` qui admet comme arguments un nombre réel `x`, un entier strictement positif `n` et qui retourne x^n en utilisant l'exponentiation « naïve », c'est-à-dire en multipliant n fois par lui-même x.

2. Écrire une fonction `puiss_rec` qui admet comme arguments un nombre réel `x`, un entier strictement positif `n` et qui retourne x^n en utilisant un programme récursif. Écrire le programme principal qui demande à l'utilisateur de saisir au clavier x et n, et qui affiche x^n.

3. Démontrer la terminaison de la fonction `puiss_rec`.

4. Démontrer la correction de la fonction `puiss_rec`.

5. Évaluer la complexité de la fonction `puiss_rec`. On pourra considérer que l'entier n est une puissance de 2.

6. On souhaite améliorer la complexité de la fonction `puiss_rec` en utilisant les propriétés : $x^0 = 1$; $x^{2p} = \left(x^p\right)^2$ et $x^{2p+1} = x\left(x^p\right)^2$.

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

Évaluer la complexité de la fonction `puiss_rapide`.

7. Écrire une fonction `puiss_rapide2` permettant d'optimiser la fonction `puiss_rapide` avec un seul appel à la fonction récursive dans le corps de la fonction. Évaluer la complexité de la fonction `puiss_rapide2`. Pourquoi utilise-t-on le terme « exponentiation rapide » ? Pourquoi l'algorithme utilise la méthode « diviser pour régner » ?

Analyse du problème

Une fonction est récursive lorsque le corps de cette fonction fait appel à cette même fonction. On va étudier plusieurs améliorations pour calculer la puissance d'un nombre réel.

1.

```
def puiss(x, n):
    # la fonction renvoie x**n avec x réel et n entier > 0
    res=1
    for i in range(n):  # i varie entre 0 inclus et n exclu
        res=res*x
    return res
```

2.

```
def puiss_rec(x, n):
    # la fonction renvoie x**n avec x réel et n entier > 0
    # calcule x**n avec x réel et n>0
    if n==0:
        return 1  # condition d'arrêt avec x**0=1
    else:
        return (x*puiss_rec(x, n-1))  # appel récursif
```

Remarque :

Il est essentiel de prévoir qu'une procédure récursive se termine ! L'instruction `return` doit être présente au moins deux fois :

- une fois pour la condition d'arrêt (premier `return` dans le programme),
- une autre fois pour l'appel récursif (dernier `return` dans le programme).

```
x=float(input('Entrez un réel x : '))
n=int(input('Entrez un entier positif n : '))
print('Le résultat x**n = ', puiss_rec(x, n))
```

3. On considère le variant de boucle n. À chaque appel de la fonction récursive `puiss_rec`, il décroît d'une unité et finit par atteindre la valeur 0 correspondant à une condition d'arrêt. Le programme se termine donc dans tous les cas si $n \geq 0$.

Remarque : Le programme ne se termine pas si l'entier n est négatif !

4. On considère la propriété (appelée invariant de boucle) P_n : « La fonction `puiss_rec(n)` retourne $n!$ ».

- P_0 est vraie puisque c'est la condition d'arrêt.
- Supposons que la propriété P_n est vraie. La fonction `puiss_rec(n+1)` réalise l'opération : $x \times \text{puiss\_rec}(n)$. Comme P_n est vrai, alors le programme retourne : $x \times \text{puiss\_rec}(n) = xx^n = x^{n+1}$. La propriété P_{n+1} est donc vraie.

On a démontré par récurrence que la propriété P_n est vraie pour tout entier naturel n. La correction de la fonction `puiss_rec` est donc démontrée.

Remarque :

On distingue parfois correction partielle et correction totale :

- La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête.
- La correction est totale si elle est partielle et si l'algorithme termine.

5. On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `puiss_rec`.

À chaque appel de la fonction récursive, on a :

- un test pour savoir si $n = 0$,
- un calcul : `x*puiss_rec(x, n-1)`.

On en déduit la relation de récurrence :

$$T(n) = T(n-1) + 2 \text{ avec } T(0) = 1$$

On a donc : $T(0) = 1$, $T(1) = 1 + 2$, $T(2) = 1 + 2 + 2$, soit $T(n) = 1 + 2n$.

La complexité de la fonction `puiss_rec(n)` est linéaire en $O(n)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

6. On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `puiss_rapide`.

À chaque appel de la fonction récursive, on a :

- un test pour savoir si $n = 0$,
- un test pour savoir si n est pair,
- deux appels de la fonction récursive pour calculer le carré : `puiss_rapide (x, n//2) * puiss_rapide (x, n//2)`,
- une ou deux opérations élémentaires.

On se place dans le pire des cas. On en déduit la relation de récurrence :

$$T(n) = 2T\left(\frac{n}{2}\right) + 4$$

Pour simplifier les calculs, on considère que l'entier n est une puissance de 2.

Il faut calculer le nombre de fois où on fait un appel de la fonction récursive.

- Après un appel de la fonction récursive, on l'appelle à nouveau avec $n/2$.
- Après un deuxième appel de la fonction récursive, on l'appelle à nouveau avec $(n/2)/2$, soit $n/(2^2)$.
- Après k appels de la fonction récursive, on l'appelle à nouveau avec $n/(2^k)$.

On n'a plus d'appel de la fonction récursive quand $\frac{n}{2^k} = 1$, soit $\ln(n) = k\ln(2)$.

Finalement, on obtient :

$$k = \frac{\ln(n)}{\ln(2)} = \log_2(n)$$

Par exemple pour $n = 2^4$, on a :

$$T(2^4) = 2T(2^{4-1}) + 4 \; ; \; T(2^3) = 2T(2^2) + 4 \; ;$$
$$T(2^2) = 2T(2^1) + 4 \; ; \; T(2^1) = 2T(2^0) + 4 = 2 \times 1 + 4$$

On considère la suite définie par récurrence : $u_k = au_{k-1} + b$ avec $a = 2$, $b = 4$ et $u_0 = 1$. On pose : $r = \frac{b}{1-a}$.

D'après l'énoncé, on a :

$$T\left(n = 2^k\right) = u_k = a^k\left(u_0 - r\right) + r = 2^k\left(1 - \frac{4}{1-2}\right) + \frac{4}{1-2} = 5n - 4.$$

La complexité est linéaire en $O(n)$.

Pour calculer `puiss_rapide(x, n//2)**2`, Python fait l'opération suivante : `puiss_rapide(x, n//2)*puiss_rapide(x, n//2)`. Il fait appel deux fois à la fonction récursive dans le corps de la fonction ! La fonction `puiss_rapide` n'a pas amélioré la complexité.

7. On peut optimiser la fonction `puiss_rapide` avec un seul appel à la fonction récursive :

```
def puiss_rapide2(x, n):
    # la fonction renvoie x**n avec x réel et n entier > 0
    if n==0:
        return 1        # condition d'arrêt
    else:
        res=puiss_rapide2(x, n//2)
        if n%2==0:      # n est pair
            return (res**2)
        else:           # n est impair
            return (x*(res**2))
```

On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `puiss_rapide2`.

À chaque appel de la fonction récursive, on a :

- un test pour savoir si $n = 0$,
- un appel de la fonction récursive : `puiss_rapide2`,
- un test pour savoir si n est pair,
- un calcul : `res**2` ou `x*(res**2)`.

On se place dans le pire des cas. On en déduit la relation de récurrence :

$$T(n) = T\left(\frac{n}{2}\right) + 4$$

Pour simplifier les calculs, on considère que l'entier n est une puissance de 2.

On a le même nombre d'appels k de la fonction récursive que dans la question précédente :

$$k = \frac{\ln(n)}{\ln(2)} = \log_2(n)$$

Par exemple pour 2^4, on a :

$$T(2^4) = T(2^{4-1}) + 4 \; ; \; T(2^3) = T(2^2) + 4 \; ;$$

$$T(2^2) = T(2^1) + 4; \; T(2^1) = T(2^0) + 4 = 1 + 4$$

Finalement, on a : $T(2^k) = T(2^4) = 4k + 1$ avec $k = 4$.

La complexité de la fonction `puiss_rapide2` est logarithmique en $O\left(\log_2(n)\right)$.

Explications du terme « exponentiation rapide » :

- « exponentiation » : calcul de la puissance entière d'un nombre réel ;
- « rapide » : la fonction `puiss_rapide2` est plus rapide pour des entiers $n > 1$ que `puiss`, `puiss_rec` et `puiss_rapide` puisque la complexité est logarithmique pour `puiss_rapide2` alors qu'elle est linéaire pour `puiss`, `puiss_rec` et `puiss_rapide`.

On utilise la méthode « diviser pour régner » qui se décompose en trois étapes :

- Diviser (ou partitionner) : on divise le problème initial en plusieurs sous-problèmes.
- Régner : on traite récursivement chacun des sous-problèmes.
- Combiner : on combine les différents sous-problèmes pour résoudre le problème de départ.

Remarques :

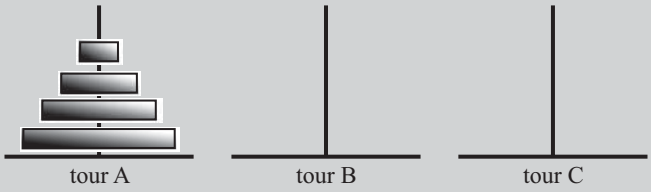
- La méthode « diviser pour régner » profite de la subdivision pour effectuer moins de calculs (voir les exercices 11.3 « Tri rapide » et 11.4 « Tri par partition-fusion » dans le chapitre «Tris »).
- La méthode dichotomique divise uniquement le problème initial et élimine une partie des données (voir exercice 6.2 « Recherche dichotomique dans un tableau trié » dans le chapitre « Algorithmes de dichotomie »).

Exercice 7.3 : Tours de Hanoï

On considère n disques de diamètre différent empilés par ordre décroissant sur une tour de départ (tour A sur la figure ci-dessous). Les deux autres tours n'ont pas de disque. L'objectif est de déplacer les disques de la tour A (tour de départ) vers la tour C (tour d'arrivée) en utilisant les deux règles suivantes :

- on ne peut déplacer qu'un disque à la fois ;
- on ne peut placer un disque que sur un disque de diamètre plus grand ou sur un emplacement vide.

On considère la suite définie par récurrence : $u_{n+1} = au_n + b$. On pose : $r = \dfrac{b}{1-a}$.
On admet que : $u_n = a^n\left(u_0 - r\right) + r$.
On définit la liste `tour` : `tour[0]` est une liste représentant les disques de la tour A, `tour[1]` (respectivement `tour[2]`) représente les disques de la tour B (respectivement tour C).
Par exemple, pour $n = 4$, on a : `tour` = [[4, 3, 2, 1], [ ], [ ]].



1. Montrer que l'on peut résoudre le problème avec 1 tour.

2. On va utiliser un programme récursif pour résoudre le problème. On suppose que l'on sait procéder pour $n-1$ tours. Montrer avec un schéma que l'on peut résoudre le problème avec n tours. On pourra prendre $n = 4$.

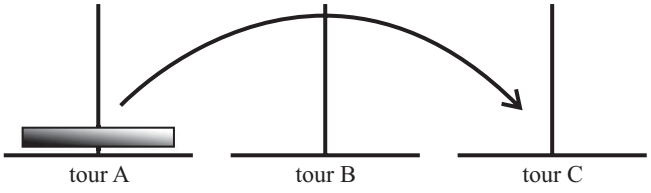
3. Écrire une fonction récursive `hanoi(tour, n, a, b, c)` qui admet comme arguments `tour` la liste décrite précédemment, `n` le nombre de disques, `a` la tour de départ, `b` la tour intermédiaire et `c` la tour d'arrivée. Dans l'exemple précédent, la fonction `hanoi(tour, 4, 0, 1, 2)` doit retourner `[[], [], [4, 3, 2, 1]]`. Le programme affichera, étape par étape, la liste *tour*.

4. Évaluer la complexité de la fonction `hanoi`.

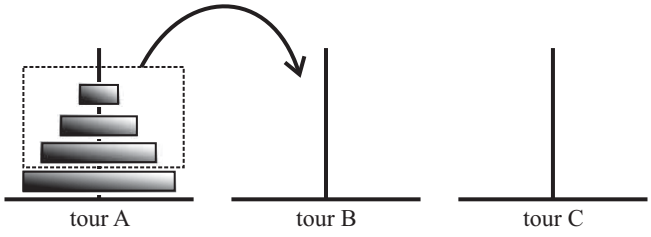
Analyse du problème

L'utilisation d'une fonction récursive permet de résoudre facilement le problème de Hanoï. La question 2 permet de comprendre la fonction récursive `hanoi`. Les différents schémas montrent comment déplacer 4 tours sachant que l'on sait résoudre le problème pour 3 tours.

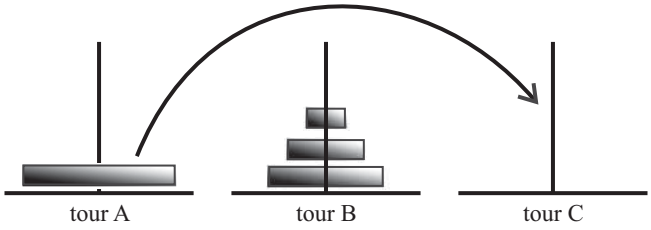
1. La résolution du problème est évidente pour $n = 1$ puisqu'il suffit de dépiler le disque de la tour A et de l'empiler dans la tour C.



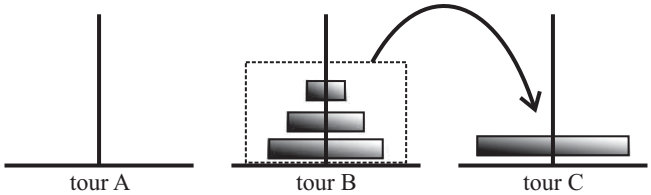
2. On souhaite déplacer n disques de la tour A vers la tour C. On considère $n = 4$ dans l'exemple suivant.



On considère $n-1 = 3$ disques. D'après l'énoncé, on sait déplacer les $n-1 = 3$ disques (en pointillés sur le schéma ci-dessus) de la tour A vers la tour B. On obtient alors la configuration suivante :



On déplace le disque restant de la tour A vers la tour C.
On obtient alors :



Il reste à déplacer les $n-1 = 3$ disques (en pointillés sur le schéma ci-dessus) de la tour B vers la tour C.

3. On met en place un programme récursif. On a vu dans la question précédente que l'on pouvait déplacer les 4 disques à condition de savoir déplacer 3 disques. Pour déplacer les 3 disques d'une tour vers une autre, on applique le programme récursif à 2 disques. Pour déplacer les 2 disques, on applique le programme récursif à 1 disque que l'on sait déplacer.

```
def hanoi(tour, n, a, b, c):
    # a : tour de départ (a peut être égal à 0, 1 ou 2)
    # b : tour intermédiaire (b peut être égal à 0, 1 ou 2)
    # c : tour d'arrivée (c peut être égal à 0, 1 ou 2)

    # tour[i] = liste représentant les disques de la tour i
    if n==1:   # un seul disque à déplacer
        disque=tour[a].pop()   # dépile le disque de la tour a
        tour[c].append(disque)   # empile le disque dans la tour c
        print(tour)   # affichage de la liste tour étape par étape
    else:
        # déplacer n-1 disques de a vers b
        hanoi(tour, n-1, a, c, b)
            # tour de départ : a
            # tour intermédiaire : c
            # tour finale : b
        print(tour)   # affichage de la liste tour étape par étape
```

```
        # déplacer le disque restant de a vers c
        disque=tour[a].pop()   # dépile le disque de la tour a
        tour[c].append(disque)   # empile le disque dans la tour c
        print(tour)   # affichage de la liste tour étape par étape
        # déplacer n-1 disques de b vers c
        hanoi(tour, n-1, b, a, c)
        print(tour)
            # tour de départ : b
            # tour intermédiaire : a
            # tour finale : c
    return  # inutile de retourner tour car passage par adresse
            # pour les listes

tour=[[4, 3, 2, 1], [], []]
n=len(tour[0])   # nombre d'éléments de la tour 0
print(tour)

hanoi(tour, n, 0, 1, 2)   # on veut déplacer les n disques
                          # de 0 vers 2
print(tour)
```

4. On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `hanoi`. On se place dans le pire des cas.

À chaque appel de la fonction récursive, on a :

- un test pour savoir si $n = 1$;
- un appel à la fonction `hanoi` avec $n-1$;
- une opération élémentaire pour dépiler ;
- une opération élémentaire pour empiler ;
- un appel à la fonction `hanoi` avec $n-1$.

On en déduit la relation de récurrence :

$$T(n) = 2T(n-1) + 3 \text{ avec } T(0) = 0$$

On considère la suite définie par récurrence : $u_{n+1} = au_n + b$ avec $a = 2$ et $b = 3$.

On pose $r = \dfrac{b}{1-a} = \dfrac{3}{1-a} = -3$. D'après l'énoncé, on a : $u_n = a^n\left(u_0 - r\right) + r$.

On en déduit que :

$$T(n) = 2^n(0+3) - 3$$

La complexité de la fonction `hanoi` est exponentielle en $O(2^n)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

Exercice 7.4 : Recherche dichotomique du zéro d'une fonction – Version récursive

On considère une fonction f continue et strictement monotone sur $[a,b]$ telle que $f(a)f(b)<0$. On cherche le zéro de cette fonction en utilisant la méthode par dichotomie.

1. Écrire une fonction récursive `rec_zero_dic_recursive` qui admet comme arguments la fonction `f`, `a`, `b` et `epsilon`. Cette fonction retourne l'abscisse `x` à `epsilon` près telle que $f(x)=0$.

2. Écrire le programme principal permettant d'appliquer la recherche du zéro à la fonction sinus sur l'intervalle $[a, b]$ avec `a` = 3.1, `b` = 3.2 et `epsilon` = 0.05.

3. Représenter les différentes activations de la fonction récursive `rec_zero_dic_recursive` sous la forme d'un arbre avec les valeurs numériques précédentes.

4. Évaluer la complexité de cet algorithme en fonction de `a`, `b` et `epsilon`.

Analyse du problème

On considère une fonction continue sur le segment $[a, b]$, telle que $f(a)f(b)<0$: par le théorème des valeurs intermédiaires, l'équation $f(x)=0$ admet au moins une solution sur l'intervalle $[a, b]$. Comme la fonction est strictement monotone sur le segment $[a, b]$, alors cette solution est unique sur le segment $[a, b]$. On utilise la méthode par dichotomie pour rechercher cette solution.

On a vu, dans le chapitre 5 « Recherche séquentielle », une version itérative de ce programme.

On divise l'intervalle en deux et on calcule la valeur de la fonction au milieu de l'intervalle. On garde alors une moitié de cet intervalle. On réitère la procédure sur ce sous-intervalle.

1.

```
def rec_zero_dic_recursive (f, a, b, epsilon):
    # la fonction renvoie l'abscisse x à epsilon près
    # telle que f(x)=0
    # recherche de x sur l'intervalle [a, b] avec
    # f strictement monotone
    if np.abs(b-a)<=epsilon:
        return(((a+b)/2))   # condition d'arrêt
    else:
        milieu=(a+b)/2
        if f(milieu)*f(a)>0:
            # recherche dans l'intervalle [milieu, b]
            return rec_zero_dic_recursive (f, milieu, b, epsilon)
            # appel récursif
        else:
            # recherche dans l'intervalle [a, milieu]
            return rec_zero_dic_recursive (f, a, milieu, epsilon)
            # appel récursif
```

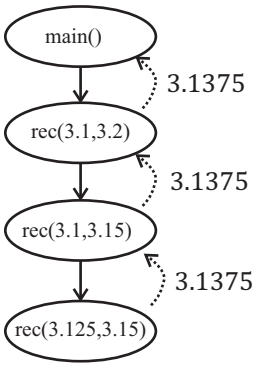
2. Le programme suivant permet de calculer le zéro de la fonction sinus entre 3.1 et 3.2 :

```
import numpy as np
def f(x):
    return np.sin(x)
a=3.1
b=3.2
epsilon=0.05
print(rec_zero_dic_recursive(f, a, b, epsilon))
```

Le programme retourne la valeur 3.1375.

3. L'arbre ci-dessous représente les différents appels de la fonction `rec_zero_dic_recursive`, que l'on note `rec` ci-dessous. Les deux valeurs correspondent aux valeurs de a et b lors des appels successifs.

Les différents appels de la fonction récursive sont stockés dans une pile : c'est la phase de descente. Quand on atteint la condition d'arrêt, on passe à la phase de remontée et les appels sont désempilés jusqu'à retourner à l'appel initial.



Au dernier appel de la fonction récursive, on a $a = 3{,}125$ et $b = 3{,}15$. On a alors $\varepsilon = b - a = 0{,}03 \leq 0{,}05$. La condition d'arrêt est vérifiée. On passe à la phase de remontée avec la valeur $\frac{a+b}{2} = 3{,}1375$.

4. On définit $T(b-a)$ le nombre d'opérations élémentaires de la fonction `rec_zero_dic_recursive`.

À chaque appel de la fonction récursive, on a :

- un test ;
- un calcul du milieu et une affectation : 2 opérations élémentaires ;
- un test `if f(milieu)*f(a) > 0` : 3 opérations élémentaires (deux calculs de la fonction et test) ;
- un appel de la fonction récursive avec $\frac{b-a}{2}$.

On se place dans le pire des cas. On en déduit la relation de récurrence :

$$T(b-a) = T\left(\frac{b-a}{2}\right) + 6$$

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

On cherche à calculer le nombre d'itérations dans le pire des cas.
Pour chaque itération, on divise par 2 la longueur de la liste.
Après une itération, la longueur de l'intervalle est $\frac{(b-a)}{2}$. Après une deuxième itération, $\frac{(b-a)}{2^2}$.
Après k itérations, la longueur de l'intervalle est $\frac{(b-a)}{2^k}$.
On quitte les itérations lorsque $\frac{(b-a)}{2^k} < \varepsilon$. On a donc $2^k > \frac{b-a}{\varepsilon}$, soit $k \ln 2 > \ln\left(\frac{b-a}{\varepsilon}\right)$.
Finalement, on obtient : $k > \log_2\left(\frac{b-a}{\varepsilon}\right)$.
Le nombre d'opérations élémentaires vaut donc $6k$. Il est donc proportionnel à $\log_2\left(\frac{b-a}{\varepsilon}\right)$.
La complexité est en $O\left(\log_2\left(\frac{b-a}{\varepsilon}\right)\right)$.

Remarque : On a la même complexité que pour la version itérative.

Exercice 7.5 : Recherche du pgcd (CCP MP Maths 2016)

Cet exercice étudie deux algorithmes permettant le calcul du pgcd (plus grand commun diviseur) de deux entiers naturels.

1. Pour calculer le pgcd de 3 705 et 513, on peut passer en revue tous les entiers 1, 2, 3, ⋯ , 512, 513 puis renvoyer parmi ces entiers le dernier qui divise à la fois 3 705 et 513. Il sera alors le plus grand des diviseurs communs à 3 705 et 513. Écrire une fonction `gcd` qui renvoie le pgcd de deux entiers naturels non nuls, selon la méthode décrite ci-dessus. On pourra éventuellement utiliser librement l'instruction `min(a, b)`, qui calcule le minimum de `a` et `b`. Par exemple `gcd(3705, 513)` renverra 57.

2. Écrire une fonction `euclide` permettant de calculer le pgcd de deux entiers naturels selon l'algorithme d'Euclide :

- Pour $b = 0$: $\mathbf{pgcd}(a, 0) = a$;
- Pour $b \neq 0$, on note r le reste de la division euclidienne de a par b, alors $\mathbf{pgcd}(a, b) = \mathbf{pgcd}(b, r)$.

3. Écrire une fonction récursive `euclide_rec` qui calcule le pgcd de deux entiers naturels selon l'algorithme d'Euclide.

4. On note $\left(F_n\right)_{n\in\mathbb{N}}$ la suite des nombres de Fibonacci définie par :

$$F_0 = 0, F_1 = 1, \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$$

a. Écrire les divisions euclidiennes successivement effectuées lorsque l'on calcule le pgcd de $F_6 = 8$ et $F_5 = 5$ avec la fonction `euclide`.

b. Soit $n \geq 2$ un entier. Quel est le reste de la division euclidienne de F_{n+2} par F_{n+1} ? On pourra utiliser librement le fait que la suite $(F_n)n \in \mathbb{N}$ est strictement croissante à partir de $n = 2$. En déduire, sans démonstration, le nombre u_n de divisions euclidiennes effectuées lorsque l'on calcule le pgcd de F_{n+2} et F_{n+1} avec la fonction `euclide`.

5. Écrire une fonction `fibo` qui prend en argument un entier naturel n et renvoie le nombre de Fibonacci F_n. Par exemple, `fibo(6)` renverra 8.

6. Écrire une fonction récursive `fibo_rec` qui permet de renvoyer le nombre de Fibonacci.

7. En utilisant la fonction `euclide`, écrire une fonction `gcd_trois` qui renvoie le pgcd de trois entiers naturels. Par exemple, `gcd_trois(18, 30, 12)` renverra 6.

Analyse du problème

On étudie dans cet exercice l'algorithme d'Euclide permettant de calculer le pgcd de deux entiers naturels. Cet exercice est extrait du concours CCP MP Maths 2016.

1. On parcourt tous les entiers i entre 1 et le minimum de a et b. On fait un test pour savoir si i est un diviseur de a et b. Le test peut se faire en calculant le reste. Si le reste est nul, i est un diviseur.

```
def gcd(a, b):
    # la fonction renvoie le pgcd de deux entiers naturels
    # non nuls a et b
    pgcd=1
    for i in range(1, min(a,b)+1):
        if a%i==0 and b%i==0:
            # si le reste est nul alors i est un
            # diviseur de a et b
            pgcd=i
    return pgcd
```

Remarque : Il est inutile de faire varier i entre 1 et le maximum de a et b.

2.

```
def euclide(a, b):
    # la fonction renvoie le pgcd de deux entiers naturels
    # algorithme d'Euclide
    while b!=0:
```

```
        r=a%b
        a,b=b,r
    return a
```

Cours :
Dans toute procédure récursive, l'instruction `return` doit être présente au moins deux fois : une fois pour la condition d'arrêt (premier `return` dans le programme) et une autre fois pour l'appel récursif (dernier `return` dans le programme).

3.

```
def euclide_rec(a, b):
    # la fonction renvoie le pgcd de deux entiers naturels
    # algorithme d'Euclide
    if b==0:   # condition d'arrêt
        return a
    else:
        return euclide_rec(b,a%b)   # appel récursif
```

Remarque :

Les appels successifs d'une fonction récursive sont stockés dans une pile.

Prenons l'exemple suivant : **pgcd**(16,12).

Appel de `euclide_rec(12,4)`

Appel de `euclide_rec(4,0)`

Retour de 4

Retour de 4

4. a.

- 1er appel de la boucle : $a = F_6 = 8$ et $b = F_5 = 5$.
  Le reste de la division euclidienne de 8 par 5 vaut 3. Le quotient vaut 1.
  On a $F_4 = 3$.
  $a = 5$ et $b = 3$.
- 2e appel de la boucle : $a = F_5 = 5$ et $b = F_4 = 3$.
  Le reste de la division euclidienne de 5 par 3 vaut 2. Le quotient vaut 1.
  On a $F_3 = 2$.
  $a = 3$ et $b = 2$.
- 3e appel de la boucle : $a = F_4 = 3$ et $b = F_3 = 2$.
  Le reste de la division euclidienne de 3 par 2 vaut 1. Le quotient vaut 1.
  On a $F_2 = 1$.
  $a = 2$ et $b = 1$.
- 4e appel de la boucle : $a = F_3 = 2$ et $b = F_2 = 1$.
  Le reste de la division euclidienne de 2 par 1 vaut 0.
  $a = 1$ et $b = 0$.

On n'a plus d'appel de la boucle et le programme retourne `1`.

b. La suite de Fibonacci est définie par : $F_{n+2} = F_{n+1} + F_n$ et $0 \leq F_n < F_{n+1}$.

On en déduit que le reste de la division euclidienne de F_{n+2} par F_{n+1} est F_n.
D'après la question précédente :

- 1er appel de la boucle : division euclidienne de F_{n+2} par F_{n+1}.
- 2e appel de la boucle : division euclidienne de F_{n+2} par F_n.
- …
- Le dernier appel de la boucle correspond à la division euclidienne de F_3 par F_2. Le reste est nul et l'algorithme retourne `1`.

On obtient alors la suite de valeurs :

$F_0 = 0$
$F_1 = 1$
$F_2 = 1$
$F_3 = 2$
$F_4 = 3$
$F_5 = 5$
$F_6 = 8$

Le nombre de divisions euclidiennes effectuées lorsqu'on calcule le pgcd de F_{n+2} et F_{n+1} est n.

5. a) Première version du programme, en utilisant une liste F pour stocker tous les résultats intermédiaires.

```
def fibo1(n):
    # renvoie le nombre de Fibonacci Fn pour l'entier naturel n
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        F=[]
        F.append(0)
        F.append(1)
        for i in range(2,n+1):
            F.append(F[i-1]+F[i-2])
        return F[i]
```

b) Deuxième version, sans utiliser de liste.

On utilise uniquement deux variables `F_2` et `F_1`.

```
def fibo2(n):
    # renvoie le nombre de Fibonacci Fn pour l'entier naturel n
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        F_2=0   # F[n-2]
        F_1=1   # F[n-1]
        somme=0
        for i in range(2, n+1):
```

```
            somme=F_2+F_1   # F[n]=F[n-1]+F[n-2]
            F_2=F_1
            F_1=somme
        return somme
```

c) Troisième version

On peut remplacer les trois lignes dans la boucle par une seule ligne : `F_2, F_1=F_1, F_2+F_1`.

```
def fibo3(n):
    # renvoie le nombre de Fibonacci Fn pour l'entier naturel n
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        # F[n]=F[n-1]+F[n-2]
        F_2=0   # F[n-2]
        F_1=1   # F[n-1]
        for i in range(2,n+1):
            F_2, F_1=F_1, F_2+F_1
        return F_1
```

6.

```
def fibo_rec(n):
    # renvoie le nombre de Fibonacci Fn pour l'entier naturel n
    if n==0:
        return 0   # condition d'arrêt
    elif n==1:
        return 1   # condition d'arrêt
    else:
        return (fibo_rec(n-1)+fibo_rec(n-2))
                   # appel récursif
```

Remarque : Ceci sera traité dans l'exercice « Suite des nombres de Fibonacci, programmation dynamique » du livre de deuxième année pour une optimisation du programme récursif.

7. Le pgcd est associatif donc **pgcd**(a, b, c) = **pgcd**(**pgcd**(a, b), c).

```
def gcd_trois(a, b, c):
    return euclide(euclide(a, b), c)
```

Exercice 7.6 : Recherche dichotomique dans une liste triée – Version récursive

On considère la liste triée : `L`= [9, 10, 11, 15, 16, 17, 18, 25, 32].

1. Écrire une fonction récursive `rec_dicho_recursive` qui admet comme arguments une liste triée et un élément `x`. Cette fonction affiche « Elément présent » si *x* est dans la liste, « Elément non présent » sinon.

2. Évaluer la complexité de la fonction `rec_dicho_recursive`. On pourra considérer que l'entier *n* est une puissance de 2.

Analyse du problème

On a étudié une version itérative de ce programme (voir exercice 6.2 « Recherche dichotomique dans un tableau trié » dans le chapitre « Algorithmes de dichotomie »).

La méthode dichotomique utilise le fait que la liste est triée. Elle consiste à comparer l'élément recherché à l'élément se trouvant au milieu d'une liste triée. Comme la liste est triée, cela permet d'éliminer la moitié de la liste comme emplacement possible de l'élément, sauf si on l'a déjà trouvé. Ensuite, on prend la moitié de la liste qui reste et on recommence.

1.

```
def rec_dicho_recursive(L, x):
    # la fonction affiche "Elément présent" si x est dans la
    # liste triée L, sinon affiche "Elément non présent"
    if L==[]:   # condition d'arrêt si liste vide
        return ("Elément non présent")
    else:
        # la liste est non vide
        milieu=len(L)//2   # calcul du milieu de la liste
        if x==L[milieu]:   # recherche si l'élément est au milieu
            return ("Elément présent")
        elif x>L[milieu]:
            # recherche entre milieu+1 et la fin de la liste
            return rec_dicho_recursive(L[milieu+1:], x)
            # appel récursif
        else:
            # recherche entre début de liste et milieu-1
            return rec_dicho_recursive(L[:milieu], x)
            # appel récursif

L= [9, 10, 11, 15, 16, 17, 18, 25, 32]
x=32
print(rec_dicho_recursive(L,x))
```

2. On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `rec_dicho_recursive`.

Pour simplifier les calculs, on considère que l'entier n est une puissance de 2.

À chaque appel de la fonction récursive, on a :

- un test pour savoir si la liste est vide ;
- un calcul du milieu ;
- un test pour savoir si l'élément est trouvé ;
- un appel de la fonction récursive avec $n/2$.

On se place dans le pire des cas. On en déduit la relation de récurrence :

$$T(n) = T\left(\frac{n}{2}\right) + 3 \text{ avec } T(1) = 1$$

Il faut calculer le nombre de fois où on fait un appel de la fonction récursive.

- Après un appel de la fonction récursive, on l'appelle à nouveau avec $n/2$.
- Après un deuxième appel de la fonction récursive, on l'appelle à nouveau avec $n/(2^2)$.
- Après k appels de la fonction récursive, on l'appelle à nouveau avec $n/(2^k)$.

Il ne reste plus qu'un seul appel de la fonction récursive quand $\frac{n}{2^k} = 1$, soit $\ln(n) = k \ln(2)$.

Finalement, on obtient :

$$k = \frac{\ln(n)}{\ln(2)} = \log_2(n)$$

On appelle $(k+1)$ fois la fonction récursive. On a donc $T(n = 2^k) = 1 + 3(k+1) = 4 + 3\log_2(n)$.

La complexité est logarithmique en $O(\log_2(n))$.

Remarque :

On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

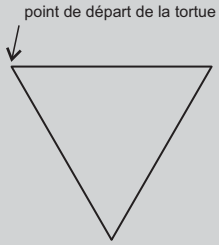
On a la même complexité que pour la version itérative.

Exercice 7.7: Dessins de fractales

On utilise le module `turtle` pour le tracé de fractales. La tortue trace un trait le long du chemin parcouru.

`from turtle import *`	: fenêtre graphique `turtle`. La tortue est le triangle affiché au centre de la fenêtre de coordonnées (0,0)
`reset()`	: efface l'écran
`forward(150)`	: avance la tortue de 150 pixels
`penup()`	: lève le crayon et permet ensuite de déplacer la tortue (avec `forward`) sans tracer
`pendown()`	: pose le crayon et permet ensuite de déplacer la tortue (avec `forward`) avec un trait
`goto(120,200)`	: déplace la tortue au point de coordonnées (120, 200)
`left(60)`	: fait tourner la tortue vers la gauche d'un angle de 60° sans avancer
`right(60)`	: fait tourner la tortue vers la droite d'un angle de 60° sans avancer
`mainloop()`	: laisse la fenêtre graphique `turtle` ouverte à la fin du programme

1. Écrire une fonction `triangle` qui admet comme argument un entier naturel `a` et qui trace un triangle équilatéral de côté `a`.



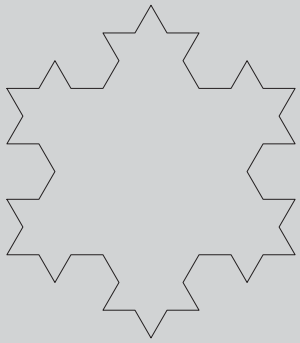
2. Écrire une fonction `polygone` qui admet comme arguments deux entiers naturels `n` et `a`. Cette fonction trace un polygone régulier à `n` côtés de même longueur `a`.

3. On souhaite tracer le segment de Koch de longueur *a* à l'ordre *n*.

- On part d'un segment de longueur *a* à l'ordre 0.
- À l'étape 1, on remplace le tiers du segment central par un triangle équilatéral sans base au-dessus.
- On réitère le processus *n* fois.

Écrire une fonction récursive `koch` qui admet comme arguments deux entiers naturels `n` et `a`. Cette fonction trace un segment de Koch de longueur *a* à l'ordre *n*.

4. Pour tracer le flocon de neige, on part d'un triangle équilatéral et on applique la fonction `koch` à l'ordre *n* à chacun des côtés du triangle équilatéral de longueur *a*. La figure représente le flocon de Von Koch à l'ordre 2.



Écrire le programme principal permettant de tracer un flocon de Von Koch à l'ordre *n*.

Analyse du problème

Le module graphique `turtle` permet de piloter une tortue afin de tracer des figures géométriques.

Une figure fractale est un objet géométrique « infiniment morcelé » dont les détails sont observables à une échelle arbitrairement choisie. Le flocon de Von Koch est un exemple de courbe fractale. En zoomant sur une partie de la figure, on retrouve toute la figure : on dit qu'elle est autosimilaire. À chaque étape, la longueur de la base est multipliée par $\left(\frac{4}{3}\right)^n$. Le périmètre du flocon à l'étape n est : $3a\left(\frac{4}{3}\right)^n$ et tend vers l'infini si n tend vers l'infini. La courbe fractale n'admet de tangente en aucun point.

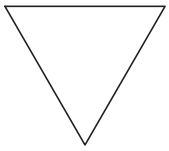
1. Pour tracer le triangle équilatéral de côté *a* :

- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.
- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.
- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.

La tortue revient à sa position initiale avec le même angle.

```
from turtle import *  # bibliothèque pour la fenêtre graphique
                      # turtle
def triangle(a): # la fonction dessine un triangle équilatéral
                 # de côté a
    forward(a)   # avance la tortue de a pixels
    right(120)   # tourne la tortue vers la droite de 120°
    forward(a)   # avance la tortue de a pixels
    right(120)   # tourne la tortue vers la droite de 120°
    forward(a)   # avance la tortue de a pixels
    right(120)   # tourne la tortue vers la droite de 120°
triangle(100)    # appel de la fonction triangle avec un côté
                 # de 100 pixels
mainloop()       # laisse la fenêtre graphique turtle ouverte
                 # à la fin du programme
```

On obtient la figure suivante :

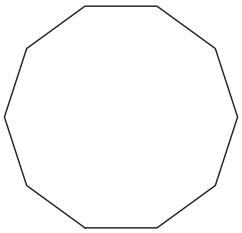


2.

```
def polygone(n, a):    # la fonction dessine un polygone
                       # régulier à n côtés de longueur a
    for i in range(n): # i varie entre 0 inclus et n exclu
```

```
        forward(a)      # avance la tortue de a pixels
        left(360/n)     # tourne la tortue vers la gauche de
                        # 360/n degrés
polygone(10, 50) :      # 10 côtés de longueur 50
mainloop()              # laisse la fenêtre graphique turtle
                        # ouverte à la fin du programme
```

On obtient la figure suivante :



3. On considère un segment de Koch à l'ordre 0 de longueur *a*. On décompose ce segment en trois segments de longueur *a*/3.

- On avance de *a*/3.
- On tourne la tortue vers la gauche de 60°, on avance de *a*/3.
- On tourne la tortue vers la droite de 120°, on avance de *a*/3.
- On tourne la tortue vers la gauche de 60° et on avance de *a*/3.

On obtient le segment de Koch à l'ordre 1 :

Pour tracer le segment de Koch de longueur *a* à l'ordre *n* :

- On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la droite de 120°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.

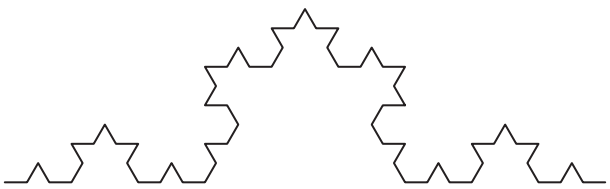
La fonction `koch(n-1, a/3)` trace le segment de Koch en faisant appel 4 fois à la fonction `koch` à l'ordre *n*–2. Cette fonction `koch` à l'ordre *n*–2 fait appel à la fonction `koch` à l'ordre *n*–3…

On peut construire le segment de Koch à l'ordre 1 à partir de quatre segments de Koch à l'ordre 0 :

- On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la droite de 120°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.

Il y a bien une condition d'arrêt à la fonction récursive. Si $n = 0$, on avance la tortue de *a* pixels pour la fonction `koch(0, a)`.

```
def koch(n, a):         # la fonction dessine un segment de Koch
                        # à l'ordre n de longueur a
    if n==0:
        forward(a)      # condition d'arrêt - avance la tortue
                        # de a pixels
    else:               # partage le segment en trois
        # premier tiers : on appelle la fonction récursive
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        left(60)        # tourne la tortue vers la gauche de 60°
        # deuxième tiers : on appelle la fonction récursive
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        right(120)      # tourne la tortue vers la droite de 120°
        # troisieme tiers : on appelle la fonction récursive
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        left(60)        # tourne la tortue vers la gauche de 60°
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
```

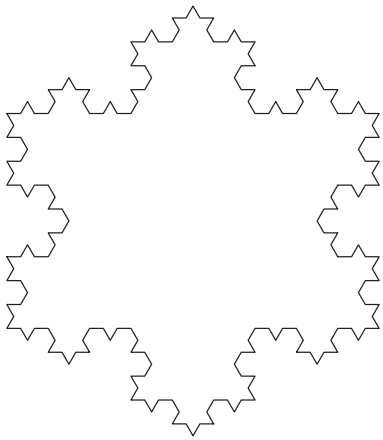


4. Il suffit d'appliquer la fonction `koch` à l'ordre *n* pour chaque côté du triangle équilatéral.

```
n=3          # flocon de Von Koch à l'ordre 3
a=300        # segment de longueur 300
koch(n, a)   # segment de Koch à l'ordre n
right(120)   # tourne la tortue vers la droite de 120°
```

```
koch(n, a)  # segment de Koch à l'ordre n
right(120)  # tourne la tortue vers la droite de 120°
koch(n, a)  # segment de Koch à l'ordre n
right(120)  # tourne la tortue vers la droite de 120°
mainloop()  # laisse la fenêtre graphique turtle ouverte
            # à la fin du programme
```

On obtient une courbe fermée :



Partie 5

Algorithmes gloutons

Plan

8 Algorithmes gloutons

Exercice 8.1 : Suite des nombres de Fibonacci

On note $(F_n)_{n\in\mathbb{N}}$) la suite des nombres de Fibonacci définie par : $F_0 = 0$, $F_1 = 1$, $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

1. Écrire une fonction récursive `fibo1` qui permet de renvoyer le nombre de Fibonacci F_n. L'algorithme utilise-t-il la méthode « diviser pour régner » ?

2. Représenter l'arbre des appels de la fonction récursive `fibo1(5)`. Combien de fois est recalculé F_2 ? Quel est l'inconvénient ?

Analyse du problème

La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes.

Afin d'éviter de calculer plusieurs fois le même nombre de Fibonacci, on utilisera la programmation dynamique (ceci sera traité dans l'ouvrage de deuxième année).

Cours :

La méthode « diviser pour régner » peut se décomposer en trois étapes :

- Diviser : on divise le problème initial en plusieurs sous-problèmes.
- Régner : on traite récursivement chacun des sous-problèmes.
- Combiner : on combine les différents sous-problèmes pour résoudre le problème de départ.

Cette méthode donne de très bons résultats dans de nombreux problèmes : dichotomie, tri par fusion, tri rapide.

La méthode « diviser pour régner » a parfois des faiblesses avec des appels récursifs redondants. Les sous-problèmes ne sont pas toujours indépendants. On peut être amené à résoudre plusieurs fois le même sous-problème.

Une solution consiste à utiliser la programmation dynamique en stockant les résultats déjà calculés : ceci sera traité dans l'ouvrage de deuxième année.

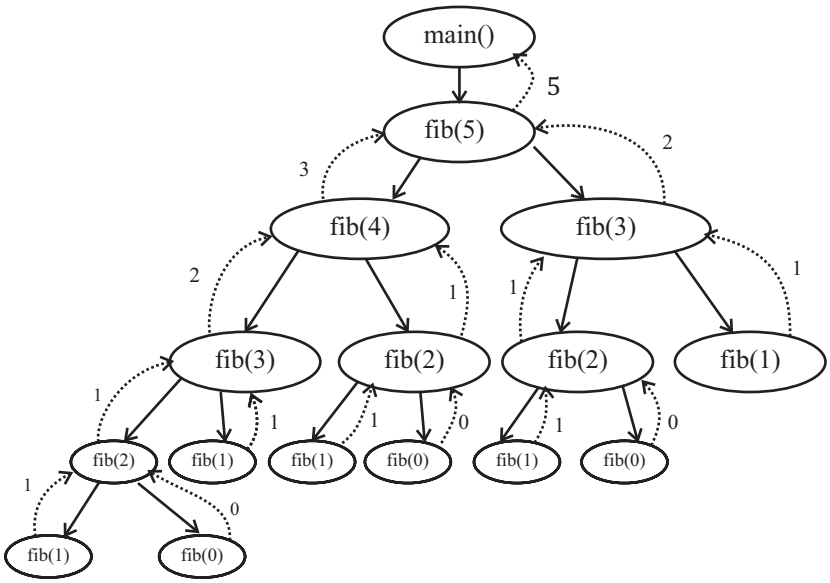
1.

```
def fibo1(n):
    # la fonction renvoie le nombre de Fibonacci Fn
    # pour l'entier n
```

```
if n==0:
    return 0 # condition d'arrêt
elif n==1:
    return 1 # condition d'arrêt
else:
    return (fibo1(n-1)+fibo1(n-2))
             # appel récursif
```

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de `fibo1(n)`) en deux sous-problèmes (calcul de `fibo1(n-1)` et `fibo1(n-2)`). On calcule récursivement chacun des deux sous-problèmes.

2. L'arbre ci-dessous représente les différents appels de la fonction `fibo1` que l'on note `fib`.



F_2 est recalculé 3 fois. F_3 est recalculé 2 fois.
L'inconvénient est que l'on augmente considérablement la complexité puisqu'on recalcule plusieurs fois la même valeur F_n.

Exercice 8.2 : Rendu de monnaie

On cherche à rendre une certaine somme entière X en utilisant le moins de pièces, qui peuvent être identiques. On dispose des pièces entières suivantes : $S = [1, 2, 5, 10, 20, 50, 100] = [S_0, S_1, \ldots, S_{n-1}]$ où $S[i]$ représente la valeur de la pièce d'indice i. On suppose que la liste S est triée par ordre croissant des valeurs.

1. On utilise la méthode la plus intuitive qui consiste à commencer par rendre la plus grande pièce possible. Pour $X = 11$, on commence par rendre la pièce de 10.

On appelle $L[x]$ le nombre de pièces nécessaires pour rendre la somme x. La récurrence (1) peut s'écrire :

- $L[0] = 0$
- Si $x \geq 1 : L[x] = 1 + L\left[x - S\left[i\right]\right]$ avec i le plus grand tel que $S\left[i\right] \leq x$.

Écrire une fonction récursive `rendu1` qui admet comme arguments une liste `S` et un entier `X`. La fonction retourne le nombre de pièces nécessaires pour rendre la somme `X` en utilisant la récurrence (1).

2. L'algorithme utilise-t-il la méthode « diviser pour régner » ? Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `rendu1([1, 4, 6], 8)` retourne la solution optimale ?

Analyse du problème

On étudie plusieurs algorithmes permettant d'optimiser le rendu de monnaie. La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes.

La programmation dynamique (ceci sera traité dans l'ouvrage de deuxième année) permet d'obtenir une solution optimale. On verra la différence entre la méthode gloutonne et la programmation dynamique.

1.

```
def rendu1(S, X):
    # la fonction renvoie le nombre de pièces nécessaires pour
    # rendre la somme X en utilisant la liste S :
    # S[i]=valeur de la pièce d'indice i
    if X==0:  # condition d'arrêt
        return 0
    else:
        # recherche de i le plus grand tel que S[i] <= X
        i=len(S)-1
        while S[i]>X:
            i=i-1
        # ajoute 1 au nombre de pièces
        # puisqu'on utilise la pièce S[i]
        # il reste donc à rendre la monnaie à X - S[i]
        return 1+rendu1(S, X-S[i]) # appel récursif
S=[1, 4, 6]
X=8
print(rendu1(S, X))  # on obtient : 3
```

Cours :
La méthode « diviser pour régner » peut se décomposer en trois étapes :

- Diviser : on divise le problème initial en plusieurs sous-problèmes.
- Régner : on traite récursivement chacun des sous-problèmes.
- Combiner : on combine les différents sous-problèmes pour résoudre le problème de départ.

Dans la méthode gloutonne (greedy, en anglais), on effectue une succession de choix, chacun d'eux semble être le meilleur sur le moment. On résout alors le sous-problème mais on ne revient jamais sur le choix déjà effectué.

2. L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul $L[X]$) en un sous-problème (calcul de $L[X - S[i]]$ avec i le plus grand tel que $S[i] \leq X$). On calcule récursivement le sous-problème.

À chaque étape de l'algorithme, on commence par rendre la plus grande pièce possible, c'est-à-dire la plus grande pièce dont la valeur est inférieure à la somme à rendre. C'est la solution qui semble être la meilleure et la plus intuitive. On déduit alors de cette pièce la somme à rendre et on est ramené à un sous-problème avec une somme à rendre plus petite. On recommence jusqu'à obtenir une somme nulle.

Cet algorithme est très simple mais à chaque étape on n'étudie pas tous les cas possibles puisqu'on se contente de choisir la pièce la plus grande que l'on peut rendre.

Dans le cas où $S = [1, 4, 6]$ et $X = 8$, on n'obtient pas la solution optimale. L'algorithme glouton (greedy algorithm, en anglais) renvoie 3 (1 pièce de 6 et 2 pièces de 1) alors que la solution optimale est 2 (2 pièces de 4).

Exercice 8.3 : Problème du sac à dos

On considère un sac à dos dont la masse maximale est notée M. On cherche à maximiser la valeur totale des objets insérés dans le sac à dos. On dispose de n objets modélisés par la liste de listes `S` :

- `S[i][0]` désigne la valeur de l'objet d'indice i notée v_i (i varie de 0 à n–1).
- `S[i][1]` désigne la masse de l'objet d'indice i notée m_i (i varie de 0 à n–1).

On suppose dans tout le problème que $\sum_{i=0}^{n-1} m_i > M$ et que les masses sont des entiers. Le premier objet de la liste `S` a pour indice 0. La liste `S` est triée par ordre décroissant du rapport valeur/masse.

1. On utilise la méthode intuitive consistant à insérer au fur et à mesure les objets qui ont le plus grand rapport valeur/masse.

Écrire une fonction itérative `algo1` qui admet comme arguments une liste `S` et un entier `M`. La fonction retourne la valeur des objets que l'on peut insérer dans le sac à dos.

2. Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `algo1 ([[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]], 30)` retourne la solution optimale ?

Analyse du problème

On étudie plusieurs méthodes permettant de maximiser la valeur des objets insérés dans un sac à dos. La programmation dynamique (ceci sera traité dans l'ouvrage de deuxième année) permet d'obtenir une solution optimale en utilisant deux techniques : « top down » et « bottom up ».

1.

```
def algo1(S, M):
    # S est une liste de listes avec [valeur, masse].
    # Les objets sont triés par ordre décroissant valeur/masse.
    # La fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    v_total=0    # initialisation de la valeur totale des objets
    m_total=0    # initialisation de la masse totale des objets
    n=len(S)
    for i in range(n):
        if m_total+S[i][1]<=M: # teste si nouvelle masse totale<=M
            v_total+=S[i][0]   # calcule la nouvelle valeur totale
            m_total+=S[i][1]   # calcule la nouvelle masse totale
    return v_total  # retourne la valeur totale des objets
```

Avant d'insérer un objet, il faut tester que la nouvelle masse totale ne dépasse pas *M*.

2. Cette méthode est appelée méthode gloutonne car elle consiste à faire le meilleur choix sur le moment, c'est-à-dire insérer l'objet qui a le plus grand rapport valeur/masse.

```
M=30
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]] # [valeur, masse]
print('ALGO1 :', algo1(S, M))                     # affiche 32
```

On peut calculer le rapport valeur/masse pour chaque objet. On obtient alors la liste suivante, qui est bien triée par ordre décroissant : [2.5, 2.4, 2.0, 0.875, 0.5].

- On insère le premier objet de valeur 15 et de masse 6.
- On ne peut pas insérer le deuxième objet de masse 25 car la masse totale 6+25 dépasse 30.
- On insère le troisième objet de valeur 10.
- On insère le quatrième objet de valeur 7.

On obtient une valeur totale 32 dans le sac à dos.

Cette méthode ne donne pas toujours la valeur optimale. Dans ce cas particulier, `algo1` renvoie 32 alors que la solution optimale est 70. On utilisera la programmation dynamique (ceci sera traité dans l'ouvrage de deuxième année) pour trouver la solution optimale.

Exercice 8.4 : Allocation de salle de spectacles

On cherche une solution au problème d'allocation d'une salle de spectacles. On définit une liste L contenant pour chaque spectacle d'indice $i \in [\![0, n-1]\!]$ le couple d'entiers (d_i, f_i) où d_i désigne l'heure de début et f_i l'heure de fin : $L = [[d_0, f_0], [d_1, f_1], ..., [d_{n-1}, f_{n-1}]]$. On suppose que les n-uplets de couples (d_i, f_i) sont triés par date de fin f_i croissante. On définit `début` l'heure de début du spectacle et `fin` l'heure de fin du spectacle.

1. On cherche à maximiser le nombre de spectacles dans la salle et non le temps d'occupation. On utilise la méthode intuitive consistant à choisir au fur et à mesure des spectacles dont l'intervalle est compatible avec celui du spectacle précédent et dont l'heure de fin est la plus petite. On suppose que la liste `L` est triée par ordre croissant des heures de fin.

Écrire une fonction itérative `gestion` qui admet comme arguments une liste `L`, un entier `début` (heure de début des spectacles) et un entier `fin` (heure de fin des spectacles). La fonction retourne le nombre maximum de spectacles que l'on peut organiser dans la salle ainsi que la liste des spectacles retenus.

2. Pourquoi cette méthode est-elle appelée gloutonne ?

3. On considère la liste `L1=[[0, 2], [1, 3], [2, 4], [1, 5], [3, 6], [4, 7], [5, 9], [6, 11], [9, 12]]`. Écrire le programme principal permettant d'afficher le nombre maximum de spectacles et la liste des spectacles que l'on peut organiser dans cette salle entre `début` = 1 et `fin` = 12.

Analyse du problème

On utilise la méthode gloutonne, qui consiste à effectuer une succession de choix, chacun d'eux semblant être le meilleur sur le moment.

1.

```
def gestion(L, début, fin):
    # retourne le nombre maximum de spectacles que l'on peut
    # organiser entre début(int) = heure de début et
    # fin(int) = heure de fin dans la salle ainsi que la liste
    # des spectacles retenus L = liste de listes
    # L[i]=[di, fi] = heure de début et de fin du spectacle i
    n=len(L)        # nombre de spectacles
    p=0             # initialisation du nb de spectacles organisés
```

```
    LISTE_CONF=[]  # initialisation de la liste des
                   # spectacles organisés
    for i in range(n): # i varie entre 0 inclus et n exclu
        if LISTE_CONF==[] and L[i][0]>=début and L[i][1]<=fin:
            p=p+1
            LISTE_CONF.append(i)    # indice du premier spectacle
                                    # ajouté
        elif p>=1:
            indice=LISTE_CONF[p-1] # indice du dernier spectacle
                                    # ajouté
            if L[i][0]>=L[indice][1] and L[i][1]<=fin:
                p=p+1
                LISTE_CONF.append(i) # indice du spectacle ajouté
    return(p, LISTE_CONF)
```

2. On appelle la méthode gloutonne puisque, à chaque étape, on sélectionne la possibilité qui semble être la meilleure sur le moment, c'est-à-dire qu'on choisit un spectacle qui finit le plus tôt.

On peut montrer que la méthode gloutonne donne ici la solution optimale, mais ce n'est pas toujours le cas, comme on peut le voir dans les exercices sur le rendu de monnaie, le sac à dos et la barre de métal (exercices 8.2, 8.3 et 8.5)

3.

```
L1=[[0, 2], [1, 3], [2, 4], [1, 5], [3, 6], [4, 7], [5, 9], \
  [6, 11], [9, 12]]
début=1     # heure de début des spectacles
fin=11      # heure de fin des spectacles
p1, LISTE_CONF1=gestion(L1, début, fin)
print('Nombre de spectacles :', p1)
print("Spectacles que l'on peut organiser :", LISTE_CONF1)
```

Le programme Python affiche :

```
Nombre de spectacles : 3
Spectacles que l'on peut organiser : [1, 4, 7]
```

Exercice 8.5 : Découpe d'une barre de métal

On considère une barre de métal de longueur n où $n \in \mathbb{N}^*$. On cherche à calculer le prix optimal que l'on peut obtenir de cette barre en la découpant à des abscisses entières. Pour chacune des abscisses de la barre, on a deux possibilités : on coupe la barre ou on ne la coupe pas. On considère la liste de listes `S` :

- `S[i][0]` désigne le prix de vente de la barre d'indice i (i varie entre 0 et n–1).
- `S[i][1]` désigne la longueur de la barre d'indice i (i varie entre 0 et n–1).

Pour une barre de longueur 5, on a par exemple : `S=[[14, 3], [22, 5], [16, 4], [3, 1], [5, 2]]`.

1. On utilise la méthode intuitive consistant à découper au fur et à mesure la barre avec le plus grand rapport prix/longueur. On suppose que la liste `S` est triée par ordre décroissant du rapport prix/longueur.

Écrire une fonction itérative `decoupe1` qui admet comme argument une liste `S` et retournant le prix optimal que l'on peut obtenir de cette barre.

2. Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `decoupe1([[14, 3], [22, 5], [16, 4], [3, 1], [5, 2]])` retourne la solution optimale ?

Analyse du problème

On utilise la méthode gloutonne, qui consiste à effectuer une succession de choix, chacun d'eux semblant être le meilleur sur le moment. On résout alors le sous-problème mais on ne revient jamais sur le choix déjà effectué. La programmation dynamique (ceci sera traité dans l'ouvrage de deuxième année) permet d'obtenir une solution optimale.

1. On peut calculer le rapport prix/longueur pour chaque barre de `S=[[14, 3], [22, 5], [16, 4], [3, 1], [5, 2]]`. On obtient alors la liste suivante, qui est bien triée par ordre décroissant prix/longueur : 4.67, 4.4, 4.0, 3.0, 2.5.

```
def decoupe1(S): # S est une liste de listes avec [prix, longueur]
                 # les barres sont triées par ordre décroissant
                 # prix/longueur
    # la fonction retourne le prix optimal que l'on peut obtenir
    # de cette barre
    prixtot=0    # initialisation du prix total
    Ltot=0       # initialisation de la longueur totale
    n=len(S)     # longueur de la barre
    i=0
    while Ltot<n:
        if Ltot+S[i][1]<=n:   # teste si nouvelle long. totale<=n
            prixtot+=S[i][0]  # calcule le nouveau prix total
            Ltot+=S[i][1]     # calcule la nouvelle longueur
                              # totale
            # il ne faut pas incrémenter i de 1 car on peut
            # utiliser i à nouveau
        else: # on ne peut plus utiliser la barre d'indice i
            i+=1
    return prixtot  # prix total de la barre de longueur n
```

On ajoute au fur et à mesure des barres de longueurs différentes pour obtenir la barre de longueur n. Au moment du choix de la barre d'indice i à ajouter (longueur $S[i][1]$ et prix $S[i][0]$), il faut tester si la nouvelle longueur `Ltot+S[i][1]` ne dépasse pas la longueur n de la barre.

Cette méthode est appelée méthode gloutonne car elle consiste à faire le meilleur choix sur le moment, c'est-à-dire choisir une barre qui a le plus grand rapport prix/longueur.

```
S=[[14, 3], [22, 5], [16, 4], [3, 1], [5, 2]]
    # barre de longueur 5
    # S[i][0] = prix pour la barre d'indice i
    # S[i][1] = longueur pour la barre d'indice i
    # liste S triée ordre décroissant prix/longueur
prix=decoupe1(S)
print("Prix découpe1 :", prix)
```

Lorsqu'on appelle la fonction `decoupe1` avec `S=[[14, 3], [22, 5], [16, 4], [3, 1], [5, 2]]` :

- La première barre choisie est la barre de longueur 3 avec un prix égal à 14. Le prix total vaut 14.
- La deuxième barre choisie est la barre de longueur 1 avec un prix égal à 3. Le prix total vaut 17.
- La troisième barre choisie est la barre de longueur 1 avec un prix égal à 3. Le prix total vaut 20.

La méthode gloutonne ne donne pas toujours la valeur optimale.

`decoupe1(S)` renvoie 20 alors que la solution optimale est 22.

Partie 6

Lecture et écriture de fichiers – Matrices de pixels et images

Plan

Lecture et écriture de fichiers

9

Exercice 9.1 : Lecture et écriture de fichiers – Régression linéaire

On fournit les fonctions suivantes pour la gestion des fichiers :

`f=open('fichier.txt','w')` : `'fichier.txt'` désigne le nom du fichier. Le mode d'ouverture peut être `'w'` pour « écriture » (*write*), `'r'` pour « lecture » (*read*) ou `'a'` pour « ajout » (*append*)

`f.readline()` : lecture d'une ligne du fichier `f`

`\n` : caractère d'échappement : saut de ligne

`c1.strip()` : renvoie une chaîne sans les espaces et les caractères d'échappement (saut de ligne par exemple) en début et fin de la chaîne de caractères `c1`

`c1.split(';')` : sépare une chaîne de caractères (`c1`) en une liste de mots avec le séparateur `';'`

`f.write('exemple')` : écrit dans le fichier `f` la chaîne de carcatères `'exemple'`

`f.close()` : ferme le fichier `f`

On utilisera les tableaux de type numpy.array et la bibliothèque pyplot pour les graphes :

```
import numpy as np
import matplotlib.pyplot as plt
```

1. Écrire un programme Python permettant de créer le fichier `'droite.txt'` contenant les éléments suivants :

`10` : nombre de points du fichier

`25;10.9` : abscisse et ordonnée du premier point

`20;9.3`

`15;8.2`

`12;7.5`

`9;6.2`

`6;5.8`

`3;4.2`

`0;3.9`

`-3;2.8`

`-6;2` : abscisse et ordonnée du dixième point

2. Écrire un programme Python permettant d'ouvrir un fichier .txt (par exemple `'droite.txt'`) et de récupérer les abscisses et les ordonnées dans deux tableaux.

3. On cherche à modéliser les n points expérimentaux (x_1, y_1), $(x_2, y_2)\ldots$ (x_n, y_n) par une fonction polynôme du premier ordre, de la forme : $y = ax + b$.

$a = \dfrac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$ et $b = \overline{y} - a\overline{x}$ en notant $\overline{x}$ la moyenne des x_i et $\overline{y}$ la moyenne des y_i.

Écrire un programme Python permettant de calculer les coefficients a et b correspondant aux points expérimentaux de la question 2.

4. Écrire un programme Python permettant de représenter graphiquement la fonction $y_{\text{modélisé}} = ax + b$.

Analyse du problème

Cet exercice utilise les fonctions d'écriture et de lecture de fichiers avec Python. Il faut convertir les entiers en chaînes de caractères avant d'utiliser `f.write`. Lors de la lecture du fichier, on parcourt les différentes lignes du fichier avec `f.readline()`. On enlève ensuite les caractères d'échappement (saut de ligne par exemple) et on sépare la chaîne de caractères obtenue en une liste de mots.

1.

```
import numpy as np   # bibliothèque numpy renommée np
x=np.array([25,20,15,12,9,6,3,0,-3,-6])   # tableau des abscisses
y=np.array([10.9,9.3,8.2,7.5,6.2,5.8,4.2,3.9,2.8,2])
                                          # tableau des ordonnées
n=len(x)   # longueur du tableau x. Le type de n est int

f=open('droite.txt', 'w')   # création du fichier 'droite.txt'
                            # en écriture
```

```
f.write(str(n)+'\n')  # première ligne avec la longueur du tableau
for i in range(n):  # i varie entre 0 inclus et n exclu
    f.write(str(x[i])+';'+str(y[i])+'\n')
f.close()   # fermeture du fichier f
```

On convertit les entiers et les flottants en chaînes de caractères avant de les insérer dans le fichier. Il ne faut pas oublier le saut de ligne à la fin de chaque ligne du fichier.

2.

```
f=open('droite.txt', 'r')
n=int(f.readline())   # nombre de points
x=np.zeros(n)   # création du tableau numpy x avec toutes
                # les valeurs égales à 0
y=np.zeros(n)   # création du tableau numpy y avec toutes
                # les valeurs égales à 0
for i in range(n):   # i varie entre 0 inclus et n exclu
    ligne=f.readline()
    ligne2=ligne.strip()   # enlève les caractères d'échappement
    xchaine,ychaine=ligne2.split(';')
    x[i]=float(xchaine)   # conversion de l'abscisse en flottant
            # xchaine est une chaîne de caractères qu'il faut
            # convertir en type float
    y[i]=float(ychaine)   # conversion de l'ordonnée en flottant
f.close()   # fermeture du fichier f
```

3.

```
# procédure pour calculer la somme des xi
sumx=0
for i in range(n):
    sumx=sumx+x[i]

# somme des yi
sumy=np.sum(y)   # autre façon plus rapide de calculer
                 # la somme des yi

# somme des xi*yi
sumxy=0
for i in range(n):
    sumxy=sumxy+x[i]*y[i]
# on pourrait écrire directement : sumxy=np.sum(x*y)

# somme des xi*xi
B=x**2
sumx2=sum(B)

moyennex=np.mean(x)   # moyenne des xi
moyenney=np.mean(y)   # moyenne des yi

a=(n*sumxy-sumx*sumy)/(n*sumx2-sumx**2)
b=moyenney-a*moyennex
```

Les fonctions `np.sum(A)` et `np.mean(A)` permettent de calculer directement la somme et la moyenne des éléments d'un tableau `A`.

Le programme Python fournit $a = 0{,}29$ et $b = 3{,}75$.

4.

```
ymodelise=a*x+b
plt.plot(x,ymodelise)
```

Le tableau `x` contient les 10 points expérimentaux.

L'instruction `ymodelise=a*x+b` permet de calculer le nouveau tableau `ymodelise` à partir du tableau `x`.

Remarque :

On pourrait écrire le programme suivant pour définir le tableau `ymodelise` :

```
ymodelise=np.zeros(n)  # le tableau ymodelise contient n valeurs nulles
for i in range(n):     # i varie entre 0 inclus et n exclu
    ymodelise[i]=a*x[i]+b
```

La première variable dans `plt.plot` est le tableau des abscisses des points. La deuxième variable est le tableau des ordonnées des points.

Exercice 9.2 : Lecture de fichiers – Méthode des trapèzes

On fournit les fonctions suivantes pour la gestion des fichiers :

`f=open('fichier.txt','w')` : `'fichier.txt'` désigne le nom du fichier. Le mode d'ouverture peut être `'w'` pour « écriture » (*write*), `'r'` pour « lecture » (*read*) ou `'a'` pour « ajout » (*append*)

`f.readlines()` : transforme toutes les lignes du fichier `f` en une liste de chaînes de caractères

`c1.strip()` : renvoie une chaîne sans les espaces et les caractères d'échappement (saut de ligne par exemple) en début et fin de la chaîne de caractères `c1`

`c1.split(';')` : sépare une chaîne de caractères (`c1`) en une liste de mots avec le séparateur `';'`

`f.write('exemple')` : écrit dans le fichier `f` la chaîne `'exemple'`

`f.close()` : ferme le fichier `f`

On utilisera la bibliothèque pyplot pour les graphes :

```
import matplotlib.pyplot as plt
```

On cherche à calculer une valeur approchée de l'intégrale d'une fonction donnée par des points dont les coordonnées sont situées dans un fichier.

1. Le fichier `"ex_001.txt"` contient une quinzaine de lignes selon le modèle suivant :

```
0.0;0.0
0.111111111111;0.0122949573053
0.222222222222;0.0485751653206
```

Chaque ligne contient deux valeurs flottantes séparées par un point-virgule, représentant respectivement l'abscisse et l'ordonnée d'un point. Les points sont ordonnés dans l'ordre croissant de leurs abscisses.

Écrire un programme Python permettant d'ouvrir le fichier en lecture, de le lire et de construire la liste `X` des abscisses et la liste `Y` des ordonnées contenues dans ce fichier.

2. Écrire un programme Python permettant de représenter les points sur une figure.

3. Les points précédents sont situés sur la courbe représentative d'une fonction *f*. On souhaite déterminer une valeur approchée de l'intégrale *I* de cette fonction sur le segment où elle est définie.

Écrire une fonction `trapeze`, d'arguments deux listes *y* et *x* de même longueur *n*, renvoyant :

$$\sum_{i=1}^{n-1}\left(x_i - x_{i-1}\right)\frac{y_i + y_{i-1}}{2}$$

`trapeze(Y, X)` renvoie donc une valeur approchée de l'intégrale *I* par la méthode des trapèzes.

Analyse du problème

Cet exercice utilise les fonctions de lecture de fichiers avec Python. Dans l'exercice précédent, on lit les lignes du fichier au fur et à mesure en utilisant l'instruction `f.readline()`. Ici on récupère directement une liste de chaînes de caractères avec l'instruction `f.readlines()`. Il faut alors parcourir cette liste pour récupérer les différentes lignes du fichier. Pour chaque ligne, on enlève les caractères d'échappement (saut de ligne par exemple) et on sépare la chaîne de caractères obtenue en une liste de mots.

1. On récupère dans la variable `data` une liste de chaînes de caractères. Pour parcourir cette liste, on utilise l'instruction `for chaine in data`.

```
f=open("ex_001.txt", 'r')   # ouverture du fichier en lecture
data=f.readlines()   # data contient une liste de chaînes
                     # de caractères
    # chaque chaîne de caractères correspond à une ligne
    # du fichier
X, Y=[], []   # création des listes vides X et Y
```

```
for chaine in data:   # on parcourt la liste de chaînes de
                      # caractères
    chaine2=chaine.strip()   # enlève les caractères d'échappement
    abs,ord=chaine2.split(';')
    X.append(float(abs))   # conversion de l'abscisse en flottant
            # on ajoute cette abscisse dans la liste X
    Y.append(float(ord))   # conversion de l'ordonnée en flottant
            # on ajoute cette ordonnée dans la liste Y
f.close()   # fermeture du fichier f
```

Cours :
Pour initialiser une liste, on utilise `X=[]`.

Pour ajouter des éléments dans une liste, on utilise `X.append(valeur)`.

Pour supprimer le dernier élément d'une liste, on utilise `X.pop()`.

Il faut bien connaître la syntaxe de `plot` : la première liste représente l'abscisse des points, la deuxième liste représente l'ordonnée des points.

2.

```
plt.figure()   # nouvelle fenêtre graphique
plt.plot(X, Y)
    # ou utilisation de la fonction ci-dessous, qui permet
    # de mettre  une croix pour les points expérimentaux
    # et de relier les points entre eux :
    # plt.plot(LX, LY, '+', linestyle="-")
plt.show()   # affiche la figure à l'écran
```

Cours :
Il faut bien connaître la syntaxe de `range` :

```
for i in range(start, stop, step):
```

`i` varie entre `start` inclus et `stop` exclu avec un pas égal à `step`.

Lorsque `step` n'est pas indiqué, le pas vaut 1 par défaut.

3.

```
def trapeze(Y, X):
    # la fonction renvoie l'intégrale I par la méthode des
    # trapèzes
    n=len(X)   # nombre d'éléments de la liste
    I=0   # initialisation de la variable I
    for i in range(1,n):   # i varie entre 1 inclus et n exclu
        I+=(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
    return I

print(trapeze(Y, X))
```

On pourrait écrire :

```
I=I+(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
```

au lieu de :

```
I+=(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
```

Traitement d'images

10

Exercice 10.1 : Traitement d'images et filtrage passe-bas

On fournit les instructions Python :

```
import numpy as np   # bibliothèque numpy renommée np
A.shape   # l'instruction renvoie (i, j) : dimension d'un tableau A,
          # tableau numpy de taille (i, j)
```

Le programme Python ci-dessous permet de récupérer dans le tableau `img` une image en niveaux de gris. Les valeurs des pixels varient entre 0 (noir) et 255 (blanc). On utilisera les tableaux numpy.

```
import matplotlib.pyplot as plt
from scipy import ndimage
img = plt.imread('photoNB.jpg')   # récupération d'une image jpeg
plt.figure()
plt.imshow(img,cmap=plt.cm.gray)   # visualisation de l'image jpeg
plt.show()
```

1. Écrire un programme Python permettant d'obtenir le nombre de lignes et le nombre de colonnes.

2. Écrire un programme permettant d'afficher la moitié inférieure de l'image `img` avec une inversion du contraste.

3. Écrire un programme permettant d'utiliser trois niveaux de gris uniquement pour représenter l'image `img` : les niveaux de gris entre 0 et 80 inclus sont remplacés par 60, les niveaux de gris entre 80 et 150 inclus sont remplacés par 120 et les autres niveaux de gris sont remplacés par 220.

4. Écrire un programme permettant de créer une nouvelle image `img2` en prenant un pixel sur trois de l'image `img` pour la largeur et la hauteur.

5. On souhaite réaliser un filtre passe-bas qui adoucit les détails de l'image `img2`. On considère le tableau suivant, noté *A* :

1/12	1/12	1/12
1/12	4/12	1/12
1/12	1/12	1/12

Le traitement suivant est appliqué à l'image `img2`. Pour calculer la nouvelle valeur du pixel de l'image traitée :

- on multiplie son ancienne valeur img2[i,j] par la valeur centrale du tableau `A` ;
- on additionne la valeur des pixels adjacents à l'image multipliée par la valeur des pixels adjacents au tableau `A` : A[0,0]×img2[i-1,j-1]+…

Écrire un programme permettant d'afficher le résultat du filtrage passe-bas de l'image `img2`.

Analyse du problème

On repère un pixel par (*i*, *j*) où *i* désigne l'indice de la ligne et *j* l'indice de la colonne. Pour chaque pixel, on a un niveau de gris compris entre 0 et 255. On obtient alors un tableau dont chaque valeur correspond au niveau de gris. On peut alors modifier facilement les valeurs du tableau pour effectuer un traitement d'images.

Cours :

Le slicing, ou accès par tranches, permet d'extraire des données d'un tableau. On considère le tableau suivant :

```
A = np.array([1,2,3,4,5])
```

Cette instruction permet d'extraire des éléments du tableau `A` :

```
A[start:stop:step]
```

Elle renvoie les éléments `A[i]` tels que `start` ≤ `i` < `stop`. Le terme d'indice `stop` n'est jamais inclus dans le slice obtenu.

`step` désigne le pas de variation de `i`.

On peut omettre n'importe lequel des argument dans `A[start:stop:step]`.

`A[1:3]` retourne `2, 3`

On a `start` = 1, `stop` = 3. Lorsque le pas `step` n'est pas indiqué, il vaut 1 par défaut.

Dans ce cas, `stop`–`start` désigne la longueur du slice obtenu. Ici, on récupère 3–1 = 2 éléments.

L'instruction `A[1:5:2]` retourne `2, 4`.

Voir l'exercice 2.4 « Slicing, tranche, `range`, `np.arrange`, `np.linspace` » dans le chapitre « Tableaux numpy – Slicing ».

1. `img.shape` retourne `(i, j)` avec `i` le nombre de lignes et `j` le nombre de colonnes. On obtient un tuple `(i, j)`.

```
img_height = img.shape[0]   # nombre de lignes = 600
img_width = img.shape[1]   # nombre de colonnes = 800
print('hauteur = nombre de lignes = ',img_height)
print('largeur = nombre de colonnes = ',img_width)
```

Remarque : On pourrait écrire également :

```
img_height, img_width =img.shape
```

800×600 représente le nombre de pixels. 800 désigne le nombre de colonnes et 600 désigne le nombre de lignes.

```
img[i, j]   # i désigne la ligne et j désigne la colonne
```

2.

```
import copy
img2=copy.deepcopy(img)
img_trait1=img2[img_height//2:, :]

lignes, colonnes=img_trait1.shape   # nombre de lignes
                                    # et de colonnes
for i in range(lignes):
    for j in range(colonnes):
        img_trait1[i, j]=255-img_trait1[i, j]   # inversion du
                                                # contraste
plt.figure()
plt.imshow(img_trait1, cmap=plt.cm.gray)
plt.show()
```

Pour obtenir la moitié inférieure de l'image, il faut extraire les lignes et les colonnes du tableau.

On aurait pu écrire :

```
img_trait1=img2[img_height//2:img_height, 0:img_width]
```

On peut éviter d'écrire deux boucles en utilisant l'instruction suivante :

```
img_trait1=255-img_trait1
```

Toutes les valeurs du tableau sont remplacées par 255–valeur. Le temps de calcul est beaucoup moins long qu'avec des boucles `for`.

Remarque :

Si on écrit uniquement : `img_trait1=img[img_height//2:, :]` sans écrire la ligne `imp2=copy.deepcopy(img)`, la variable `img_trait1` ne contient pas l'image mais fait juste référence à l'image `img` qui existe dans la mémoire de la machine.

Si on affiche l'image de départ après avoir fait le traitement, on constate que l'image de départ est modifiée (voir exercice 2.2 « Affectation, objet immuable, copie de listes et de tableaux » dans le chapitre « Tableaux numpy – Slicing »).

`img` et `img_trait1` font donc référence au même tableau.

Pour effectuer une copie profonde d'un tableau dans un tableau, on peut utiliser :

```
img2=copy.deepcopy(img)
img_trait1=img2[img.shape[0]//2:, :]
```

3.

```
for i in range(img_height):
    for j in range(img_width):
        if img[i, j]<=80:   # teste si le niveau de gris est <= 80
            img[i, j]=60
        elif img[i, j] <=150:   # teste si niveau de gris <= 150
            img[i, j]=120
        else:   # le niveau de gris est > 150
            img[i, j]=220
```

On pourrait écrire `elif img[i, j]>80 and img[i,j]<=150`. C'est inutile car, si la première condition est vérifiée, Python ne teste pas l'instruction après `elif`.

4.

```
img2=img[::3, ::3]   # extraction d'un pixel sur 3 pour les lignes
                     # et les colonnes
```

Pour les lignes et les colonnes, on aurait pu écrire :

```
img_trait1=img2[0:img_height:3, 0:img_width:3]
```

Il est plus compact d'écrire `[::3]` plutôt que `[0:img_height:3]`.

5.

```
for i in range(1, img2.shape[0]-1):
    for j in range(1, img2.shape[1]-1):
        B=img2[i-1:i+2, j-1:j+2]   # tableau de 9 valeurs
        C=A*B   # fait le produit case par case
        img2[i, j]=sum(sum(C))   # on fait la somme des 9 cases
                                 # de C

plt.figure()
plt.imshow(img2, cmap=plt.cm.gray)
plt.show()
```

Les tableaux `A` et `B` ont la même dimension (3,3). L'instruction `C=A*B` permet de multiplier chaque valeur de la case d'un tableau par la valeur de la case de l'autre tableau.

Il reste à faire la somme de toutes les valeurs du tableau `C` de dimension (3,3). On pourrait écrire deux boucles pour faire la somme des 9 éléments du tableau `C`.

Exercice 10.2 : Filtrage d'images

Une image 800 × 600 comprend $n = 800$ colonnes et $m = 600$ lignes. Le point supérieur gauche de l'image a pour coordonnées [0, 0], le point inférieur droit [599, 799]. Une image en niveaux de gris est représentée sous forme de tableau à deux dimensions de taille $m \times n$ contenant des valeurs de type flottant, comprises entre 0 et 255. On utilisera les tableaux numpy.

1. On souhaite réaliser un lissage de l'image `img` en niveaux de gris. On considère `A` le tableau :

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Le traitement suivant est appliqué à l'image `img`. Pour calculer la nouvelle valeur du pixel de l'image traitée :

- On multiplie $\mathrm{img}[\mathrm{i},\mathrm{j}]$ par la valeur centrale du tableau `A`.
- On additionne la valeur des pixels adjacents à l'image multipliée par la valeur des pixels adjacents au tableau `A` : $\mathrm{A}[0,0]\times\mathrm{img}[\mathrm{i}-1,\mathrm{j}-1]+...$
- La nouvelle valeur du pixel est égale à la valeur absolue de la somme précédente.

Proposer un programme permettant de réaliser le filtrage de l'image `img`.

2. La dérivée dans la direction horizontale peut être approchée par $img[i,j]-img[i-1,j]$. Proposer un filtre permettant de détecter le changement d'intensité d'une couleur selon la direction horizontale. On appelle `img3` l'image ainsi filtrée.

3. Proposer un programme permettant de détecter le contour selon la direction verticale. On appelle `img4` l'image ainsi filtrée.

4. Proposer un programme permettant de détecter les contours dans les deux directions en utilisant pour chaque point de l'image $img5[i,j]=\sqrt{\left(img3[i,j]\right)^2+\left(img4[i,j]\right)^2}$.

Analyse du problème

On repère un pixel par (*i, j*) où *i* désigne l'indice de la ligne et *j* l'indice de la colonne. Pour chaque pixel, on a un niveau de gris compris entre 0 et 255. On utilise deux boucles `for` pour décrire tous les pixels de l'image.

1. `np.zeros((m, n))` permet de créer un tableau de type flottant avec *m* lignes et *n* colonnes.

```
A=np.array([[1,1,1], [1,1,1], [1,1,1]])
A=A/9       # toutes les valeurs du tableau A sont divisées par 9
m=img.shape[0]  # m = nombre de lignes de l'image img
n=img.shape[1]  # n = nombre de colonnes de l'image img
img2=np.zeros((m, n))
for i in range(1, m-1):
    for j in range(1, n-1):
```

```
        B=img[i-1:i+2, j-1:j+2]    # tableau de 9 valeurs
        C=A*B                      # fait le produit case par case
        img2[i, j]=np.abs(sum(sum(C)))  # fait la somme des
                                        # 9 cases de C
```

Remarque : On peut écrire deux boucles `for` imbriquées pour calculer `C` et la somme des 9 cases de `C`.

2. La dérivée dans la direction horizontale peut être approchée par $img[i,j]-img[i-1,j]$. Le tableau `A` suivant permet de détecter le contour selon la direction horizontale :

0	0	0
−1	1	0
0	0	0

```
A=np.array([[0,0,0], [-1,1,0], [0,0,0]])
img3=np.zeros(img.shape)
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        B=img[i-1:i+2, j-1:j+2]    # tableau de 9 valeurs
        C=A*B                      # fait le produit case par case
        img3[i, j]=np.abs(sum(sum(C)))  # fait la somme des
                                        # 9 cases de C
```

3. La dérivée dans la direction verticale peut être approchée par $img[i,j]-img[i,j-1]$. Le tableau `A` suivant permet de détecter le contour selon la direction verticale :

0	−1	0
0	1	0
0	0	0

```
A=np.array([[0,-1,0], [0,1,0], [0,0,0]])
img4=np.zeros(img.shape)
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        B=img[i-1:i+2, j-1:j+2]    # tableau de 9 valeurs
        C=A*B                      # fait le produit case par case
        img4[i, j]=np.abs(sum(sum(C)))  # fait la somme des
                                        # 9 cases de C
```

4.

```
img5=np.sqrt(img3**2+img4**2)
```

On peut tester les programmes suivants avec les instructions suivantes :

```
img= plt.imread('photo.png')  # récupération d'une image
```

Parfois, la valeur d'un pixel est un flottant compris entre 0 et 1, obtenu en divisant la valeur entière (8 bits) par 255.

```
img=np.int_(img*255)
plt.figure()
plt.imshow(img, cmap='gray')  # visualisation en niveaux de gris
plt.show()
```

Partie 7

Tris

Plan

Tris

11

Exercice 11.1 : Tri par insertion

Le tri par insertion est souvent utilisé pour trier des cartes. Il consiste à insérer les éléments d'une partie de la liste non triée dans la liste triée.

Présentation du tri par insertion :

On considère la liste non triée : 8 5 3 9 2 comprenant $n = 5$ éléments. Avec Python, on a : $L[0] = 8$ … et $L[n-1] = 2$. On parcourt la liste du deuxième au dernier élément. Lorsqu'on est à l'étape k (k variant de 1 à n–1, les éléments précédents $L[k]$ sont déjà triés. Il faut donc insérer cet élément d'indice k dans la liste triée.

Mise en place de l'algorithme :

On envisage deux boucles pour réaliser le tri par insertion :

- **Première boucle d'indice *k*** (k variant de 1 à n–1). Quand on considère l'élément d'indice k, on considère que les éléments précédents sont déjà triés.

  Par exemple, pour $k = 2$. On a la liste suivante : 5 8 3 9 2 avec L[2]=3.

  Les éléments avant L[k] sont déjà triés : 5 8.

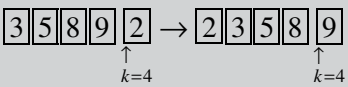
  On pose $x = L[k] = 3$.

- **Deuxième boucle d'indice *i*** (i variant de k–1 à 0). Il faut trouver où l'élément x doit être inséré dans cette liste triée (5 8 dans l'exemple). On compare $L[i]$ à x. Si $L[i] > x$, alors il faut décaler cet élément vers la droite : $L[i+1] = L[i]$. Sinon, il suffit de mettre x dans la valeur du trou qui a été laissé.

Exemple :

Différentes étapes pour la boucle d'indice k :

- liste non triée : 8 5 3 9 2
- $k = 1$: 8 5 3 9 2 → 5 8 3 9 2 (k=1 → k=1)
- $k = 2$: 5 8 3 9 2 → 3 5 8 9 2 (k=2 → k=2)
- $k = 3$: 3 5 8 9 2 → 3 5 8 9 2 (k=3 → k=3)

• $k = 4$: 

On considère une liste non triée d'entiers ou de flottants.

1. Écrire une fonction `tri_insertion` qui admet comme argument une liste `L` et permet de la trier par ordre croissant en utilisant la méthode du tri par insertion.

2. Écrire le programme principal permettant de trier la liste `L=[8, 5, 3, 9, 2]`.

3. Partant d'une liste de couples (entier, chaîne de caractères), on souhaite trier la liste `L2` par ordre croissant de la population en millions d'habitants : `L2=[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`. Écrire une fonction `tri_insertion2` permettant de trier la liste `L2` par ordre croissant de la population.

4. Donner les caractéristiques du tri par insertion.

Analyse du problème

On étudie dans cet exercice le tri par insertion qui est un tri en place car il n'utilise pas de liste auxiliaire. Sa complexité spatiale est faible. On calculera la complexité temporelle (appelée complexité dans les exercices) dans l'exercice 11.6 « Tri par insertion (Mines Ponts IPT 2016) ».

1.

```
def tri_insertion(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1   # deuxième boucle démarre à k-1
        # les éléments entre 0 et k-1 sont triés
        x=L[k]  # mémorisation de la valeur de L[k]
        while i>=0 and L[i]>x:
            L[i+1]=L[i] # décale les éléments de la liste
            i=i-1       # décrémente de 1 la valeur de i
        L[i+1]=x        # met la valeur dans le trou
    # return L est inutile car L est passé en référence
```

Si on modifie un argument d'entrée mutable (liste, dictionnaire, deque, tableau numpy) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing »).

La variable `L` dans la fonction `tri_insertion` est une liste qui est un objet mutable. Il est donc inutile de retourner la variable `L` dans cette fonction !

2.

```
L= [8, 5, 3, 9, 2]
print(L)
tri_insertion(L)
print(L)
```

Python affiche :

```
[2, 3, 5, 8, 9]
```

Remarques :

Le programme principal suivant affiche `None` puisqu'il n'y a pas de `return`.

```
print(tri_insertion([5, 2, 3, 1, 4]))  # affiche None

def tri_insertion_test(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1   # deuxième boucle démarre à k-1
        # les éléments entre 0 et k-1 sont triés
        x=L[k]  #mémorisation de la valeur de L[k]
        while i>=0 and L[i]>x:
            L[i+1]=L[i] # décale les éléments de la liste
            i=i-1       # décrémente de 1 la valeur de i
        L[i+1]=x        # met la valeur dans le trou
    return      # ne retourne pas de variable
```

Le programme principal suivant affiche `None` puisque `return` ne retourne pas de variable.

```
print(tri_insertion_test ([5, 2, 3, 1, 4]))  # affiche None
```

L'instruction `return` est inutile dans cette fonction.

3.

```
def tri_insertion2(L):
    # la fonction trie par ordre croissant la liste L
    # L[i][0] valeur à trier
    # attention : une liste Python est un paramètre passé
    # par référence
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1             # deuxième boucle démarre à k-1
                          # les éléments entre 0 et k-1 sont triés
        x, pays=L[k] # mémorisation de la valeur de L[k]
        while i>=0 and L[i][0]>x:
            L[i+1]=L[i]   # décale les éléments de la liste
            i=i-1         # décrémente de 1 la valeur de i
        L[i+1]=[x, pays]  # met la valeur dans le trou
    # return L est inutile car L est passé en référence
L2=[[67, 'France'], [40, 'Irak'],[47, 'Kenya'], \
  [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]
print(L2)
tri_insertion2(L2)
print(L2)
```

Cours :
On cherche à trier un ensemble d'éléments, c'est-à-dire à les ordonner en fonction d'une relation d'ordre définie sur ces éléments.

- Un tri comparatif est basé sur la comparaison des éléments entre eux.
- Un tri itératif est basé sur un ou plusieurs parcours itératifs de la liste ou du tableau.
- Un tri récursif est basé sur une procédure récursive.
- Un tri en place n'utilise qu'un espace mémoire de taille constante en plus de l'espace servant à stocker les éléments à trier. Il n'utilise pas de liste auxiliaire.
- Un tri stable conserve l'ordre initial des éléments de même clé. Deux éléments avec des clés égales apparaîtront dans le même ordre dans la liste triée et dans la liste non triée.

On rencontre différents algorithmes de tri :

- Tri par insertion : comparatif, itératif, stable. Tri en place.
- Tri par sélection : tri comparatif, itératif, stable. Tri en place.
- Tri rapide : comparatif, récursif, instable. Tri en place.
- Tri par partition-fusion : comparatif, récursif, stable. Le tri n'est pas en place
- Tri par comptage : itératif. Le tri n'est pas comparatif. Le tri n'est pas en place. On n'étudie pas la stabilité pour le tri par comptage.
- Tri à bulles : tri comparatif, itératif, stable. Tri en place.

4. Le tri par insertion est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`.

La liste triée par ordre croissant de la population est : `[[32, 'Pérou'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [67, 'France']]`.

Le tri par insertion est donc stable puisqu'on garde l'ordre alphabétique dans la liste triée pour les pays qui ont la même population.

Le tri par insertion est un tri en place car il n'utilise pas de liste auxiliaire.

Exercice 11.2 : Tri par sélection

On considère une liste `L` contenant n éléments. Le tri par sélection consiste à :

- rechercher le plus petit élément de la liste et le placer en première position ;
- rechercher le deuxième plus petit élément de la liste et le placer en deuxième position ;
- répéter itérativement le processus tel que la liste soit entièrement triée.

Mise en place de l'algorithme :

On parcourt la liste `L[i]` en faisant varier i entre 0 inclus et $n-2$ inclus :

- On cherche `ind_mini` l'indice correspondant à l'élément le plus petit de la liste `L[i : n]`.
- Si `ind_mini` est différent de i, alors on permute `L[i]` avec `L[ind_mini]`.

1. Écrire une fonction `tri_sélection` qui admet comme argument une liste `L` et permet de la trier par ordre croissant en utilisant la méthode du tri par sélection. Écrire le programme principal permettant de trier la liste `L = [8, 5, 3, 9, 2]`.

2. Partant d'une liste de couples (entier, chaîne de caractères), on souhaite trier la liste `L2` par ordre croissant de la population en millions d'habitants : `L2=[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`. Écrire une fonction `tri_sélection2` permettant de trier la liste `L2` par ordre croissant de la population.

3. Donner les caractéristiques du tri par sélection.

4. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_sélection`.

Analyse du problème

On étudie dans cet exercice le tri par sélection, qui est un tri en place car il n'utilise pas de liste auxiliaire. Le premier élément d'une liste `L` a pour indice 0 avec Python.

1.

```
def tri_sélection(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for i in range(0, n-1):
        # recherche du minimum de la liste L[i: n]
        mini=L[i]
        ind_mini=i
        for j in range(i+1, n): # parcourt L[j] pour j>i
            if L[j]<mini:
                ind_mini=j
                mini=L[j]
        # on permute L[i] et L[ind_mini] si on a trouvé
        # un nouveau minimum
        if ind_mini!=i: # ind_mini est différent de i
            L[i], L[ind_mini]=L[ind_mini], L[i]
        # return L est inutile car L est passé en référence

L= [8, 5, 3, 10, 2, 9]
tri_sélection(L)
print(L)
```

2.

```
def tri_sélection2(L):
    # la fonction trie par ordre croissant la liste L
    # L[i][0] valeur à trier
    n=len(L)
    for i in range(0, n-1):
        # recherche du minimum de la liste L[i: n]
```

```
        mini=L[i][0]
        ind_mini=i
        for j in range(i+1, n): # parcourt L[j] pour j>i
            if L[j][0]<mini:
                ind_mini=j
                mini=L[j][0]
        # on permute L[i] et L[ind_mini] si on a trouvé
        # un nouveau minimum
        if ind_mini!=i:         # ind_mini est différent de i
            L[i], L[ind_mini]=L[ind_mini], L[i]
        # return L est inutile car L est passé en référence
```

3. Le tri par sélection est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`.

La liste triée par ordre croissant de la population est : `[[32, 'Pérou'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [67, 'France']]`.

Le tri par sélection est donc stable puisqu'on garde l'ordre alphabétique dans la liste triée pour les pays qui ont la même population.

Le tri par sélection est un tri en place car il n'utilise pas de liste auxiliaire.

4. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n = len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Boucle `for i in range(n-1)` :
  - 3 opérations élémentaires (appel de l'élément `L[i]`, affectation de `mini`, affectation de `ind_mini`) ;
  - boucle `for j in range (i+1, n)` : on se place dans le pire des cas, on a 5 opérations élémentaires (appel de l'élément `L[j]`, test, affectation, appel de l'élément `L[j]` et affectation) ;
  - dans le pire des cas, on a 6 opérations élémentaires (test, appel de l'élément `L[i]`, appel de l'élément `L[ind_mini]` et 3 affectations).

Le nombre total d'opérations élémentaires vaut :

$$2+\sum_{i=0}^{n-2}\left(3+5\left((n-1)-(i+1)\right)+6\right)=-2+\frac{3n}{2}+\frac{5}{2}n^2$$

La complexité est quadratique en $O\left(n^2\right)$.

Remarque :

On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

On peut montrer que la complexité est quadratique dans tous les cas.

Exercice 11.3 : Tri rapide

On considère la liste `L` = |10|3|9|6|8| non triée. On modifie la liste `L` en place. Lors des différents appels récursifs, on travaille sur une partie de la liste `L`. On repère les éléments d'une sous-liste par les pointeurs *a* et *b*.

Fonction pivot :

- On choisit le dernier élément de la liste qui est appelé le pivot *p* (ici *p* = 8) : |10|3|9|6|8| (p)
- On réordonne la liste en plaçant tous les éléments inférieurs ou égaux au pivot à gauche de celui-ci et les éléments strictement supérieurs à droite du pivot. Le pivot *p* est alors à sa bonne place dans la liste à trier.

  L'indice du pivot `ind_p` vaut 0 au début de la procédure.
- On compare `L[0]` = 10 au pivot *p* = 8. Comme `L[0]` > *p*, pas de modification : `ind_p` = 0. |10|3|9|6|8|
- On compare `L[1]` = 3 au pivot *p* = 8. Comme `L[1]` <= *p*, on échange `L[1]` et `L[ind_p]` et on incrémente `ind_p`. On a alors : `ind_p` = 1. |3|10|9|6|8|
- On compare `L[2]` = 9 au pivot *p* = 8. Comme `L[2]` > *p*, pas de changement : `ind_p` = 1. |3|10|9|6|8|
- On compare `L[3]` = 6 au pivot *p* = 8. Comme `L[3]` <= *p*, on échange `L[3]` et `L[ind_p]` et on incrémente `ind_p`. On a alors : `ind_p` = 2. |3|6|9|10|8|
- Pour cette dernière étape, on ne modifie pas `ind_p` mais on permute le dernier élément et `L[ind_p]`. |3|6|8|10|9|

  On est sûr que le pivot est à la bonne place.

Algorithme principal :

Il reste à appliquer la procédure précédente aux deux sous-listes à gauche et à droite du pivot, c'est-à-dire |3|6| et |10|9|.

On a une procédure récursive puisqu'on applique la fonction `pivot` à deux sous-listes. On a des sous-listes de longueur de plus en plus petite. On arrive à une sous-liste comportant un seul élément, qui est donc triée (condition d'arrêt de la fonction récursive) !

1. Écrire une fonction `pivot` qui admet comme arguments une liste `L` et deux indices `a` et `b` permettant de définir une sous-liste de `L` avec des indices compris entre `a` et `b`. Le dernier élément de la sous-liste est appelé le pivot. Cette fonction réordonne les éléments de la sous-liste en plaçant tous les éléments inférieurs ou égaux au pivot à gauche de celui-ci et les éléments strictement supérieurs au pivot à droite de celui-ci. Cette fonction retourne la position du pivot dans la liste.

2. Écrire une fonction récursive `tri_rapide` permettant de trier une liste par ordre croissant en utilisant la fonction `pivot`.

3. Écrire le programme principal permettant de trier la liste `L=[10, 3, 9, 6, 8]`. Représenter l'arbre des appels de la fonction récursive `tri_rapide`.

4. Donner les caractéristiques du tri rapide.

Analyse du problème

Le tri rapide s'appuie sur le principe « diviser pour régner » comme le tri par fusion. On réalise un tri en place.

1.

```
def pivot(L, a, b): # a = indice de début, et b = indice de fin
    # la fonction renvoie la position du pivot dans la liste
    # éléments inférieurs ou égaux sont à gauche du pivot
    # L[i][0] valeur à trier
    p=L[b] # valeur du pivot = dernier élément de la liste
    ind_p=a                    # indice du pivot
    for i in range(a, b):      # i varie entre a inclus et b exclu
        if L[i]<=p:
            L[i], L[ind_p]=L[ind_p], L[i] # on échange les
                                          # 2 éléments
            ind_p+=1
    L[b], L[ind_p]=L[ind_p], L[b] # échange les 2 éléments
        # inutile de retourner L car passage par référence
        # la valeur L[ind_p] est bien placée dans la liste à trier
    return ind_p
```

Si on modifie un argument d'entrée mutable (liste, dictionnaire, deque, tableau numpy) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing »).

La variable `L` dans la fonction `pivot` est une liste qui est un objet mutable. Il est donc inutile de retourner la variable `L` dans cette fonction !

2.

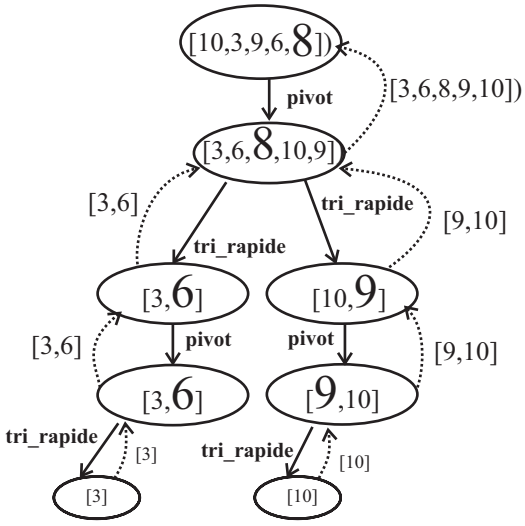
```
def tri_rapide(L, a, b):
    # a = indice de début, et b = indice de fin
    # la fonction trie par ordre croissant la liste L
    if b>a:
        ind_p=pivot(L, a, b)
        tri_rapide(L, a, ind_p-1) # tri_rapide pour les indices
                                  # entre a et ind_p-1
        tri_rapide(L, ind_p+1, b) # tri rapide pour les indices
                                  # entre ind_p+1 et b
    # si a=b alors la sous-liste est triée : condition d'arrêt
```

```
    # si a>b pas de changement. Il faut bien considérer ce cas
    # comme condition d'arrêt
```

3.

```
L=[10, 3, 9, 6, 8]
n=len(L)
print(L)
tri_rapide(L, 0, n-1)  # la fonction retourne none
                       # puisque L est triée en place
print(L)
```

L'arbre ci-dessous représente les différents appels de la fonction `tri_rapide`. La valeur du pivot est représentée avec une taille de police de caractères plus grande.



Remarque :

Lorsqu'on applique la fonction `tri_rapide` à la sous-liste [10, 9] avec $a = 3$ et $b = 4$, il y a trois actions :

- appel de la fonction `pivot` : le pivot vaut 9 et l'indice du pivot vaut `ind_p` = 3 puisque 9 fait partie de la liste `L` = [3, 6, 8, 9, 10] ;
- appel de la fonction `tri_rapide` à la sous-liste définie par $a = 3$ et `ind_p` – 1 = 2. La fonction `tri_rapide` ne modifie rien : c'est une condition d'arrêt ;
- appel de la fonction `tri_rapide` à la sous-liste définie par `ind_p` + 1 = 4 et $b = 4$. La fonction `tri_rapide` ne modifie rien puisque la sous-liste constituée d'un seul élément est déjà triée : c'est une condition d'arrêt.

La complexité du tri rapide est quasi linéaire en $O(n \log n)$. Le tri rapide est plus efficace que le tri par insertion dont la complexité est quadratique en $O(n^2)$ dans le pire des cas.

4. Les caractéristiques du tri rapide sont : comparatif, récursif et instable. Le tri rapide est en place.

On utilise la méthode « diviser pour régner » qui peut se décomposer en trois étapes :

- Diviser (ou partitionner) : on divise le problème initial en plusieurs sous-problèmes.
- Régner : on traite récursivement chacun des sous-problèmes.
- Combiner : on combine les différents sous-problèmes pour résoudre le problème de départ.

Exercice 11.4 : Tri par partition-fusion

On considère une liste `L` non triée d'entiers ou de flottants. On cherche à trier cette liste par ordre croissant en utilisant la méthode du tri par fusion.

Algorithme principal :

On partage la liste initiale `L` de longueur n en deux sous-listes `L1` et `L2` de longueurs $n//2$ et $n-n//2$. On trie de façon récursive les deux sous-listes puis on fusionne les deux sous-listes triées.

Fusion de deux sous-listes L1 et L2 triées :

On considère par exemple `L1=[3, 5, 8]` et `L2=[4, 6]`.

On construit la nouvelle liste `L` en retirant le premier élément de la première liste ou de la deuxième liste.

- Première étape : on considère le premier élément de chaque liste : 3 5 8 et 4 6.

  On cherche le minimum de 3 et 4 que l'on ajoute dans la nouvelle liste : `L=[3]`.

  Il reste alors `L1=[5, 8]` et `L2=[4, 6]`.

- Deuxième étape : on considère le premier élément de chaque liste : 5 8 et 4 6.

  On cherche le minimum de 5 et 4 que l'on ajoute dans la nouvelle liste : `L=[3, 4]`.

  Il reste alors `L1=[5, 8]` et `L2=[6]`.

- Étapes suivantes :

  `L=[3, 4, 5]` ; `L1=[8]` et `L2=[6]`

  `L=[3, 4, 5, 6]` ; `L1=[8]` et `L2=[]`

  `L=[3, 4, 5, 6, 8]` ; `L1=[]` et `L2=[]`

On obtient la liste `L` triée.

1. Écrire une fonction itérative `fusion` qui admet comme arguments deux listes `L1` et `L2` triées et retourne la fusion triée des deux listes.

2. Réécrire une version récursive de la fonction précédente que l'on appellera `fusion_rec`.

3. Écrire une fonction récursive `tri_fusion` permettant de trier la liste `L`. On pourra partager la liste initiale `L` en deux sous-listes `L1` et `L2`.

4. Écrire le programme principal permettant de trier par ordre croissant la liste `L=[8, 3, 5, 1, 9, 5, 12, 15]`. La fonction `tri_fusion` comporte plusieurs appels récursifs. Représenter l'arbre des appels de la fonction `tri_fusion` pour la liste `L`.

5. Donner les caractéristiques du tri par fusion.

Analyse du problème

Le tri par partition-fusion s'appuie sur le principe « diviser pour régner », c'est-à-dire que l'on divise (partitionne) le problème en deux sous-problèmes que l'on sait résoudre. Il reste à utiliser les deux solutions pour résoudre le problème initial.

1.

```
def fusion(L1, L2):
    # la fonction L retourne la fusion triée des deux listes
    # L1 et L2
    i1, i2=0, 0                 # position du pointeur de chaque
                                # liste
    n1, n2=len(L1), len(L2)     # longueur des listes
    n=n1+n2                     # longueur de L1+L2
    L=[]
    while i1+i2<n:              # il faut parcourir tous les
                                # éléments de L1+L2
        if i1==n1:              # la liste L1 est parcourue
                                # entièrement
            return L+L2[i2:]    # il faut ajouter les éléments
                                # restants de L2
        elif i2==n2:            # la liste L2 est parcourue
                                # entièrement
            return L+L1[i1:]    # il faut ajouter les éléments
                                # restants de L1
        elif L1[i1]<L2[i2]:
            L=L+[L1[i1]]        # ajoute L1[i1]
            i1+=1               # incrémente de 1 le pointeur de L1
        else:                   # on a forcément L1[i1]>=L2[i2]
            L=L+[L2[i2]]
            i2+=1
    return L
```

Il faut bien connaître le slicing ou accès par tranches pour les listes : instruction `L[start:stop]` (voir exercice 2.4 « Slicing, tranche, `range`, `np.arange`, `np.linspace` » dans le chapitre « Tableaux numpy – Slicing »).

- `start` désigne l'indice de départ.
- `stop–start` désigne la longueur de la liste extraite (lorsque le pas vaut 1). L'indice final vaut `stop–1` !

Remarque : Il est préférable d'utiliser deux pointeurs `i1` et `i2` plutôt que d'enlever au fur et à mesure les éléments de `L1` et `L2` quand on construit la nouvelle liste `L`. On ne modifie pas les listes `L1` et `L2` en utilisant les pointeurs `i1` et `i2`.

Cours :
Dans toute procédure récursive, l'instruction `return` doit être présente au moins deux fois : une fois pour la condition d'arrêt (premier `return` dans le programme) et une autre fois pour l'appel récursif (dernier `return` dans le programme).

2.

```
def fusion_rec(L1, L2):
    # la fonction retourne la fusion triée des deux listes
    # L1 et L2
    if L1==[]:
        return L2   # condition d'arrêt
    elif L2==[]:
        return L1   # condition d'arrêt
    elif L1[0]<L2[0]:
        return ([L1[0]]+fusion_rec(L1[1:], L2))
    else:
        return ([L2[0]]+fusion_rec(L1, L2[1:]))
```

3.

```
def tri_fusion(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    if n==1:
        return L    # condition d'arrêt
    else:
        L1=L[:n//2] # indices compris entre 0 inclus et n//2 exclu
                    # la longueur de L1 vaut n//2
        L2=L[n//2:] # indices compris entre n//2 inclus et n exclu
                    # de la liste
        return fusion_rec(tri_fusion(L1), tri_fusion(L2))
```

4.

```
L=[8, 3, 5, 1, 9, 5, 12, 15]
L_tri=tri_fusion(L)
print(L_tri)
```

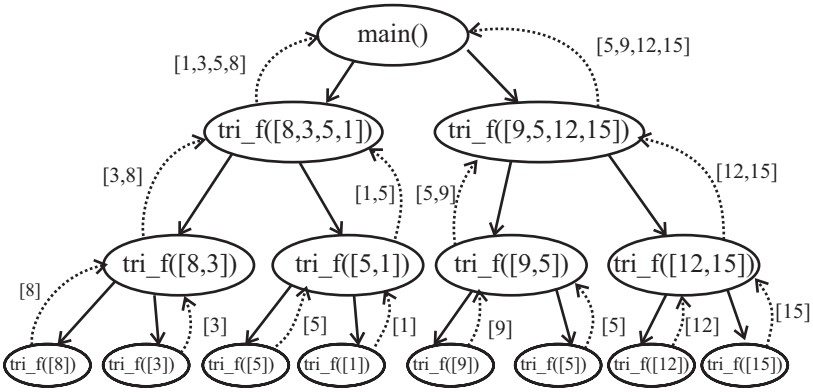
Remarque :

À chaque appel de la fonction récursive, on coupe la liste en deux. On arrive toujours à une sous-liste comportant un seul élément, qui est donc triée (condition d'arrêt de la fonction récursive) !

La complexité spatiale est très mauvaise puisqu'on utilise une fonction récursive qui appelle elle-même une fonction récursive. Le tri n'est pas en place comme avec le tri par insertion.

1er appel de la fonction `tri_fusion` : `L1 = [8, 3, 5, 1]` et `L2 = [9, 5, 12, 15]`. Avant de fusionner `L1` et `L2`, il faut les trier de façon récursive en appelant la fonction `tri_fusion`.

L'arbre ci-dessous représente les différents appels de la fonction `tri_fusion` notée `tri_f`.



Remarque : La complexité du tri par fusion est quasi linéaire en $O(n \log n)$. Le tri par fusion est plus efficace que le tri par insertion dont la complexité est quadratique en $O(n^2)$dans le pire des cas.

5. Les caractéristiques du tri par partition-fusion sont : comparatif, récursif, stable. Le tri n'est pas en place.

On utilise la méthode « diviser pour régner », qui peut se décomposer en trois étapes :

- Diviser (ou partitionner) : on divise le problème initial en plusieurs sous-problèmes.
- Régner : on traite récursivement chacun des sous-problèmes.
- Combiner : on combine les différents sous-problèmes pour résoudre le problème de départ.

Exercice 11.5 : Transformation d'un pseudo-code en programme Python

On considère le pseudo-code suivant :

procédure quicksort

Entrée : Liste L

Sortie : Liste

début
si longueur de L <=1 retourner L
sinon
  pivot ← dernier élément de la liste L
  enlever le dernier élément de la liste L
  créer deux listes vides L1 et L2
  pour x du premier élément au dernier élément de L
    si x <= pivot alors
      ajouter x à la liste L1
    sinon
      ajouter x à la liste L2
  créer une liste vide L3
  ajouter quicksort(L1), x et quicksort(L2) à L3
retourner L3

1. Écrire le programme Python correspondant à ce pseudo-code.

2. On considère la liste `L = [10, 3, 9, 6, 8]`. Écrire le programme principal permettant de trier la liste. Comparer la fonction `quicksort` avec la fonction `tri_rapide` (voir exercice 11.3 « Tri rapide »). Que vaut la liste `L` en sortie de programme ? Représenter l'arbre des appels de la fonction récursive `quicksort`.

3. Proposer une amélioration du programme pour ne pas modifier la liste initiale `L` en sortie de programme.

Analyse du problème

Ce pseudo-code réalise le tri rapide par ordre croissant de la liste `L`. Il ne réalise pas le tri en place.

1.

```
def quicksort(L):
    if len(L)<=1:
        return L   # condition d'arrêt de la fonction récursive
    else:
        n=len(L)   # longueur de la liste L
        p=L[n-1]   # pivot = dernier élément de L.
                   # On pourrait écrire L[-1]
        L.pop()   # on enlève le dernier élément de L
        L1,L2=[],[]   # création de deux listes vides
        for x in L:   # on parcourt les éléments de la liste L
            if x<=p:
                L1+=[x]   # on ajoute x à L1
            else:
                L2+=[x]   # on ajoute x à L2
```

```
    L3=quicksort(L1)+[p]+quicksort(L2)    # concaténation
                                          # de 3 listes
    return L3
```

2.

```
L=[10,3,9,6,8]
L_tri=quicksort(L)
print(L_tri)   # liste triée [3,6,8,9,10]
print(L)   # on obtient L=[10,3,9,6]
```

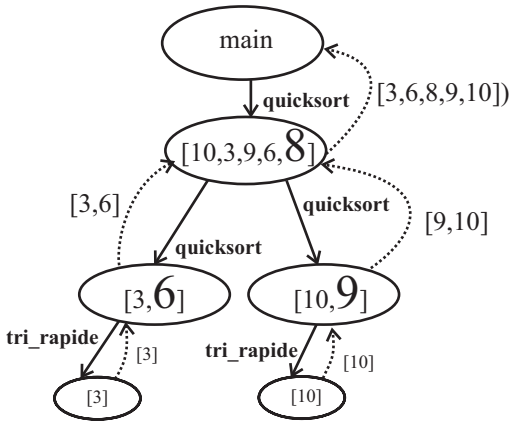
La fonction `quicksort` fonctionne de la même façon que la fonction `tri_rapide` de l'exercice 11.3 « Tri rapide » : on choisit le dernier élément de la liste, qui est appelé le pivot. On place dans la liste `L1` tous les éléments inférieurs ou égaux au pivot et dans la liste `L2` tous les éléments strictement supérieurs au pivot. On crée une liste `L3` qui est la même que celle obtenue à la fin de la fonction `pivot` de l'exercice 11.3 « Tri rapide ». On est sûr que le pivot est à la bonne place.

Il reste à appliquer de façon récursive la fonction `quicksort` aux listes `L1` et `L2` comme avec la fonction `tri_rapide`. On obtient ainsi deux listes `L1` et `L2` triées. Il ne reste plus qu'à retourner la liste `L3 = L1 + [pivot] + L2`.

La différence est que l'on a un tri en place avec la fonction `tri_rapide` alors qu'il y a création de listes `L1`, `L2` et `L3` à chaque appel de la fonction récursive `quicksort`. La liste `L` de départ est modifiée puisqu'on a enlevé le pivot.

On obtient en sortie de programme `L = [10, 3, 9, 6]` et `L_tri = [3, 6, 8, 9, 10]`.

Les différents appels de la fonction `quicksort` sont représentés dans l'arbre ci-dessous. La valeur du pivot est représentée avec une taille de police de caractères plus grande.



Remarque :

Lorsqu'on applique la fonction `quicksort` à [10, 3, 9, 6, 8], le pivot vaut 8.

On obtient : `L1 = [3, 6]` et `L2 = [10, 9]`.

3. La fonction suivante ne modifie pas `L`. On a enlevé l'instruction `L.pop()` et modifié la boucle `for` pour parcourir tous les éléments de la liste `L` sauf le dernier, qui est le pivot.

```
def quicksort2(L):
    if len(L)<=1:
        return L   # condition d'arrêt de la fonction récursive
    else:
        n=len(L)   # longueur de la liste L
        p=L[n-1]   # dernier élément. On pourrait écrire L[-1]
        L1,L2=[],[]   # création de deux listes vides
        for i in range(0,n-1):   # on parcourt tous les éléments
                                 # de la liste L sauf le dernier
            if L[i]<=p:
                L1.append(L[i])   # on ajoute L[i] à L1
            else:
                L2.append(L[i])   # on ajoute L[i] à L2
    L3=quicksort2(L1)+[p]+quicksort2(L2)   # concaténation
                                           # de 3 listes
    return L3
```

Exercice 11.6 : Tri par insertion (Mines Ponts IPT 2016)

On considère la fonction `tri` suivante :

```
1  def tri(L):
2      n=len(L)
3      for i in range(1,n):
4          j = i
5          x=L[i]
6          while 0<j and x<L[j - 1]:
7              L[j] = L[j - 1]
8              j = j - 1
9          L[j]=x
```

1. Lors de l'appel `tri(L)` lorsque `L` est la liste `[5, 2, 3, 1, 4]`, donner le contenu de la liste `L` à la fin de chaque itération de la boucle `for`.

2. Soit `L` une liste non vide d'entiers ou de flottants. Montrer que « la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération `i` » est un invariant de boucle. En déduire que `tri(L)` trie la liste `L`.

3. Évaluer la complexité dans le meilleur et dans le pire des cas de l'appel `tri(L)` en fonction du nombre `n` d'éléments de `L`. Citer un algorithme de tri plus efficace dans le pire des cas. Quelle en est la complexité ?

4. On souhaite, partant d'une liste constituée de couples (chaîne, entier), trier la liste par ordre croissant de l'entier associé selon le fonctionnement suivant :

```
>>> L=[['Brésil', 76], ['Kenya', 26017], ['Ouganda', 8431]]
>>> tri_chaine(L)
>>> L
[['Brésil', 76], ['Ouganda', 8431], ['Kenya', 26017]]
```

Écrire une fonction `tri_chaine` réalisant cette opération.

Analyse du problème

On étudie dans cet exercice le tri par insertion, qui est un tri en place car il n'utilise pas de liste auxiliaire. Cet exercice est extrait du concours Mines Ponts IPT 2016.

Cours :
Un rappel de cours sur le tri par insertion est donné dans l'exercice 11.1 « Tri par insertion ».

1. On considère la liste `L = [5, 2, 3, 1, 4]`. Le nombre d'éléments est $n = 5$.

1re itération : `[2, 5, 3, 1, 4]`

2e itération : `[2, 3, 5, 1, 4]`

3e itération : `[1, 2, 3, 5, 4]`

4e itération : `[1, 2, 3, 4, 5]`

2. On considère la propriété P_i : « la liste `L[0:i+1]` (avec la convention Python) est triée par ordre croissant à l'issue de l'itération `i` » est un invariant de boucle.

Cours :
Pour démontrer la correction d'un programme (voir chapitre 4 « Terminaison – Correction – Complexité »), on utilise une propriété appelée **invariant de boucle**, qui doit avoir trois caractéristiques :

- Être vérifiée avant d'entrer en boucle.
- Si elle est vérifiée avant une itération, alors elle est vérifiée après cette itération.
- Sa vérification en fin de boucle permet d'en déduire que le programme est correct.

- $i = 0$. La liste contient un élément. On a donc une liste triée.

`L[0:1]` permet de faire une extraction de la liste `L`. Voir exercice 2.4 « Slicing, tranche, `range`, `np.arange`, `np.linspace` » dans le chapitre « Tableaux numpy – Slicing ».

- Si la propriété P_i est vérifiée, la liste `L[0:i+1] = [L[0], L[1],… L[i]]` est triée.

Au début de la boucle `for`, on a l'indice `i+1`. Avant la boucle `while`, `j` vaut `i+1` et `x` vaut `L[i+1]`, qui est l'élément à ajouter dans la liste.

À chaque itération de la boucle `while`, `j` décroît de 1.

- La boucle `while` s'arrête dès que `x >= L[j-1]`. On a alors `[L[0], L[1],… L[j-1], L[j-1],… L[i]]`. On obtient à la ligne 9 du programme : `[L[0], L[1],… L[j-1], x,… L[i]]`.
- Si `x` est inférieur à tous les éléments de la liste triée, la boucle `while` s'arrête pour `j = 0`. On a alors `[L[0], L[0]L[1],… L[i]]`. On obtient à la ligne 9 du programme : `[x, L[1],… L[i]]`.

On a bien une liste triée. La propriété P_{i+1} est donc vérifiée.

- En fin de boucle, `i = n-1` alors la liste contenant `n` éléments est triée.

3. On cherche à calculer le nombre d'opérations élémentaires :

Ligne 2 : 2 opérations élémentaires (affectation et fonction `len`).

La variable `i` prend toutes les valeurs de 1 à `n-1` et, pour une valeur `i` fixée, on a :

- Ligne 4 : 1 opération élémentaire (affectation).
- Ligne 5 : 2 opérations élémentaires (affectation et appel de l'élément `L[i]`).
- Ligne 6 : 4 opérations élémentaires (deux tests, opération `and`, appel de l'élément `L[j-1])`.
- Passage dans la boucle `while` (lignes 7 et 8) :

  Dans le meilleur des cas (liste triée par ordre croissant) : pas de passage dans la boucle `while`.

  Dans le pire des cas (liste triée par ordre décroissant, d'où passage `i` fois dans la boucle `while`) : 5 opérations élémentaires (appel de l'élément `L[j-1]`, appel de l'élément `L[j]`, affectation, calcul `j-1`, affectation).
- Ligne 9 : 2 opérations élémentaires (appel de l'élément `L[j]` et affectation).

  Dans le meilleur des cas, le nombre total d'opérations élémentaires est : $2+\sum_{i=1}^{n-1} 9 = 2+9(n-1) = 9n-7$. La complexité dans le meilleur des cas est linéaire en $O(n)$.

  Dans le pire des cas, le nombre total d'opérations élémentaires est : $2+\sum_{i=1}^{n-1}(9+5i) = 2+9(n-1)+5\frac{(n-1)n}{2}$. La complexité dans le pire des cas est quadratique en $O(n^2)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

Le tri par fusion est plus efficace dans le pire des cas puisque la complexité est quasi linéaire en $O(n \log n)$.

4.

```
def tri_chaine(L):
    # la fonction trie par ordre croissant la liste L
    # L[i][1] valeur à trier
    # attention une liste Python est un paramètre passé
    # par référence
    n = len(L)
    for i in range(1, n):
        j = i
        x,y = L[i]
        while 0 < j and y < L[j-1][1]:
            L[j] = L[j-1]
            j = j-1
        L[j] = [x,y]
```

Exercice 11.7 : Tri par comptage, histogramme

Le tri par comptage est un algorithme de tri d'entiers. On considère des entiers de 0 à p dans une liste `L` contenant n éléments : `L=[10, 20, 12, 12, 16, 16, 12, 20, 15]`. L'algorithme compte le nombre d'occurrences de chaque entier.

Les fonctions suivantes permettent le tracé d'histogrammes :

```
import matplotlib.pyplot as plt
plt.hist(L, bins=10)
    # bins = 10 = nombre d'intervalles
```

Mise en place de l'algorithme :

- On définit une liste vide `L_tri`.
- On définit un tableau `HISTO` de p+1 valeurs initialisées à 0.
- On parcourt la liste `L`, on compte le nombre fois qu'apparaît `L[i]` et on incrémente `HISTO[L[i]]` de 1 à chaque fois.
- On parcourt le tableau `HISTO` pour construire au fur et à mesure la liste triée `L_tri`.

1. Écrire une fonction `tri_comptage` réalisant cette opération.

2. Donner les caractéristiques du tri par comptage.

3. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_comptage` en fonction de n et p.

4. Afficher graphiquement l'histogramme de la liste `L`. L'histogramme doit avoir les caractéristiques suivantes :

- Afficher « Valeurs de L » pour l'axe des abscisses et « Nombre d'occurrences » pour l'axe des ordonnées.
- Afficher le titre : « Histogramme de la liste L ».

5. Proposer une amélioration de la fonction `tri_comptage` en tenant compte du minimum et du maximum de la liste `L`.

Analyse du problème

On étudie dans cet exercice le tri par comptage. On crée un tableau `HISTO` qui représente l'histogramme (ou tableau de comptage) des éléments de `L`. Voir exercice 3.2 « Tracé d'un histogramme avec matplotlib » dans le chapitre « Graphiques ».

1.

```
import numpy as np

def tri_comptage(L):
    # la fonction retourne L_tri, qui est la liste L triée par
    # ordre croissant
    n=len(L)              # nombre d'éléments de L
    L_tri=[]
    # recherche du maximum de L[i] noté p
    p=L[0]
    for i in range(n):
        if L[i]>p:
            p=L[i]        # nouvelle valeur du maximum
    HISTO=np.zeros(p+1)   # tableau contenant p+1 valeurs nulles
    # on parcourt la liste L pour incrémenter HISTO
    for i in range(n):
        valeur=L[i]
        HISTO[valeur]+=1  # incrémente HISTO[valeur] de 1
                          # on pourrait écrire : HISTO[L[i]]+=1
    # on parcourt le tableau HISTO pour construire L_tri
    for i in range(p+1):
        if HISTO[i]>0:
            for j in range(int(HISTO[i])):
                L_tri.append(i)
    return L_tri

L=[10, 20, 12, 12, 16, 16, 12, 20, 15]
L1_tri=tri_comptage(L)
print(L1_tri)
```

Le tableau `HISTO` vaut : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 0, 1, 2, 0, 0, 0, 2].

2. Le tri par comptage n'est pas comparatif (on ne compare pas les éléments de la liste entre eux). Le tri est itératif. Le tri n'est pas en place puisqu'il utilise un tableau auxiliaire `HISTO`.

Remarque :

Plusieurs éléments de la liste `L` sont représentés par un unique élément dans l'histogramme. Le tri par comptage ne peut pas être appliqué pour des structures plus complexes telles que `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [32, 'Pérou']]`. On n'étudie donc pas la stabilité pour le tri par comptage.

Le tri par comptage est bien adapté pour des entiers relativement proches les uns des autres.

3. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Ligne `L_tri=[]` : 1 opération élémentaire.
- Boucle `for i in range(n)` : à chaque étape, 4 opérations élémentaires (appel de l'élément `L[i]`, test, appel de l'élément `L[i]` et affectation.

  On a donc $4n$ opérations élémentaires.
- Ligne `HISTO=np.zeros(p+1)` : $p+1$ opérations élémentaires.
- Boucle `for i in range(n)` : à chaque étape, 4 opérations élémentaires (appel de l'élément `L[i]`, affectation, appel de l'élément `HISTO[valeur]`, incrément de 1).

  On a donc $4n$ opérations élémentaires.
- Boucle `for i in range(p+1)` : à chaque étape, on a un test avec 2 opérations élémentaires (appel de l'élément `HISTO[i]` et test).

  L'ajout d'un élément dans `L_tri` se fait au maximum n fois lorsque toutes les étapes de la boucle `for i in range(p+1)` ont été effectuées. Lorsqu'on ajoute un élément dans `L_tri`, on a 3 opérations élémentaires (appel de l'élément `HISTO[i]`, fonction `int`, ajout de i dans `L_tri`).

  On a donc $2(p+1) + 3n$ opérations élémentaires pour la boucle `for i in range(p+1)`.

Le nombre total d'opérations élémentaires est : $2+1+4n+(p+1)+4n+2(p+1)+3n$.

La complexité est linéaire en $O(n+p)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

4.

```
import matplotlib.pyplot as plt
plt.figure()                              # nouvelle fenêtre graphique
plt.hist(L, range=(10, 20), bins=10)
                                          # tracé de l'histogramme
                                          # minimum = 10 et maximum = 20
                                          # avec 10 intervalles
```

```
plt.title('Histogramme de la liste L') # titre de l'histogramme
plt.xlabel('Valeurs de L')
plt.ylabel("Nombre d'occurrences")
plt.show()                             # affiche la figure à l'écran
```

5. On peut réduire le nombre d'éléments du tableau `HISTO` en calculant le minimum et le maximum de `L`, notés respectivement `mini` et `maxi`. Le tableau `HISTO` contient alors `maxi-mini+1` éléments au lieu de `maxi+1` éléments dans la question 2. On a une translation des indices de `mini`.

```
def tri_comptage2(L):
    # la fonction retourne L_tri, qui est la liste L triée par
    # ordre croissant
    n=len(L)                    # nombre d'éléments de L
    L_tri=[]
    # recherche du minimum et du maximum de L[i]
    mini, maxi=L[0], L[0]
    for i in range(n):
        if L[i]<mini:           # nouvelle valeur du minimum
            mini=L[i]
        if L[i]>maxi:
            maxi=L[i]           # nouvelle valeur du maximum
    HISTO=np.zeros(maxi-mini+1)
    # on parcourt la liste L pour incrémenter HISTO
    for i in range(n):
        HISTO[L[i]-mini]+=1     # incrémente HISTO[L[i]-mini] de 1
    # on parcourt le tableau HISTO pour créer L_tri
    for i in range(maxi+1-mini):
        if HISTO[i]>0:
            for j in range(int(HISTO[i])):
                L_tri.append(i+mini)
    return L_tri

L=[10, 20, 12, 12, 16, 16, 12, 20, 15]
L2_tri=tri_comptage2(L)
print(L2_tri)
```

Exercice 11.8 : Tri à bulles

Le tri à bulles consiste à comparer les deux premiers éléments d'une liste `L` et à les échanger s'ils ne sont pas triés par ordre croissant. On recommence ensuite avec le deuxième et le troisième élément de la liste, et ainsi de suite…

1. Écrire une fonction `tri_bulles` réalisant cette opération.

2. Donner les caractéristiques du tri à bulles.

3. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_bulles`.

4. Proposer une amélioration de la fonction `tri_bulles` en tenant compte qu'aucun élément n'est échangé lors d'un parcours d'une liste triée.

Analyse du problème

On étudie dans cet exercice le tri à bulles. On parcourt la liste en comparant les éléments consécutifs deux à deux et en faisant remonter vers la fin de la liste les plus grands éléments. Au bout du premier parcours, l'élément le plus grand est remonté comme une bulle, d'où le nom « tri à bulles ».

1. Soit la liste `L=[10, 30, 12, 12, 16, 12, 20, 15]`. On considère la boucle :

```
for i in range(n-1, 0, -1):  # i varie entre n-1 inclus
                             # et 0 exclu avec pas=-1
```

- Première étape : $i = 7$

  On a une deuxième boucle `for j in range(1, i+1)` : j varie entre 1 inclus et 8 exclu qui permet de comparer deux éléments consécutifs de `L` et de les échanger s'ils sont mal triés.
  L'élément 30 va remonter comme une bulle jusqu'à l'indice n–1 du tableau. On obtient : `L=[10, 12, 12, 16, 12, 20, 15, 30]`. L'élément 30 est donc bien placé.
- Deuxième étape : $i = 6$

  On considère uniquement les éléments : 10, 12, 12, 16, 12, 20, 15 puisque la deuxième boucle fait varier j de 1 inclus à 7 exclu (ou 6 inclus). On obtient : `L=[10, 12, 12, 12, 16, 15, 20, 30]`.
- ...
- Dernière étape : $i = 1$. On obtient : `[10, 12, 12, 12, 15, 16, 20, 30]`.

```
def tri_bulles(L):
    # la fonction trie par ordre croissant la liste L
    # attention : une liste Python est un paramètre passé par
    # référence
    n=len(L)
    for i in range(n-1, 0, -1): # i varie entre n-1 inclus
                                # et 0 exclu avec pas=-1
        for j in range(1, i+1): # j varie entre 1 inclus
                                # et i+1 exclu
            if L[j-1]>L[j]:
                L[j-1], L[j]=L[j], L[j-1] # échange de L[j-1]
                                          # et L[j]
L=[10, 30, 12, 12, 16, 12, 20, 15]
print(L)
tri_bulles(L)
print(L)
```

Remarque : On peut parcourir le tableau à trier à l'envers. On compare deux éléments consécutifs deux à deux et on fait remonter vers le début de la liste les plus petits éléments.

2. Le tri à bulles est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`.

La liste triée par ordre croissant de la population est : `[[32, 'Pérou'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [67, 'France']]`.

Pour trier la liste précédente, il suffit de remplacer la ligne `if L[j-1]>L[j]` par `if L[j-1][0]>L[j][0]`.

Le tri à bulles est donc stable puisqu'on garde l'ordre alphabétique dans la liste triée pour les pays qui ont la même population.

Le tri à bulles est un tri en place car il n'utilise pas de liste auxiliaire.

3. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Boucle `for i in range(n-1, 0, -1)` :
  - Boucle `for j in range (1, i+1)` : on se place dans le pire des cas. On a 7 opérations élémentaires (appel de l'élément `L[j]`, appel de l'élément `L[j-1]`, test, appel de l'élément `L[j]`, appel de l'élément `L[j-1]`, 2 affectations).

Le nombre total d'opérations élémentaires vaut : $2+\sum_{i=1}^{n-1} 7i = \frac{7}{2}n^2 - \frac{7}{2}n + 2$.

La complexité est quadratique en $O(n^2)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

4. Si aucun élément n'est échangé lors d'un parcours d'indice *i*, alors la liste est bien triée.

On ajoute une variable `liste_triée` qui passe à `False` si des éléments de la liste sont échangés. Dans ce cas, la liste n'est pas encore triée pour ce parcours d'indice *i*.

```
def tri_bulles2(L):
    # la fonction trie par ordre croissant la liste L
    # attention : une liste Python est un paramètre passé par
    # référence
    n=len(L)
    for i in range(n-1, 0, -1): # i varie entre n-1 inclus
                                # et 0 exclu avec pas=-1
        liste_triée=True
        for j in range(1, i+1): # j varie entre 1 inclus
                                # et i+1 exclu
            if L[j-1]>L[j]:
                L[j-1], L[j]=L[j], L[j-1] # échange de L[j-1]
                                          # et L[j]
```

```
                liste_triée=False
        if liste_triée==True:
            break  # sort de la boucle for i in range(n-1, 0, -1)
    # return L est inutile car L est passé en référence

L=[5, 2, 3, 1, 4]
print(L)     # affichage de L avant le tri
tri_bulles2(L)
print(L)     # affichage de L après le tri
```

Remarque : L'instruction `break` fait sortir de la boucle `for i in range(n-1, 0, -1)` et passe à l'instruction suivante (`# return L est inutile car L est passé en référence`). Comme il n'y a pas d'instruction après ce commentaire, on sort de la fonction `tri_bulles2`.

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante alors que l'instruction `return` quitte la fonction.

Partie 8

Dictionnaire – Pile – File – Deque

Plan

Dictionnaire – Pile – File – Deque

12

Exercice 12.1 : Opérations de base sur les dictionnaires

On considère des opérations de base sur les dictionnaires.

1. Écrire une fonction `dico_vide` qui renvoie un dictionnaire vide.

2. Écrire une fonction `ajout_cle` qui admet comme arguments un dictionnaire, une clé et une valeur. Cette fonction ajoute le couple (`clé`, `valeur`) au dictionnaire.

3. Écrire une fonction `supp_cle` qui admet comme arguments un dictionnaire et une clé. Cette fonction supprime le couple (`clé`, `valeur`) correspondant à `clé`.

Analyse du problème

Les dictionnaires sont très souvent utilisés en informatique. Chaque élément du dictionnaire a une clé unique. Les éléments du dictionnaire ne sont pas ordonnés.

Cours :

Rappels sur les listes et les tuples

Les éléments d'une liste ou d'un tuple sont ordonnés. Pour récupérer un élément, on utilise un indice.

```
L1=["MPSI", "PTSI"] # liste (objet modifiable) contenant 2 éléments
print(L1[1])        # affiche "PTSI" d'indice 1 dans la liste L1
L2=(48, 46)         # tuple (objet non modifiable) contenant 2 éléments
print(L2[0])        # affiche 48 d'indice 0 dans la liste L2
```

Dictionnaires

Une table de hachage est une structure de données permettant de stocker des couples (`clé`, `valeur`). Elle permet de retrouver une clé très rapidement. Les tables de hachage sont appelées dictionnaires avec Python. Dans un dictionnaire, on associe une valeur à une clé.

Le type de la clé peut être un entier, un nombre flottant, une chaîne de caractères… mais pas une liste ou un tableau.

Le type de la valeur associée à la clé peut être quelconque.

Les éléments d'un dictionnaire ne sont pas ordonnés. On ne peut pas utiliser un indice comme pour les listes pour accéder à un élément.

Chaque élément du dictionnaire est identifié par une clé unique.

On utilise la clé pour rechercher la valeur correspondante du couple (`clé`, `valeur`).

On définit un élément du dictionnaire dans Python en précisant la clé, suivie de « : » et de la valeur associée.

```
dico1={"MPSI":48} # dictionnaire dico1 constitué d'un seul élément
print(dico1)      # affiche {'MPSI': 48}. On visualise la clé
                  # et la valeur
print(type(dico1))# affiche le type de dico1 : dict (type dictionnaire)
```

`dico1` est un objet de type `dict`.

On crée le dictionnaire `dico2` constitué de deux éléments :

```
dico2={"MPSI":48,"PTSI":46}
```

Les éléments sont délimités par des accolades.

Création d'un dictionnaire vide

```
dico_classe={}  # dictionnaire vide avec des accolades {}
```

On peut aussi utiliser l'instruction suivante :

```
dico_classe=dict()
```

Ajout d'un élément dans le dictionnaire

```
dico_classe["MPSI"]=48   # ajoute "MPSI" : 48
dico_classe["MP"]=46     # ajoute "MP" : 46
print(dico_classe)       # affiche {'MPSI': 48, 'MP': 46}
```

La clé est unique dans un dictionnaire. On ne peut pas ajouter "MPSI" : 45 dans `dico_classe`.

Par contre, on peut modifier une valeur :

```
dico_classe["MPSI"]=45
print(dico_classe)     # {'MPSI': 45, 'MP': 46}
```

Suppression d'un élément

```
del dico_classe["MPSI"]
print (dico_classe)  # {'MP': 46}
dico_classe["MPSI"]=48
dico_classe["PCSI"]=48
dico_classe["PTSI"]=46
print (dico_classe)  # {'MP': 46, 'MPSI': 48, 'PCSI': 48, 'PTSI': 46}
```

Teste si une clé est dans un dictionnaire

```
print("PCSI" in dico_classe)  # affiche True
```

Nombre d'éléments d'un dictionnaire

```
print("Nombre d'éléments de dico_classe :", len(dico_classe))
    # Python affiche : Nombre d'éléments de dico_classe : 4
```

Première possibilité pour parcourir les éléments d'un dictionnaire

On utilise `.items()` avec `elt`.

```
for elt in dico_classe.items():          # elt est un tuple
    print("Elément du dictionnaire : ", elt)
    a=elt[0]                             # récupère la clé
    b=elt[1]                             # récupère la valeur
```

Deuxième possibilité pour parcourir les éléments d'un dictionnaire

On utilise `.items()` avec `clé`, `valeur`.

```
for clé, valeur in dico_classe.items():  # dépaquette le tuple
    print("clé :",clé,"valeur :",valeur)
```

On ne peut pas utiliser un indice pour accéder à un élément du dictionnaire alors que `L[i]` permet d'obtenir l'élément d'indice `i` de la liste `L`.

```
L1=dico_classe.keys()   # L1 est de type list
                        # récupère une liste contenant les clés
                        # du dictionnaire
L2=dico_classe.values() # L2 est de type list
                        # récupère une liste contenant les valeurs
                        # du dictionnaire
```

Copie d'un dictionnaire

```
import copy
dico2=dico_classe
dico3=copy.deepcopy(dico_classe)
dico_classe['MP']=48
print(dico_classe['MP'])  # la valeur vaut 48
print(dico2['MP'])        # la valeur vaut 48
print(dico3['MP'])        # la valeur vaut 46
```

La ligne `dico2=dico_classe` n'a pas réalisé une copie de `dico_classe` puisque `dico2` et `dico_classe` pointent vers la même adresse mémoire.

Si on modifie un élément du dictionnaire `dico_classe`, alors cet élément est modifié dans `dico2`.

Par contre, la modification n'apparaît pas dans `dico3` puisque `dico_classe` et `dico3` pointent vers des adresses mémoire différentes.

Les dictionnaires sont des objets mutables (voir exercice 2.2 « Affectation, objet immuable, copie de listes et de tableaux » dans le chapitre « Tableaux numpy – Slicing »).

1.

```
def dico_vide():   # la fonction renvoie un dictionnaire vide
    return dict()  # ou return {}
```

2.

```
def ajout_cle(dico, clé, valeur):
    # la fonction ajoute le couple (clé, valeur) à dico
    dico[clé]=valeur
```

3.

```
def supp_cle(dico, clé):
    # la fonction supprime le couple (clé, valeur)
    # correspondant à clé pour dico
    del dico[clé]
```

Remarque :

Le programme suivant permet de tester les fonctions précédentes :

```
dico=dico_vide()
print(dico)
ajout_cle(dico,"MPSI",48)
ajout_cle(dico,"MP",46)
ajout_cle(dico,"PCSI1",48)
ajout_cle(dico,"PCSI2",48)
print(dico)
print()
supp_cle(dico,"MPSI")
print(dico)
```

Exercice 12.2 : Comptage des éléments d'un tableau à l'aide d'un dictionnaire

On considère le tableau de nombres :

```
import numpy as np
L=np.array([10, 12, 10, 8, 6, 10, 12, -5, 8.2, 8.2])
```

1. Écrire une fonction `comptagedico` qui admet comme argument un tableau. Cette fonction renvoie un dictionnaire permettant de connaître le nombre d'occurrences de chaque élément du tableau.

2. Écrire le programme principal permettant d'afficher le nombre d'occurrences de chaque élément du tableau `L`.

Analyse du problème

Les éléments du dictionnaire ne sont pas ordonnés. Chaque élément du dictionnaire a une clé unique. La valeur de la clé est égale au nombre d'occurrences de la clé dans le tableau.

1.

```
def comptagedico(L):
    # la fonction renvoie un dictionnaire permettant de connaître
    # le nombre d'occurrences de chaque élément du tableau,
    # de la liste ou de la chaîne de caractères L
    dico=dict()       # création d'un dictionnaire vide ou dico={}
    for i in range(len(L)): # i varie entre 0 inclus
                            # et len(L) exclu
        elt=L[i]         # élément d'indice i de la liste L
        if elt in dico:
```

```
            dico[elt]=dico[elt]+1
                        # incrémente de 1 la valeur de la clé elt
        else:
            dico[elt]=1  # ajoute clé elt au dictionnaire
                        # elt apparaît la première fois dans dico
                        # la valeur de la clé elt vaut 1
    return dico
```

2.

```
import numpy as np          # bibliothèque numpy renommée np
L=np.array([10, 12, 10, 8, 6, 10, 12, -5, 8.2, 8.2])
                            # tableau numpy de type ndarray
dico=comptagedico(L)
print(dico)                 # affichage du dictionnaire
```

Le programme Python affiche : `{10.0: 3, 12.0: 2, 8.0: 1, 6.0: 1, -5.0: 1, 8.2: 2}`.

```
mot="C'est un mot"
dico=comptagedico(mot)
print(dico)
```

Le programme Python affiche : `{'C': 1, "'": 1, 'e': 1, 's': 1, 't': 2, ' ': 2, 'u': 1, 'n': 1, 'm': 1, 'o': 1}`.

L'avantage d'utiliser un dictionnaire pour stocker le nombre d'occurrences est qu'il n'est pas nécessaire de connaître à l'avance les éléments de `mot`. On n'utilise de la place mémoire que pour les caractères qui apparaissent réellement dans `mot`.

Exercice 12.3 : Opérations de base sur les piles

On considère trois opérations de base sur les piles. On modélise une pile avec une liste `P` dont on ne peut ajouter et supprimer un élément qu'à une extrémité appelée sommet de pile (ou tête de pile). On utilise `P=[]` pour créer une pile vide.

1. Écrire une fonction `empiler` qui admet comme arguments une pile `P` et un élément `x`. Cette fonction ajoute l'élément `x` au sommet de la pile `P`.

2. Écrire une fonction `depiler` qui admet comme argument une pile `P` non vide. Cette fonction supprime le dernier élément entré dans la pile et retourne cet élément.

3. Écrire une fonction `pile_vide` qui admet comme argument une pile `P`. Cette fonction retourne `True` si la pile est vide et `False` sinon.

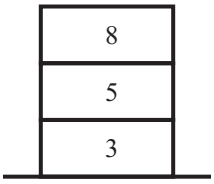
Analyse du problème

Les piles sont très souvent utilisées en informatique. On étudie dans ce chapitre une modélisation des piles avec des listes. Toutes les opérations sur la pile sont

effectuées sur la même extrémité : on utilise le principe LIFO (Last In, First Out). On peut envisager d'autres modélisations, avec des tableaux par exemple. Dans ce cas, il faut prévoir à l'avance le nombre maximum d'éléments à insérer. On utilisera les piles dans le parcours des graphes.

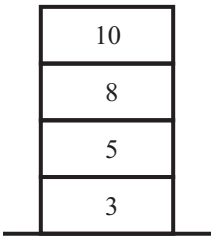
Cours :
Une pile est une structure de données qui utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »). On peut comprendre le fonctionnement d'une pile en considérant une pile d'assiettes.

- La fonction `empiler` consiste à ajouter une assiette sur le sommet de la pile (ou tête de la pile).
  Soit une pile `P` contenant 3 éléments : 3, 5 et 8.



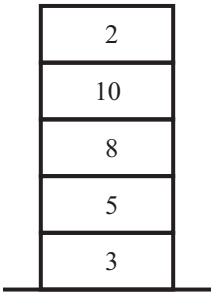
pile P

On souhaite empiler l'élément 10 à la pile `P`. On obtient alors la pile suivante :



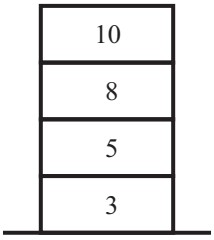
pile P

Si on empile l'élément 2, on obtient :



pile P

- La fonction `depiler` consiste à supprimer un élément de la pile. L'élément à supprimer est toujours situé au sommet de la pile. On a bien une structure LIFO puisque le dernier élément rentré est le premier sorti.
  On obtient alors la pile :



pile P

La fonction « Undo » (Annulation de la frappe) des traitements de texte utilise une pile.

Remarque : L'ajout et la suppression d'un élément en fin de liste Python est très rapide. L'utilisation des listes Python pour gérer des piles est très efficace.

1.

```
def empiler(P,x):
    # la fonction ajoute l'élément x au sommet de la pile P
    P.append(x)   # on ajoute l'élément x à la liste P
```

Si on modifie un argument d'entrée mutable (liste, dictionnaire, deque, tableau numpy) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing »).

La liste `P` dans la fonction `empiler` est un objet mutable. Il est donc inutile de retourner `P` dans cette fonction !

2.

```
def depiler(P):
    # la fonction supprime le dernier élément entré dans la pile P
    # et retourne cet élément
    x=P.pop()    # on supprime le dernier élément de la liste P
    return x     # retourne la valeur du dernier élément
                 # de la liste P
```

3.

```
def pile_vide(P):
    # la fonction retourne True si la pile P est vide
    # et False sinon
    if P==[]:   # on pourrait écrire : if len(P)==0:
        return True
    else:
        return False
```

Remarque :

Le programme suivant permet de visualiser les étapes du rappel de cours précédent :

```
P1=[]   # création d'une liste vide
print(pile_vide(P1))
empiler(P1,3)
empiler(P1,5)
empiler(P1,8)
empiler(P1,10)
empiler(P1,2)
print(P1)
y=depiler(P1)
print(y)
print(P1)
print(pile_vide(P1))
```

Exercice 12.4 : Manipulations de piles

Pour manipuler les piles, on utilisera exclusivement les trois fonctions décrites dans l'exercice précédent « Opérations de base sur les piles » : `empiler`, `depiler` et `pile_vide` ainsi que `P = []` pour créer une pile `P`.

1. Écrire une fonction `copier_coller` qui admet comme argument une pile `P`. Cette fonction renvoie une copie sans modifier la pile initiale. On pourra utiliser la fonction `copier_coller` dans les autres questions.

2. Écrire une fonction `inverser` qui admet comme argument une pile `P`. Cette fonction renvoie une pile `Q` contenant les éléments de `P` dans l'ordre inverse, sans modifier la pile initiale.

3. Écrire une fonction `rotation` qui admet comme argument une pile `P`. Cette fonction renvoie une pile `Q` contenant les éléments de `P` avec une rotation. L'élément au sommet passe en queue de pile et tous les autres éléments sont décalés. Cette fonction ne modifie pas la pile initiale.

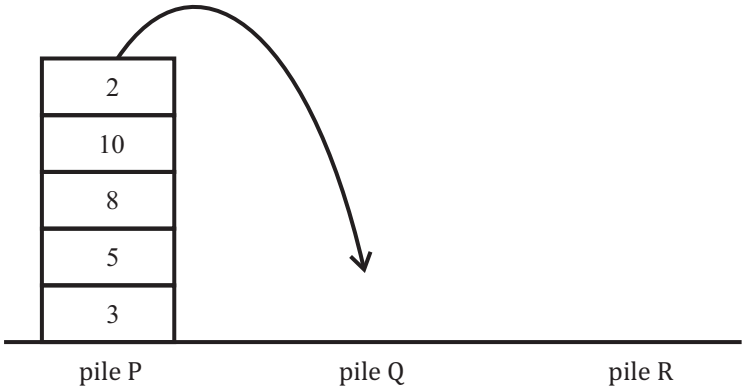
4. Écrire une fonction `dernier_premier` qui admet comme argument une pile `P`. Cette fonction renvoie une pile `Q` telle que le dernier élément inséré dans `P` est échangé avec le premier élément inséré dans `P`.

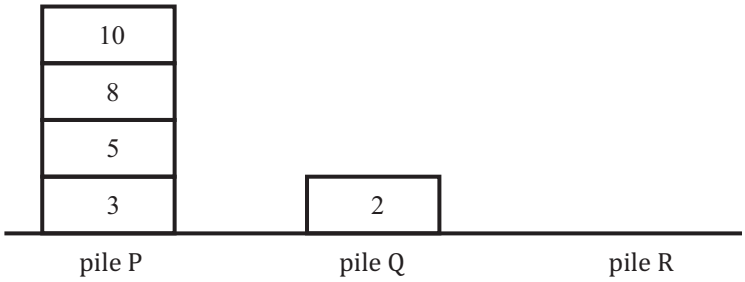
Analyse du problème

Lorsqu'on utilise une pile, on ne peut pas enlever un élément quelconque d'une pile mais uniquement le dernier élément rentré (pile LIFO : Last In, First Out). Il faut donc dépiler cette pile dans une autre pile pour récupérer au fur et à mesure les éléments. Les éléments pourront être stockés dans une variable ou une autre pile.

1.

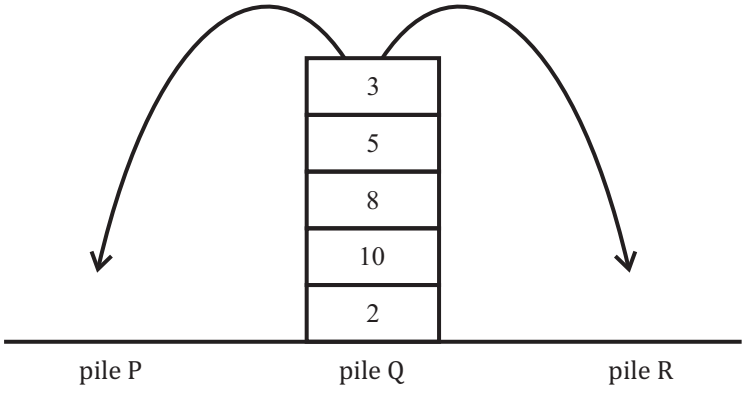
On dépile le premier élément de `P` que l'on empile dans `Q`.

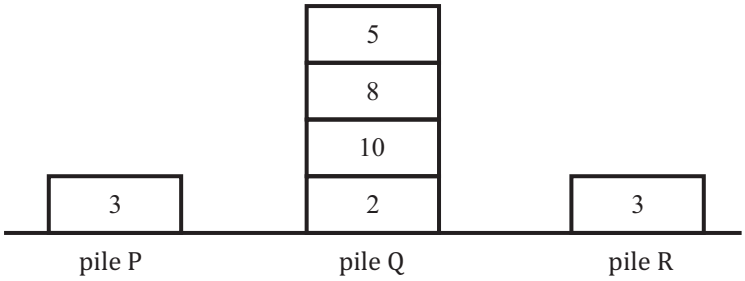




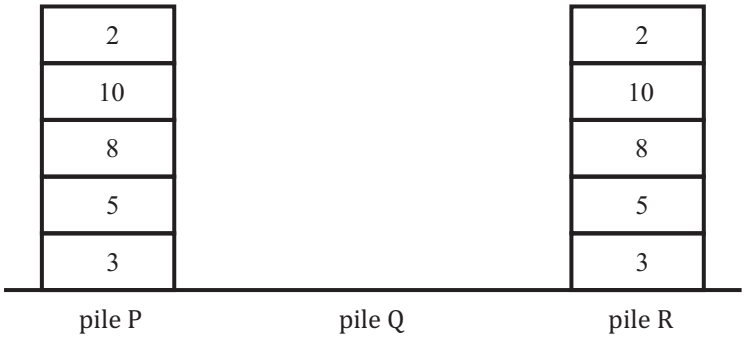
On continue à dépiler tous les éléments de `P` et à les empiler au fur et à mesure dans `Q`.

On dépile le premier élément de `Q` que l'on empile dans `P` et `R`.





On recommence la même opération en dépilant tous les éléments de `Q` et en les empilant dans `P` et `R`.



La fonction `copier_coller` retourne la pile `R`, qui est une copie de la pile `P`.

```
def copier_coller(P):
    Q=[]   # initialisation de la pile Q
    R=[]   # initialisation de la pile R
    # on dépile les éléments de P dans Q
    # on obtient la liste Q avec les éléments inversés
    while pile_vide(P)==False:
        x=depiler(P)
        empiler(Q,x)

    # on dépile les éléments de Q dans R et P
    # on obtient les listes R et P dans le bon ordre
    while pile_vide(Q)==False:
        x=depiler(Q)
        empiler(P,x)
        empiler(R,x)
    return R
```

Remarque :

La fonction `copier_coller_pb` est la même que précédemment sauf qu'on a enlevé la ligne « `empiler(P,x)` » :

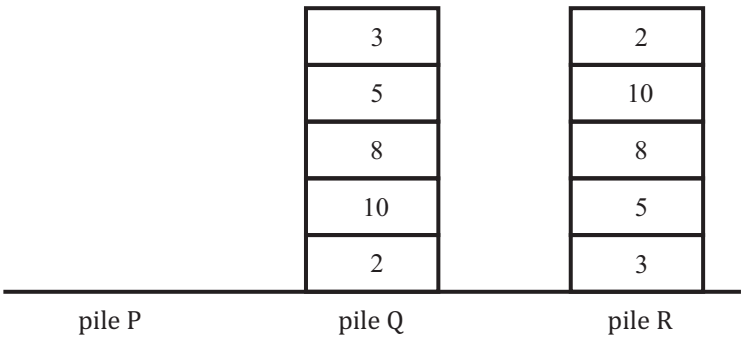
```
def copier_coller_pb(P):
    Q=[]
    R=[]
    while pile_vide(P)==False:
        x=depiler(P)
        empiler(Q,x)
    while pile_vide(Q)==False:
        x=depiler(Q)
       empiler(R,x)
    return R

# programme principal
P=[]
empiler(P,3)
empiler(P,5)
empiler(P,8)
empiler(P,10)
empiler(P,2)
print('Pile P : ',P)
Q=copier_coller_pb(P)
print('Pile P : ',P)
print('Pile Q : ',Q)
```

Python affiche :

```
Pile P : [3, 5, 8, 10, 2]
Pile P : []
Pile Q : [3, 5, 8, 10, 2]
```

Après l'exécution de la fonction `copier_coller_pb`, on ne retrouve plus la liste `P` initiale !

Dans la fonction `copier_coller_pb`, la liste `P` est passée par référence et non par valeur. Lors de l'exécution de cette fonction, on dépile `P` sans jamais rempiler ses éléments. On obtient donc une liste `P` vide.

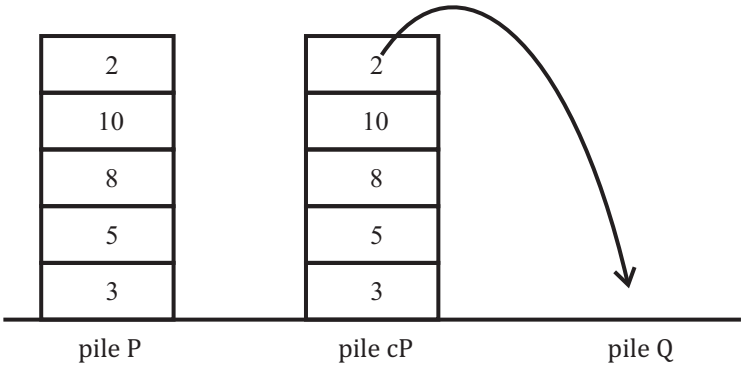


Lorsqu'on manipule les piles dans une fonction, il faut nécessairement dépiler la pile de départ pour obtenir ses éléments. On a deux possibilités pour retrouver la pile de départ lorsqu'on quitte la fonction :

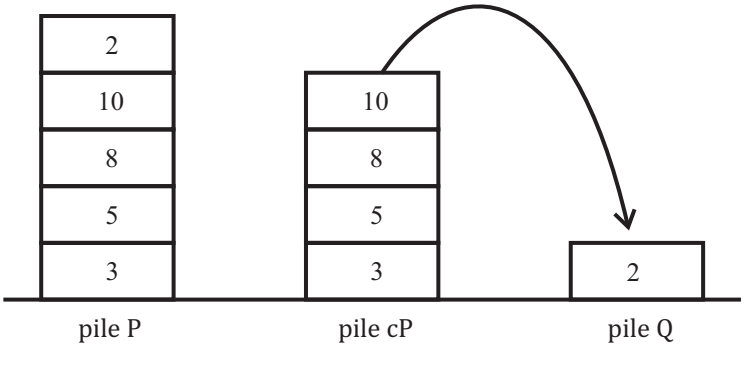
- utiliser la fonction `copier_coller` et dépiler la copie ;
- recréer la pile de départ dans le corps de la fonction.

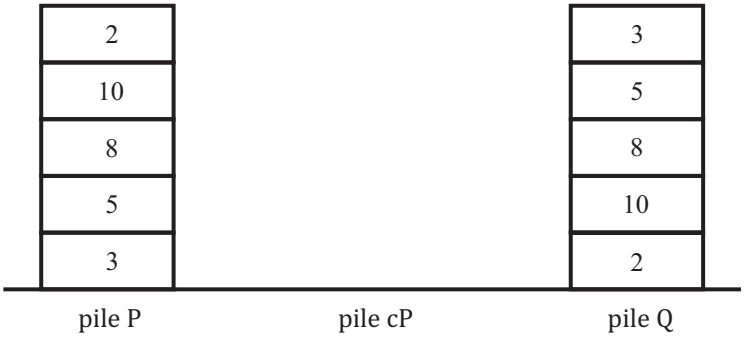
2. On utilise la fonction `copier_coller` pour obtenir la pile `cP`, qui est une copie de la pile `P`.

On dépile le premier élément de `cP`, que l'on empile dans `Q`.



On continue à dépiler tous les éléments de `cP` et à les empiler au fur et à mesure dans `Q`.

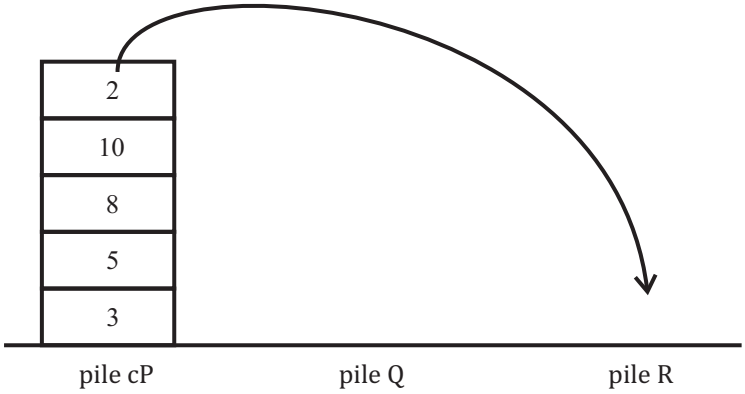




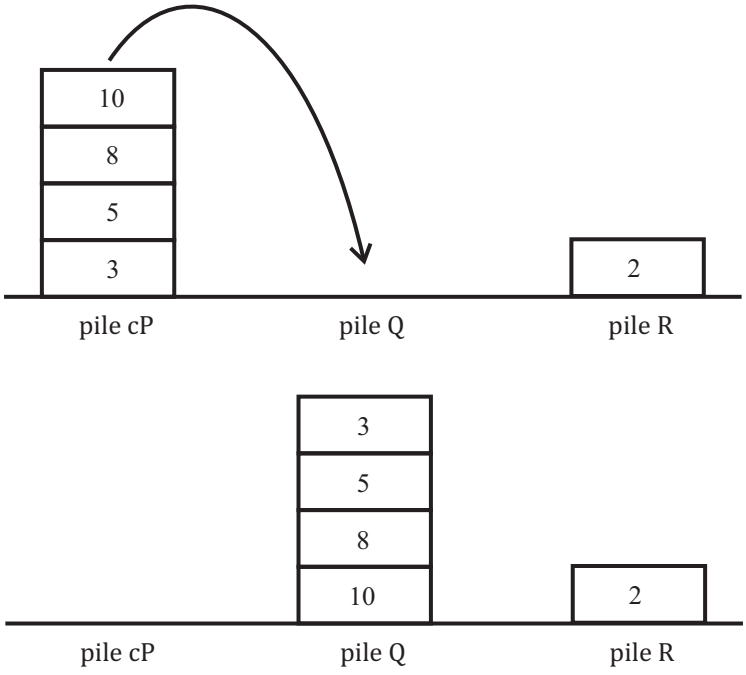
```
def inverser(P):
    cP=copier_coller(P)   # copie de la pile de départ P
    Q=[]
    while pile_vide(cP)==False:
        x=depiler(cP)
        empiler(Q,x)
    return Q
```

3. On utilise la fonction `copier_coller` pour obtenir la pile `cP`, qui est une copie de la pile `P`.

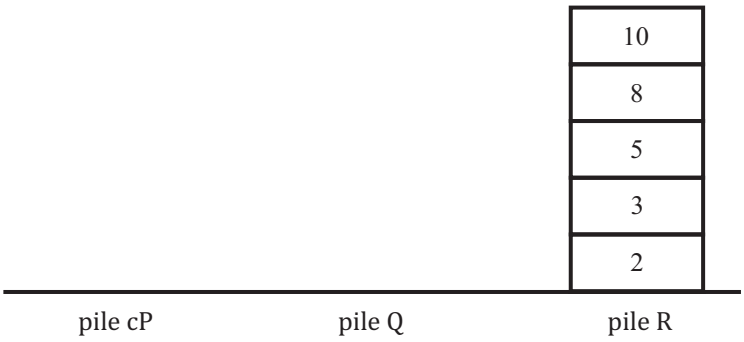
On dépile le premier élément de `cP`, que l'on empile dans `R`.



On dépile les éléments suivants de `cP` et on les empile au fur et à mesure dans `Q`.



Il reste à dépiler tous les éléments de `Q` et à les empiler au fur et à mesure dans `R`.




```
def rotation(P):
    # P contient les éléments : P[0],… P[n-1]
    # on cherche à obtenir R : P[n-1], P[0], P[1],… P[n-2]
    cP=copier_coller(P)   # copie de P
    Q=[]
    R=[]
    x=depiler(cP)   # on récupère P[n-1]
    empiler(R,x)    # on empile le premier élément P[n-1]
    # on dépile cP dans Q en inversant les éléments
    while pile_vide(cP)==False:
        x=depiler(cP)
        empiler(Q,x)
    # on obtient alors Q : P[n-2], P[n-3],… P[0]
    # on dépile Q pour empiler dans R
    while pile_vide(Q)==False:
        x=depiler(Q)
        empiler(R,x)
    return R
```

4.

```
def dernier_premier(P):
    # P contient les éléments : P[0] P[1] P[2]… P[n-1]
    # on cherche à obtenir R : P[n-1], P[1], P[2],… P[n-2], P[0]
    cP=copier_coller(P)   # copie de P
    Q=[]
    R=[]

    x=depiler(cP)   # on récupère P[n-1]
    empiler(R,x)   # on empile le premier élément P[n-1]

    # il faut dépiler cP dans Q en inversant les éléments
    while pile_vide(cP)==False:
        x=depiler(cP)
        empiler(Q,x)
    # on obtient alors Q : P[n-2], P[n-3],… P[0]

    # on dépile un seul élément que l'on garde en mémoire
```

```
    y=depiler(Q)    # y=P[0]

    # on obtient alors Q : P[n-2], P[n-3],… P[1]
    # il faut dépiler Q pour empiler dans R
    while pile_vide(Q)==False:
        x=depiler(Q)
        empiler(R,x)
    # on obtient R : P[n-1], P[1], P[2],… P[n-2]

    # il reste à  ajouter y=P[0]
    empiler(R,y)
    return R
```

Exercice 12.5 : Parenthésage

On cherche à vérifier si une chaîne de caractères `L` est bien parenthésée, c'est-à-dire si le nombre ainsi que l'ordre des parenthèses ouvrantes et fermantes sont corrects. On note « (» la parenthèse ouvrante et «) » la parenthèse fermante.

On utilisera exclusivement les trois fonctions décrites dans l'exercice 12.3 « Opérations de base sur les piles » : `empiler`, `depiler` et `pile_vide` ainsi que `P = []` pour créer une pile `P`.

Écrire une fonction `parenthesage` qui admet comme argument une chaîne de caractères `L`. Cette fonction retourne `True` si la chaîne de caractères est bien parenthésée, `False` sinon.

Exemples :

- `L='(2+8)/(5+9)'`

  `parenthesage(L)` doit retourner `True`.

- `L='(2+8)/((5+9)'`

  `parenthesage(L)` doit retourner `False`.

Analyse du problème

La structure de piles est parfaitement adaptée à la résolution de cet exercice. On parcourt les différents caractères de la chaîne `L`. On empile les parenthèses ouvrantes et on dépile dès qu'on a une parenthèse fermante.

On définit une pile `P` initialement vide. On parcourt tous les caractères de la chaîne `L`.

Dès qu'on rencontre une parenthèse ouvrante, on empile le caractère « (» dans `P`.

Quand on rencontre une parenthèse fermante, plusieurs cas interviennent :

- Si la pile `P` est vide, alors la fonction `parenthesage` retourne `False` puisqu'il manque une parenthèse ouvrante avant la parenthèse fermante.
- Sinon, on dépile la parenthèse ouvrante de `P`.

Lorsqu'on a parcouru tous les caractères de `L`, on doit avoir rencontré autant de parenthèses ouvrantes que fermantes. La pile `P` est nécessairement vide. Si ce n'est pas le cas, la fonction `parenthesage` retourne `False`.

```
def parenthesage(L):
    P=[]   # initialisation de la pile
    for elt in L:
        # on parcourt tous les caractères de L
        if elt=="(":   # parenthèse ouvrante empilée dans P
            empiler(P,elt)
        elif elt==")":   # parenthèse fermante
            if pile_vide(P)==True:
                # la pile ne doit pas être vide
                # il manque une parenthèse ouvrante
                return(False)
            else:
                depiler(P)
                # on dépile la parenthèse ouvrante

    if pile_vide(P)==True:
        return True
    else:
        return False
```

Exercice 12.6 : Notation polonaise inversée

La notation polonaise inversée permet d'écrire des expressions mathématiques sans utiliser de parenthèse. Elle consiste à placer l'opérateur « + », « – », « * » ou « / » juste après ses deux opérandes. On utilise une liste `L` contenant les opérateurs et les opérandes.

On définit une pile `P` initialement vide.

Le programme parcourt tous les éléments de la liste `L`. On rencontre deux cas :

- Si l'élément n'est pas un opérateur, alors on empile l'élément dans `P`.
- Si l'élément est un opérateur, alors on dépile deux éléments de `P`, notés successivement x et y. On effectue l'opération correspondant à l'opérateur : par exemple « $y - x$ » pour l'opérateur « – ». On empile le résultat dans la pile `P`.

La pile contient à la fin un seul élément, correspondant au résultat final.

1. Représenter les différents états de la pile `P` lors de l'appel de la fonction `npi([ 9 , 3 , '+' , 2 , '*' , 4 , -2 , '-' , '/'])`.

2. Écrire une fonction `npi` qui admet comme argument une liste `L`. Cette fonction retourne le résultat correspondant à la notation polonaise inversée.

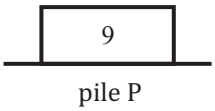
Analyse du problème

La structure de piles est parfaitement adaptée à la résolution de cet exercice. On peut écrire des expressions mathématiques compliquées sans utiliser de parenthèse.

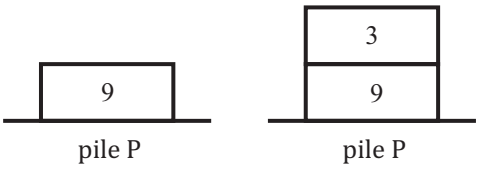
1. On considère la liste `L= [9, 3, '+', 2, '*', 4, -2, '-', '/']`.

La fonction `npi` parcourt successivement les différents caractères de L.

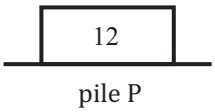
- Premier caractère : on empile 9 dans `P`.



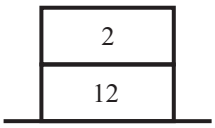
- Deuxième caractère : on empile 3 dans `P`.



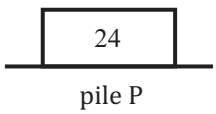
- Troisième caractère : comme le caractère est un opérateur, on dépile `P` ($x = 3$). On dépile à nouveau `P` ($y = 9$). On empile dans `P` le résultat de l'opération $y + x = 12$.



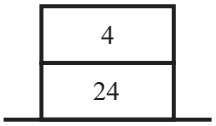
- Quatrième caractère : on empile 2 dans `P`.



- Cinquième caractère : comme le caractère est un opérateur, on dépile `P` ($x = 2$). On dépile à nouveau `P` ($y = 12$). On empile dans `P` le résultat de l'opération $y \times x = 24$.

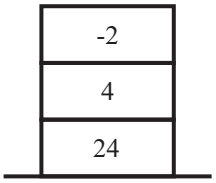


- Sixième caractère : on empile 4 dans `P`.



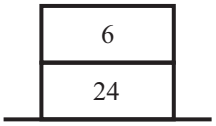
pile P

- Septième caractère : on empile –2 dans `P`.



pile P

- Huitième caractère : comme le caractère est un opérateur, on dépile `P` ($x = -2$). On dépile à nouveau `P` ($y = 4$). On empile dans `P` le résultat de l'opération $y - x = 6$.



pile P

- Neuvième caractère : comme le caractère est un opérateur, on dépile `P` ($x = 6$). On dépile à nouveau `P` ($y = 24$). On empile dans `P` le résultat de l'opération $y / x = 4$.

pile P

On a parcouru toute la liste `L`. La pile `P` contient le résultat du calcul : $\frac{(9+3)\times 2}{4-(-2)} = 4$.

2.

```
def npi(L):
    P=[]   # initialisation de la pile P
    for elt in L:   # on parcourt les éléments de la liste L
        if elt=='+' or elt=='-' or elt=='*' or elt=='/':
            # c'est un opérateur
```

```
            x=depiler(P)
            y=depiler(P)
            if elt=='+':
                res=y+x
            elif elt=='-':
                res=y-x
            elif elt=='*':
                res=y*x
            else:
                res=y/x
            empiler(P,res)
        else:
            empiler(P,elt)
    return(depiler(P))
```

Exercice 12.7 : Opérations de base sur les files

On considère trois opérations de base sur les files. On modélise une file avec une liste `F` dont on ne peut ajouter un élément qu'à une extrémité et supprimer un élément qu'à l'autre extrémité. On rappelle que `F.pop(0)` permet de supprimer `F[0]` dans la liste `F`.

1. Écrire une fonction `enfiler` qui admet comme arguments une file `F`, un élément `x`. Cette fonction ajoute l'élément `x` en queue de file.

2. Écrire une fonction `défiler` qui admet comme argument une file `F` non vide. Cette fonction supprime le premier élément entré dans la file et retourne cet élément.

3. Écrire une fonction `file_vide` qui admet come argument une file `F`. Cette fonction retourne `True` si la file est vide et `False` sinon.

4. On considère le programme suivant :

```
F=[3, 5, 8]
enfiler(F, 10)
print(F)
enfiler(F, 2)
print(F)
x=défiler(F)
print('F :', F, 'x :',x)
print(file_vide(F))
```

Qu'affiche la console Python lors de l'exécution de ce programme ?

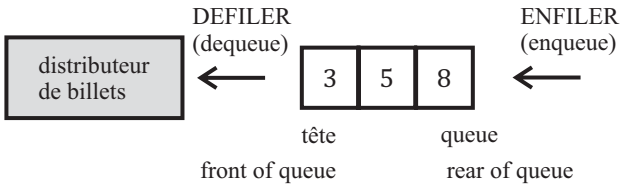
Analyse du problème

Les files sont très souvent utilisées en informatique. On étudie une modélisation des files avec des listes. On utilise le principe FIFO (First In, First Out). On peut envisager d'autres modélisations, avec des tableaux par exemple. Dans ce cas, il faut prévoir à l'avance le nombre maximum d'éléments à insérer. On utilisera les files dans le parcours des graphes.

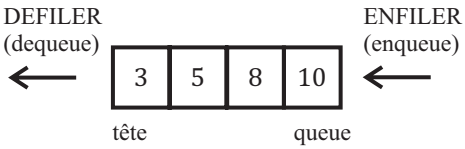
Cours :
Une file (queue en anglais) est une structure de données qui utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

Dans une file d'attente à un distributeur de billets, les personnes font la queue les unes derrière les autres. Le premier arrivé dans la queue est le premier sorti (c'est-à-dire le premier servi pour obtenir les billets).

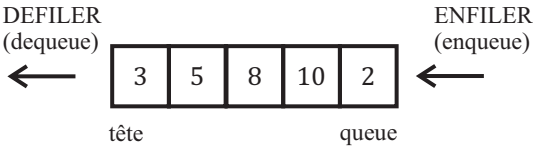
- La fonction `enfiler` (enqueue) consiste à ajouter un élément à la queue de la file (on dit aussi à l'arrière de la file d'attente).
  Soit une file d'attente `F` contenant 3 éléments :



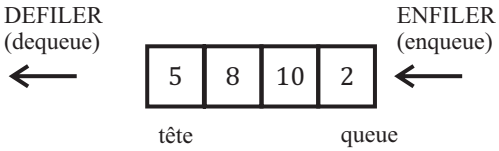
  On enfile l'élément 10 à la file `F`. On obtient alors :



  On enfile l'élément 2, on obtient :



  On modélise la file d'attente par une liste Python : `F = [3, 5, 8, 10, 2]`. On dit que 3 est à la tête de la file `F` et 2 est à la queue de la file `F`.
- La fonction `défiler` (dequeue) consiste à supprimer l'élément situé à la tête de la file (on dit aussi au début de la file d'attente). On a bien une structure FIFO puisque le premier élément rentré est le premier sorti.
  On obtient alors la file :



  La liste `F` s'écrit : `F = [5, 8, 10, 2]`.

Remarque : La suppression d'un élément en tête de liste Python n'est pas très rapide puisque tous les autres éléments doivent être décalés d'une position. On utilisera dans l'exercice suivant « Utilisation de deques » la classe `collections.deque` qui est conçue pour ajouter et supprimer rapidement des éléments aux deux extrémités.

1.

```
def enfiler(F, x):    # F est une liste Python
    F.append(x)       # ajoute x à la queue de la file ou à la fin
                      # de la liste F
```

Si on modifie un argument d'entrée mutable (liste, file, dictionnaire, deque, tableau numpy) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing »).

La liste `F` dans la fonction `enfiler` est un objet mutable. Il est donc inutile de retourner `F` dans cette fonction !

2.

```
def défiler(F):  # F est une liste Python
    x=F.pop(0)   # supprime l'élément situé à la tête de la file F
                 # c'est le premier élément de la liste F
    return x     # retourne x
```

3.

```
def file_vide(F): # F est une liste Python
                  # la fonction retourne True si la file F est vide
                  # et False sinon
    return F==[ ]
```

4. Le programme Python affiche :

```
[3, 5, 8, 10]
[3, 5, 8, 10, 2]
F : [5, 8, 10, 2] x : 3
False
```

Exercice 12.8 : Utilisation de deques

L'instruction `from collections import deque` permet de manipuler des deques.

`D=deque()` permet de créer une deque vide.

`len(D)==0` retourne `True` si la deque est vide, `False` sinon.

On utilise les fonctions : `D.append()`, `D.appendleft()`, `D.pop()` et `D.popleft()` pour manipuler les deques.

1. Écrire une fonction `insere_gauche_deque` d'arguments une deque `D` et un élément *x* permettant d'ajouter *x* à l'extrémité gauche de la deque.

2. Écrire une fonction `insere_droite_deque` d'arguments une deque `D` et un élément *x* permettant d'ajouter *x* à l'extrémité droite de la deque.

3. On considère le programme suivant :

```
L1=[8, 3, 5]
D=deque()
for elt in L1:
    insere_gauche_deque(D, elt)
print(D)
L2=[10, 13, 1]
for elt in L2:
    insere_droite_deque(D, elt)
print(D)
E=deque([3, 2])
print(len(E)==0)
```

Qu'affiche la console Python lors de l'exécution de ce programme ?

Analyse du problème

On considère les deques (double-ended queue), qui sont une généralisation des piles et des files. Les deques permettent d'ajouter et de supprimer très rapidement des éléments aux deux extrémités. On utilisera les deques dans le parcours des graphes.

Cours :

Une deque (se prononce « dèque ») est une structure de données qui généralise le fonctionnement des piles et des files. On peut ajouter et supprimer des éléments aux deux extrémités.

```
from collections import deque    # module permettant d'utiliser
                                 # les deques
```

Pour créer une deque vide :

```
D=deque() # création d'une deque vide
```

On peut ajouter des éléments aux extrémités droite et gauche de la deque.

```
D.append(3)      # ajoute un élément à l'extrémité droite de D
D.append(5)      # ajoute un élément à l'extrémité droite de D
D.appendleft(8)  # ajoute un élément à l'extrémité gauche de D
D.appendleft(10) # ajoute un élément à l'extrémité gauche de D
```

On obtient alors la deque suivante :

10	8	3	5

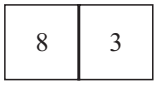
On a un nouveau type : `deque`. Ne pas confondre avec le type `list`.

```
print(type(D))   # le type de D est : deque
```

On peut supprimer des éléments à l'extrémité gauche ou droite de la deque.

```
x=D.pop()      # supprime et renvoie 5, l'élément à l'extrémité droite
               # de la deque
y=D.popleft() # supprime et renvoie 10, l'élément à l'extrémité gauche
               # de la deque
```

On obtient alors la deque suivante :




```
print('Teste si D est vide : ', len(D)==0) # teste si la deque est vide
```

On obtient sur l'afficheur : `False`

```
F=deque()      # création d'une deque vide
print('Teste si F est vide : ', len(F)==0) # teste si la deque est vide
```

On obtient sur l'afficheur : `True`

Dans Python, les deques (comme les listes, dictionnaires et tableaux numpy) sont des objets mutables.

```
E=D
```

`D` et `E` sont des objets mutables. Si on modifie `E`, alors `D` est également modifié puisque `D` et `E` font référence à la même adresse mémoire (voir exercice 2.3 « Passage par référence pour les listes et les tableaux numpy, effet de bord » dans le chapitre « Tableaux numpy – Slicing »).

```
E.pop() # supprime l'élément à l'extrémité droite de la deque E
        # mais aussi de D puisque D et E pointent vers la même
        # adresse mémoire
print('Teste si D = E :', D==E)  # affiche True
```

Pour réaliser une copie profonde de `D`, il ne faut pas écrire `E = D` mais utiliser la fonction `deepcopy`.

```
import copy
E=copy.deepcopy(D)  # copie profonde de D
E.pop()         # supprime l'élément à l'extrémité droite de la deque E
# D n'est pas modifié puisque D ne pointe pas vers la même
# adresse mémoire que E
print('Teste si D = E :', D==E)  # affiche False
```

1.

```
def insere_gauche_deque(D, x):
    D.appendleft(x) # ajoute x à l'extrémité gauche de la deque D
```

2.

```
def insere_droite_deque(D, x):
    D.append(x)     # ajoute x à l'extrémité droite de la deque D
```

3. Le programme Python affiche :

```
deque([5, 3, 8])
deque([5, 3, 8, 10, 13, 1])
False
```

La deque `E` vaut : 3, 2. L'afficheur retourne `False` puisque la deque `E` n'est pas vide.

Remarque :

L'instruction `D=deque(3)` n'est pas correcte et renvoie un message d'erreur.

`D=deque([3])` définit la deque valant 3.

`D=deque('ijk')` définit la deque valant 'i', 'j', 'k'.

`D=deque(['ijk'])` définit la deque valant 'ijk'.

`D=deque([['abc'], ['ijk']])` définit la deque valant ['abc'], ['ijk'].

`x=D.popleft()` permet de supprimer ['abc'] et d'obtenir `x = ['abc']`.

Partie 9

Graphes

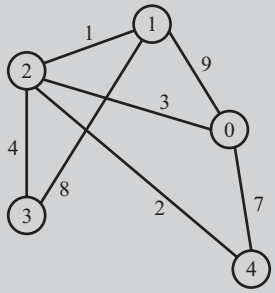
Plan

Graphes

13

Exercice 13.1 : Matrice d'adjacence

On considère le graphe $G = (S, A)$ non orienté, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



1. Construire la matrice d'adjacence $\left(\mathbf{M}_{i,j}\right)_{0 \leq i,j \leq 4}$ (appelée également **matrice de distances**) du graphe G, définie par :

Pour tous les indices i, j, $\mathbf{M}_{i,j}$ représente la distance entre les sommets i et j, ou encore la longueur de l'arête reliant les sommets i et j.

On convient que, lorsque les sommets ne sont pas reliés, cette distance vaut l'infini. On définit la variable `inf = float("inf")` qui représente une distance infinie.

Écrire la matrice **M** en utilisant un tableau numpy de type `ndarray`.

2. Écrire une fonction `voisins`, d'arguments une matrice d'adjacence `M`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degré`, d'arguments une matrice d'adjacence `M`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

4. Écrire une fonction `longueur`, d'arguments une matrice d'adjacence `M`, une liste `L` de sommets de G, renvoyant la longueur du trajet décrit par la liste `L`, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la fonction renverra -1.

Analyse du problème

On définit une matrice de distances contenant les distances entre les différents sommets. Si deux sommets i et j ne sont pas reliés, alors $\mathbf{M}[i, j] = \text{inf}$. Cet exercice est

extrait du concours banque PT 2015 Sujet 0. Comme le graphe n'est pas orienté, la matrice est symétrique : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

Cours :
Un **graphe** *G* est un schéma contenant des points appelés **sommets** (ou **nœuds** ou points), reliés ou non par des arêtes (ou segments ou liens ou lignes).

On utilise la notation suivante : $\boldsymbol{G = (S, A)}$ est un couple d'ensemble finis, dont :

- *S* est l'ensemble des sommets de *G* ;
- *A* est l'ensemble des arêtes de *G*.

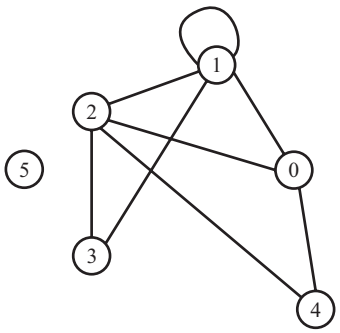
Si une arête relie les sommets *s* et *s'*, on dit que **les sommets *s* et *s'* sont voisins ou adjacents**.

L'**ordre** d'un graphe est le nombre total de sommets.

Un graphe est orienté si les arêtes sont orientées, c'est-à-dire si on ne peut les parcourir que dans un sens.

Pour les graphes non orientés :

- Deux sommets sont **adjacents** lorsqu'ils sont reliés par une **arête**.
- La **taille** d'un graphe non orienté est le nombre total d'arêtes.
- Le **degré** d'un sommet *s*, noté $d(s)$ est égal au nombre d'arêtes dont ce sommet est une extrémité. Une **boucle** est une arête reliant un sommet à lui-même. Les boucles sont comptées deux fois.



  Le degré du sommet 2 vaut 4. Le degré du sommet 1 vaut 5. Le degré du sommet 5 vaut 0 : $d(5) = 0$.
  On n'étudiera par la suite que des graphes sans boucle.
- Une **chaîne** reliant un sommet *s* à un sommet *s'* est une suite d'arêtes consécutives permettant de se rendre de *s* à *s'*.
- La **longueur d'une chaîne** est le nombre d'arêtes de la chaîne, ou la somme des poids des arêtes qui le constituent.
- La **distance** $dist(s, s')$ entre deux sommets *s* et *s'* est la longueur de la plus courte chaîne reliant *s* à *s'*. S'il n'existe pas de chaîne entre *s* et *s'*, alors $dist(s, s') = \infty$, notée inf.
- Une chaîne est simple si toutes les arêtes de la chaîne sont différentes.
- Une chaîne est élémentaire si tous les sommets sont différents sauf pour le sommet d'arrivée, qui peut être confondu avec le sommet de départ (dans le cas des cycles).

- Un **cycle** est une chaîne simple telle que le sommet d'arrivée est le même que le sommet de départ.
- Un sommet *s'* est **accessible** à partir d'un sommet *s* s'il existe une chaîne reliant *s* à *s'*.
- Un **graphe non orienté** est **connexe** (ou simplement connexe) si tous les sommets de *G* sont accessibles entre eux, c'est-à-dire que, pour toute paire de sommets *s* et *s'*, il existe une chaîne reliant *s* à *s'*. Il n'y a pas de sommet isolé.

Pour les graphes orientés :

- Pour un graphe orienté, les arêtes sont orientées. On les appelle des **arcs**. Si on note *s* l'origine de l'arc et *s'* son extrémité, on dit aussi que *s'* est le **successeur** de *s* et *s* le **prédécesseur** de *s'*.
- La **taille** d'un graphe orienté est le nombre total d'arcs.
- Le **degré sortant** d'un sommet *s*, noté $d_+(s)$ est le nombre d'arcs dont *s* est le point de départ.
- Le **degré entrant** d'un sommet *s*, noté $d_-(s)$ est le nombre d'arcs dont *s* est le point d'arrivée.
- Le degré du sommet *s* est : $d(s) = d_+(s) + d_-(s)$
- Un **chemin** reliant un sommet *s* à un sommet *s'* est une suite d'arcs consécutifs permettant de se rendre de *s* à *s'*.
- La l**ongueur d'un chemin** est le nombre d'arcs du chemin, ou la somme des poids des arcs qui le constituent.
- La **distance** $dist(s,s')$ entre deux sommets *s* et *s'* est la longueur du plus court chemin reliant *s* à *s'*. S'il n'existe pas de chemin entre *s* et *s'*, alors $dist(s,s') = \infty$, notée inf.
- Un chemin est simple si tous les arcs du chemin sont différents.
- Un chemin est élémentaire si tous les sommets sont différents sauf pour le sommet d'arrivée, qui peut être confondu avec le sommet de départ (dans le cas des circuits).
- Un **circuit** est un chemin simple tel que le sommet d'arrivée est le même que le sommet de départ.
- Un sommet *s'* est **accessible** à partir d'un sommet *s* s'il existe un chemin reliant *s* à *s'*.
- Un **graphe orienté** est **fortement connexe** si tous les sommets de *G* sont accessibles entre eux, c'est-à-dire que, pour toute paire de sommets *s* et *s'*, il existe un chemin reliant de *s* à *s'* et aussi un chemin de *s'* à *s*. Il n'y a pas de sommet isolé.
- Le poids des arcs peut être négatif. Un **circuit négatif (ou absorbant)** est un circuit pour lequel le poids est négatif, c'est-à-dire que la somme des poids des arcs est négative.

Remarque : Des arêtes reliant la même paire de sommets sont des arêtes multiples. Un graphe est simple s'il ne contient ni boucles ni arêtes multiples. Conformément au programme, on n'étudiera par la suite que des graphes simples.

On appelle **graphe pondéré** un graphe dont les arêtes sont affectées d'un nombre appelé poids (ou coût). Le poids d'un arc peut représenter la distance entre deux sommets voisins pour un réseau routier. Dans certains graphes, le poids des arcs peut être négatif.

On peut implémenter un graphe par une matrice d'adjacence ou une liste d'adjacence :

Matrice d'adjacence :

Dans une matrice d'adjacence $n \times n$, n désigne le nombre de sommets du graphe (ordre du graphe). On peut savoir pour chaque paire de sommets s'ils sont voisins ou non. On rencontre plusieurs possibilités pour définir la matrice d'adjacence :

1) Graphe non pondéré (voir exercice 13.6 « Parcours en largeur d'un graphe avec une matrice d'adjacence ») :
   - Graphe non orienté : Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i, j] = \mathbf{M}[j, i] = 1$ sinon $\mathbf{M}[i, j] = 0$.
   - Graphe orienté : S'il existe un arc allant de i vers j, alors $\mathbf{M}[i, j] = 1$, sinon $\mathbf{M}[i, j] = 0$.
2) Graphe pondéré (voir exercice 14.3 « Algorithme de Dijkstra avec des arcs » dans le chapitre « Recherche du plus court chemin ») :
   - Graphe non orienté : Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i, j] = \mathbf{M}[j, i]$ = distance entre les sommets i et j, sinon $\mathbf{M}[i, j] = \mathbf{M}[j, i]$ = inf.
   - Graphe orienté : S'il existe un arc allant de i vers j, alors $\mathbf{M}[i, j]$ = distance entre les sommets d'origine i et d'extrémité j, sinon cette distance vaut l'infini : $\mathbf{M}[i, j]$ = inf. Le poids des arcs peut être négatif (ceci sera traité dans l'exercice « Algorithme de Bellman-Ford » du livre de deuxième année).

Liste d'adjacence :

On peut utiliser un dictionnaire :

- Graphe non orienté : La clé associée à chaque sommet représente la liste des sommets adjacents (voir exercice 13.3 « Graphe avec liste d'adjacence. Liste des sommets adjacents »).
- Graphe orienté : La clé associée à chaque sommet représente la liste des successeurs (voir exercice « Recherche du plus court chemin – Graphe orienté » dans le chapitre « Recherche du plus court chemin »). On peut également avoir la liste des prédécesseurs de ce sommet.

On peut également utiliser une liste au lieu d'un dictionnaire.

Algorithmes de parcours de graphe :

On va étudier plusieurs algorithmes de parcours de graphe : parcours en largeur (avec une file FIFO – First In, First Out : « premier entré, premier sorti »), parcours en profondeur (avec une pile LIFO – Last In, First Out : « dernier entré, premier sorti »).

Un graphe non orienté, connexe et acyclique est un arbre.

Chaque élément de l'arbre est appelé un nœud. Au niveau élevé, on trouve le nœud racine.

Au niveau juste en dessous, on trouve les nœuds fils (ou descendants). Un nœud n'ayant aucun fils est appelé feuille.

Le nombre de niveaux de l'arbre est appelé hauteur.

On peut construire un graphe à partir d'un autre. Un sous-graphe G' d'un graphe G est composé de certains des sommets de G et de certaines des arêtes de G.

Un graphe H est un **arbre couvrant** du graphe G si H est un arbre que l'on peut obtenir en supprimant des arêtes de G.

Remarque : On peut définir une matrice d'incidence $n \times p$ où n désigne le nombre de sommets du graphe et p le nombre d'arêtes (ou d'arcs).

Les exemples suivants peuvent être modélisés par des graphes :

- Site web : chaque page est un sommet du graphe. Un lien hypertexte est une arête entre deux sommets.

- Réseau ferroviaire : chaque gare est un sommet. Les voies entre deux gares sont des arêtes.
- Réseau routier : chaque ville est un sommet. Les routes entre deux villes sont des arêtes.
- Réseau social : les sommets sont des personnes. Deux personnes sont adjacentes lorsqu'elles sont amies. Le graphe est orienté si l'amitié n'est pas réciproque entre deux personnes.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 9 & 3 & \infty & 7 \\ 9 & 0 & 1 & 8 & \infty \\ 3 & 1 & 0 & 4 & 2 \\ \infty & 8 & 4 & 0 & \infty \\ 7 & \infty & 2 & \infty & 0 \end{pmatrix}$.

Remarque :

Tous les éléments de la diagonale sont nuls puisque la distance entre les sommets i et i est nulle : $\mathbf{M}[i, i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque la distance entre les sommets i et j est la même qu'entre les sommets j et i : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

```
import numpy as np
inf=float("inf")
M=np.array([[0, 9, 3, inf, 7], [9, 0, 1, 8, inf], \
  [3, 1, 0, 4, 2], [inf, 8, 4, 0, inf], [7, inf, 2, inf, 0]])
```

2.

```
def voisins(M, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour la matrice M
    n=np.shape(M)[0]    # nb de lignes de la matrice d'adjacence
    L=[]               # initialisation de la liste L
    for j in range(n):  # j varie entre 0 inclus et n exclu
        if   M[i,j]>0 and M[i,j]<inf:
            L.append(j) # si 0 < distance < inf, on ajoute le
                        # sommet dans L
    return (L)
```

On obtient par exemple `voisins(M, 4)` = [0,2].

3.

```
def degré(M, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour la matrice M
    n=np.shape(M)[0]   # nb de lignes de la matrice d'adjacence
    somme=0            # le nombre d'arêtes vaut 0
    for j in range(n): # j varie entre 0 inclus et n exclu
```

```
        if M[i,j]>0 and M[i,j]<inf:
            somme+=1    # incrémente de 1 le nombre d'arêtes
                        # si distance>0
    return somme
```

On obtient par exemple `degré(2)` = 4. On écrit alors : $d(2) = 4$.

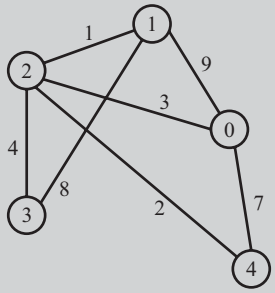
4.

```
def longueur(M, L):
    # la fonction renvoie la longueur du trajet décrit par
    # la liste L pour la matrice M
    somme=0
    n=len(L)
    for i in range(n-1): # i varie entre 0 inclus et n-1 exclu
        if M[L[i],L[i+1]]>=0:
            somme+=M[L[i],L[i+1]]
        else:
            somme=-1
            return somme # la fonction s'arrête et retourne -1
    return somme
```

On obtient par exemple : `longueur(M, [0,1,3,2])` = 21.

Exercice 13.2 : Dictionnaire des sommets adjacents

On considère le graphe $G = (S, A)$ non orienté, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



1. Définir un dictionnaire représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente le dictionnaire des sommets adjacents, dont la valeur de la clé est la distance entre les deux sommets.

2. Écrire une fonction `voisins_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degre_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

4. Écrire une fonction `longueur_dict`, d'arguments un dictionnaire `dico`, une liste `L` de sommets de *G*, renvoyant la longueur du trajet décrit par cette liste `L`, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la fonction renvoie –1.

Analyse du problème

On définit un dictionnaire où chaque clé représente un sommet. Pour un sommet donné, la valeur de la clé est un dictionnaire qui contient l'ensemble des sommets adjacents avec les distances entre les deux sommets.

1. Le dictionnaire est :

```
dico={0:{1:9, 2:3, 4:7},\
      1:{0:9, 2:1, 3:8},\
      2:{0:3, 1:1, 3:4, 4:2},\
      3:{1:8, 2:4},\
      4:{0:7, 2:2} }
```

Voir l'exercice 12.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire – Pile – File – Deque » pour l'utilisation des dictionnaires.

On peut écrire également :

```
dico=dict()
dico[0]={1:9, 2:3, 4:7}      # ajoute la clé 0 dans dico
dico[1]={0:9, 2:1, 3:8}      # ajoute la clé 1 dans dico
dico[2]={0:3, 1:1, 3:4, 4:2} # ajoute la clé 2 dans dico
dico[3]={1:8, 2:4}           # ajoute la clé 3 dans dico
dico[4]={0:7, 2:2}           # ajoute la clé 4 dans dico
```

2.

```
def voisins_dict(dico, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour le dictionnaire dico
    L=[] # initialisation de la liste L
    if i in dico:  # teste si la clé i est dans le diction. dico
        for clé, valeur in dico[i].items():
                    # parcourt les couples (clé, valeur) de dico[i]
            L.append(clé)
    return L
```

On obtient par exemple `voisins_dict(dico, 4)` = [0, 2].

3.

```
def degre_dict(dico, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour le dictionnaire dico
    somme=0
    if i in dico:  # teste si la clé i est dans le diction. dico
        for clé, valeur in dico[i].items():
                    # parcourt les couples (clé, valeur) de dico[i]
            somme+=1
    return somme
```

On obtient par exemple `degre_dict(dico, 2)` = 4. On écrit alors : $d(2) = 4$.

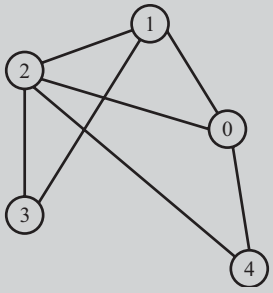
4.

```
def longueur_dict(dico, L):
    # la fonction renvoie la longueur du trajet décrit
    # par la liste L pour le dictionnaire dico
    somme=0
    n=len(L)
    for i in range(n-1): # i varie entre 0 inclus et n-1 exclu
        if L[i] in dico: # teste si la clé L[i] est dans dico
            dico2=dico[L[i]]
            if L[i+1] in dico2:
                somme+=dico2[L[i+1]]
            else:
                somme=-1
                return somme # la fonction s'arrête et retourne -1
        else:
            somme=-1
            return somme     # la fonction s'arrête et retourne -1
    return somme
```

On obtient par exemple : `longueur_dict(dico, [0, 1, 3, 2])` = 21.

Exercice 13.3 : Graphe avec liste d'adjacence. Liste des sommets adjacents

On considère le graphe $G = (S, A)$ non orienté :



1. Définir un dictionnaire représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction `voisins_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degre_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

Analyse du problème

On définit un dictionnaire où chaque clé représente un sommet. Pour un sommet donné, la valeur de la clé est une liste qui contient l'ensemble des sommets adjacents.

1. Pour chaque sommet, on écrit la liste des sommets accessibles.

Le dictionnaire est :

```
dico_liste={0:[1, 2, 4],\
            1:[0, 2, 3],\
            2:[0, 1, 3, 4],\
            3:[1, 2],\
            4:[0, 2]}
```

Le sommet 0 est relié aux sommets 1, 2 et 4. La valeur de la clé 0 est la liste des sommets adjacents [1, 2, 4].

Voir l'exercice 12.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire – Pile – File – Deque » pour l'utilisation des dictionnaires.

2.

```
def voisins_dict_liste(dico, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour le dictionnaire dico
    L=[]          # initialisation de la liste L
    if i in dico:  # teste si la clé i est dans le diction. dico
        for elt in dico[i]:
            L.append(elt)
    return L
```

On obtient par exemple `voisins_dict(dico, "4")` = [0, 2].

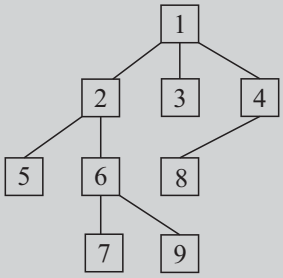
3.

```
def degre_dict_liste(dico, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour le dictionnaire dico
    somme=0
    if i in dico:  # teste si la clé i est dans le diction. dico
        for elt in dico[i]:
            somme+=1
    return somme
```

On obtient par exemple `degre_dict(dico, 2)` = 4. On écrit alors : $d(2) = 4$.

Exercice 13.4 : Parcours en largeur d'un arbre en utilisant une file

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère l'arbre $G = (S, A)$ représentant la structure du site web.



On utilisera les deux opérations de base sur les files : `défiler` pour la suppression d'un élément et `enfiler` pour l'ajout d'un élément. On rappelle que `F.pop(0)` permet de supprimer `F[0]` dans la liste `F`.

On cherche à parcourir en largeur tous les sommets de cet arbre *G* (toutes les pages web de ce site).

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

- Ajouter le sommet de départ dans la file `L` initialement vide.
- Tant que la file `L` n'est pas vide, supprimer l'élément *x* à la tête de la file. Ajouter les fils de *x* dans la file.

1. Définir un dictionnaire `dico` représentant l'arbre *G*. Chaque clé est associée à un sommet *x*. La valeur de la clé représente la liste des fils du sommet *x*.

2. Écrire une fonction `parcourslargeur` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en largeur un arbre.

3. Dans quel ordre sont parcourus les sommets dans la fonction `parcourslargeur(dico, 1)` ?

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet d'explorer un sommet, puis les sommets adjacents à ce sommet et ainsi de suite.

L'implémentation repose sur une file `L` dans laquelle on place le premier sommet à la queue de `L` et les sommets adjacents non explorés à la queue de `L`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

Voir exercice 12.7 « Opérations de base sur les files » dans le chapitre « Dictionnaire – Pile – File – Deque » pour l'implémentation des files par les listes de Python en utilisant le principe FIFO.

Cours :

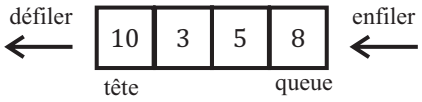
Un graphe non orienté connexe et acyclique est un arbre. Chaque élément de l'arbre est appelé un nœud.

Au niveau élevé, on trouve le nœud racine (1). Au niveau juste en dessous, on a trois nœuds fils (2, 3 et 4). Un nœud n'ayant aucun fils est appelé feuille. Les nœuds 3, 5, 7, 8 et 9 sont des feuilles.

Le nombre total de niveaux de l'arbre est appelé hauteur. La hauteur de l'arbre *G* vaut 4.

G est un arbre ternaire puisque chaque nœud comporte au plus trois fils au niveau inférieur. Du point de vue d'un fils, le nœud dont il est issu au niveau supérieur est appelé père.

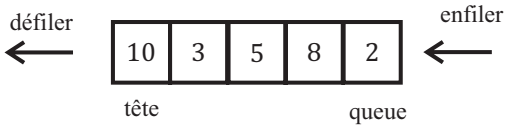
On utilise une liste `L` pour modéliser une file. Par exemple `L = [10, 3, 5, 8]`.



Si on ajoute l'élément 2 (ou si on enfile l'élément 2), on obtient `L = [10, 3, 5, 8, 2]`.

Enfiler un élément à une file consiste à ajouter un élément à la queue de la file.

Défiler un élément d'une file consiste à enlever l'élément situé à la tête de la file.



1. Le premier niveau de l'arbre contient un sommet : 1.
Le deuxième niveau contient trois sommets : 2, 3 et 4.
Le troisième niveau contient trois sommets : 5, 6 et 8.
Le quatrième niveau contient deux sommets : 7 et 9.

```
dico={1:[2, 3, 4], 2:[5, 6], 3:[],4:[8],\
  5:[], 6:[7, 9], 7:[], 8:[],9:[]}
```

Le sommet 2 a deux fils : 5 et 6. La valeur de la clé 2 est la liste des fils : [5, 6].

2. On utilise dans Python les listes pour manipuler les files.

`F.pop(0)` permet de supprimer `F[0]` dans la liste `F`, c'est-à-dire l'élément situé à la tête de `F`.

Remarque : Ne pas confondre avec `L.pop()`, qui permet d'enlever `L[-1]`, c'est-à-dire l'élément situé à la fin de la liste `L`.

```
def enfiler(F, x):
    F.append(x)    # ajoute x à la queue de la file
                   # ou à la fin de la liste F

def défiler(F):
    x=F.pop(0)     # supprime l'élément situé à la tête de F : F[0]
    return x       # retourne x

def parcourslargeur(dico, début):
    # fonction permettant un parcours en largeur du dictionnaire
    # dico en partant de début
    L=[début]          # modélisation d'une file avec la liste L
```

```
while L!=[]:
    x=défiler(L)          # supprime l'élément à la tête de
                          # la file L : L[0]
    for elt in dico[x]:   # ajoute les fils du sommet x
        enfiler(L, elt)   # à la queue de F
```

Remarque :

`dico[x]` contient les fils du sommet *x*. *G* ne contient pas de cycle par définition d'un arbre. Tous les fils sont nécessairement non explorés. On peut donc tous les ajouter dans la file `L`.

On étudiera un graphe dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque » : il faudra tester si les sommets adjacents ont déjà été explorés.

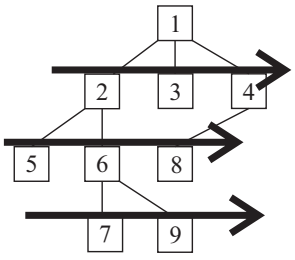
3. `L=[1]` au début de l'algorithme.

- On enlève 1 de la file `L` et on explore ce sommet. Les fils de « 1 » sont ajoutés au fur et à mesure à la queue de `L`. La file `L` vaut alors : `[2, 3, 4]`.
- On enlève 2, le premier élément de `L` (élément situé à la tête de la file). On explore le sommet 2. Les fils de « 2 » sont ajoutés à la queue de `L`. La file `L` vaut alors : `[3, 4, 5, 6]`.
- On enlève 3, le premier élément de `L` (élément situé à la tête de la file). On explore le sommet 3. Il n'y a pas de fils. La file `L` vaut alors : `[4, 5, 6]`.
- On enlève 4, le premier élément de `L` (élément situé à la tête de la file). On explore le sommet 4. Les fils de « 4 » sont ajoutés à la queue de `L`. La file `L` vaut alors : `[5, 6, 8]`.
- On enlève 5, le premier élément de `L` (élément situé à la tête de la file). On explore le sommet 5.
- …

Finalement, on a exploré les sommets dans l'ordre :

- 1,
- 2, 3, 4,
- 5, 6, 8,
- 7, 9.

Les flèches sur le graphe représentent le sens de parcours niveau par niveau et de gauche à droite. On a bien exploré les sommets en largeur.



> On va étudier une amélioration du parcours en largeur pour un graphe non orienté, connexe et possédant des cycles dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque ».

Remarques :

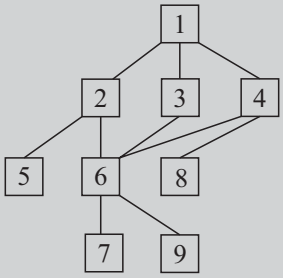
L'ordre de parcours des sommets pour un niveau donné n'a pas d'importance. Ici, on les parcourt dans le sens croissant, soit de gauche à droite.

Le dictionnaire `dico2` ci-dessous contient tous les sommets adjacents à un sommet donné. La fonction `parcourslargeur(G)` ne se termine jamais puisque la boucle `for` ajoute tous les sommets adjacents. Dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque », on va résoudre ce problème en ne considérant que les sommets adjacents non explorés.

```
dico2={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6],\
  4:[1, 6, 8], 5:[2], 6:[2, 7, 9], 7:[6], \
  8:[4], 9:[6]}
```

Exercice 13.5 : Parcours en largeur d'un graphe avec une deque

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



On cherche à parcourir en largeur tous les sommets de ce graphe (toutes les pages web de ce site). On utilise la liste `VISITED` représentant les sommets déjà explorés.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

- Ajouter le sommet de départ dans la deque `D` initialement vide.
- Tant que `D` n'est pas vide, supprimer le sommet *x* à l'extrémité gauche de `D`. Ajouter à l'extrémité droite de `D` les sommets non explorés adjacents au sommet *x*.

1. Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des sommets adjacents.

2. Écrire une fonction `parcourslargeur2` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en largeur un graphe non orienté, connexe.

3. Dans la fonction `parcourslargeur2(dico, 1)`, dans quel ordre sont parcourus les sommets ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un.

L'implémentation repose sur une deque `D` dans laquelle on place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »). Voir exercice 12.8 « Utilisation de deques » dans le chapitre « Dictionnaire – Pile – File – Deque » pour la manipulation des deques.

Le graphe contient des cycles. Pour ne pas explorer plusieurs fois un même sommet, on marque les sommets déjà explorés.

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6],3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9],7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2. Les sommets déjà visités sont marqués pour éviter d'explorer plusieurs fois un même sommet. La liste `VISITED` contient les sommets visités.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque `D` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED=[début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *x* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets non explorés adjacents au sommet *x*.

```
def parcourslargeur2(dico, début):
    # fonction permettant un parcours en largeur du dictionnaire
    # dico en partant de début et retournant
    # VISITED la liste des sommets visités
    from collections import deque # module permettant d'utiliser
                                  # les deques
```

```
D=deque()    # deque vide
D.append(début)   # ajoute le sommet de départ
VISITED=[début]
while len(D)!=0:
    x=D.popleft() # supprime le sommet x à l'extrémité
                  # gauche de D
    L=dico[x]     # liste contenant les sommets adjacents à x
    for elt in L: # parcourt les éléments de L
        if elt not in VISITED:  # teste si le sommet n'a pas
                                # déjà été exploré
            D.append(elt)       # ajoute le sommet elt à
                                # l'extrémité droite de D
            VISITED.append(elt) # le sommet elt a été exploré
```

Remarque :

`dico[x]` contient les sommets adjacents au sommet *x*. *G* peut contenir des cycles puisqu'on ne considère pas d'arbre dans cet exercice. Il faut tester si les sommets adjacents ont déjà été explorés.

Dans l'exercice précédent « Parcours en largeur d'un arbre en utilisant une file », on n'avait pas besoin de tester si les fils étaient déjà explorés puisqu'on considérait un arbre, sans cycle par définition.

3. La deque `D` vaut [1] au début de l'algorithme.

- On supprime l'élément à l'extrémité gauche de `D`. On explore ce sommet. Les sommets adjacents à 1 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [2, 3, 4].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 2. Les sommets adjacents à 2 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [3, 4, 5, 6].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 3. Le sommet 6 a déjà été exploré. On ne l'ajoute pas. La deque `D` vaut alors : [4, 5, 6].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 4. Les sommets adjacents à 4 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [5, 6, 8].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 5. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [6, 8].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 6. La deque `D` vaut alors : [8, 7, 9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 8. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [7, 9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 7. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 9. Pas de nouveau sommet non exploré. La deque `D` est vide.

Avec la fonction `parcourslargeur2(dico, 1)`, on explore les sommets suivants : `VISITED = [1, 2, 3, 4, 5, 6, 8, 7, 9]`.

4. Soit n le nombre de sommets et m le nombre d'arêtes.

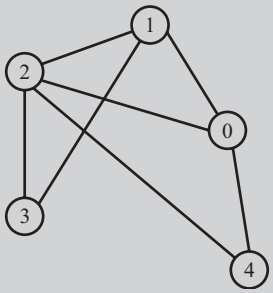
À chaque passage dans la boucle `while len(D)!=0`, on enlève un sommet à l'extrémité gauche de la deque `D`. Cette boucle sera donc exécutée au plus n fois.

À chaque fois que l'on supprime un sommet à l'extrémité gauche de la deque `D`, la boucle `for elt in L` parcourt tous les successeurs du sommet supprimé. Les lignes `D.append(elt)` et `VISITED.append(elt)` seront exécutées au plus une fois pour chaque arête, soit au plus m fois comme il y a m arêtes.

La complexité dans le pire des cas pour un graphe représenté par une liste d'adjacence est linéaire en $O(n + m)$.

Exercice 13.6 : Parcours en largeur d'un graphe avec une matrice d'adjacence

On considère le graphe non orienté $G = (S, A)$:



1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe G. Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i, j] = 1$ sinon $\mathbf{M}[i, j] = 0$.

2. Écrire une fonction `parcourslargeur_mat` d'arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de parcourir en largeur le graphe. On utilise une deque pour parcourir en largeur le graphe.

3. Dans quel ordre sont parcourus les sommets dans la fonction `parcourslargeur_mat(M, 0)` ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. Il utilise une deque `D` dans laquelle il place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
import numpy as np   # bibliothèque numpy renommée np
M=np.array([[0, 1, 1, 0, 1], [1, 0, 1, 1, 0], [1, 1, 0, 1, 1],\
  [0, 1, 1, 0, 0], [1, 0, 1, 0, 0]])
```

Remarque :

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $\mathbf{M}[i, i] = 1$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

2. Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque `D` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED=[début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *i* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets *j* non explorés adjacents au sommet *i*.

```
def parcourslargeur_mat(M, début):
    # fonction permettant un parcours en largeur de la matrice M
    # en partant de la chaîne de caractères début et retournant
    # VISITED la liste des sommets visités
    from collections import deque # module permettant d'utiliser
                                  # les deques
    D=deque()                     # deque vide
    D.append(début)               # ajoute le sommet de départ
    VISITED=[début]               # liste des sommets explorés
    while len(D)!=0:
        i=D.popleft()             # supprime le sommet i à
                                  # l'extrémité gauche de D
        for j in range(len(M[i, :])):
            if M[i, j]==1 and (j not in VISITED):
                D.append(j)       # ajoute le sommet j à
                                  # l'extrémité droite de D
                VISITED.append(j) # le sommet j a été exploré
```

3. La deque `D` vaut [0] au début de l'algorithme.

- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 0. Les sommets adjacents à 0 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [1, 2, 4]. La liste `VISITED` vaut : [0, 1, 2, 4].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 1. Les sommets adjacents à 1 et non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [2, 4, 3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 2. Pas de sommet non exploré adjacent à 2. La deque `D` vaut alors : [4, 3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. Pas de sommet non exploré voisin de 4. La deque `D` vaut alors : [3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. Pas de sommet non exploré voisin de 3. La deque `D` vaut alors : [ ]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].

La boucle `while` est terminée lorsque `D` est vide. Avec la fonction `parcourslargeur_mat`, on explore les sommets suivants : `VISITED` = [0, 1, 2, 4, 3]. On a bien réalisé un parcours en largeur.

4. Soit *n* le nombre de sommets.

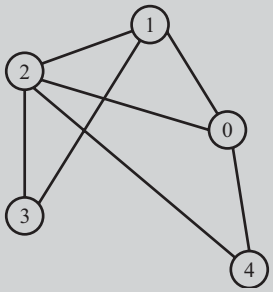
À chaque passage dans la boucle `while len(D)!=0`, on supprime un sommet à l'extrémité gauche de la deque `D`. Cette boucle sera donc exécutée au plus *n* fois.

À chaque itération de la boucle `while`, la boucle `for j in range` est exécuté au maximum *n* fois.

La complexité dans le pire des cas pour un graphe représenté par une matrice d'adjacence est quadratique en $O(n^2)$.

Exercice 13.7 : Test de connexité d'un graphe – Parcours en largeur – Arbre couvrant

On considère le graphe non orienté $G = (S, A)$:



1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe *G*. Si deux sommets différents *i* et *j* sont reliés par une arête, alors $\mathbf{M}[i, j] = 1$ sinon $\mathbf{M}[i, j] = 0$.

2. Écrire une fonction `testgrapheconnexe_largeur` qui admet comme argument une matrice d'adjacence `M` et retournant `True` si le graphe est connexe et `False` sinon. On utilise une deque pour parcourir en largeur le graphe.

3. Écrire une fonction `arbrecouvrant_largeur` d'arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de récupérer une matrice d'adjacence représentant l'arbre couvrant correspondant au parcours en largeur pour un graphe non orienté et connexe.

4. Représenter l'arbre couvrant du graphe *G*.

Analyse du problème

Un graphe est connexe (ou simplement connexe) si on peut relier, directement ou non, n'importe quel sommet à n'importe quel autre sommet du graphe par une chaîne. Il n'y a pas de sommet isolé. On utilise l'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) décrit dans l'exercice 13.5 « Parcours en largeur d'un graphe avec une deque » avec le principe FIFO (First In, First Out : « premier entré, premier sorti »).

On utilise le parcours en largeur pour construire un arbre inclus dans le graphe *G* et qui relie tous les sommets de ce graphe.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
import numpy as np   # bibliothèque numpy renommée np
M=np.array([[0, 1, 1, 0, 1], [1, 0, 1, 1, 0], [1, 1, 0, 1, 1],\
  [0, 1, 1, 0, 0], [1, 0, 1, 0, 0]])
```

Remarque :

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $\mathbf{M}[i, i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

2. L'implémentation repose sur une deque `D` dans laquelle on place le premier sommet dans `D` et les sommets adjacents non explorés à l'extrémité droite de `D`.

Pour savoir si le graphe *G* est connexe, on récupère la liste des sommets explorés avec l'algorithme de parcours en largeur décrit dans l'exercice précédent « Parcours en largeur d'un graphe avec une matrice d'adjacence ». Si tous les sommets du graphe *G* sont dans cette liste, alors *G* est connexe.

```
def testgrapheconnexe_largeur(M):
    # la fonction retourne True si le graphe est connexe
    # et False sinon pour la matrice d'adjacence M
    from collections import deque # module permettant d'utiliser
                                  # les deques
    début=0          # on prend un sommet quelconque par exemple 0
    D=deque()        # deque vide
    D.append(début)  # ajoute le sommet de départ
    VISITED=[début]  # liste des sommets explorés
    while len(D)!=0:
        i=D.popleft() # supprime le sommet i à l'extrémité
                      # gauche de D
        for j in range(len(M[i, :])):
            if M[i,j]==1 and (j not in VISITED):
                D.append(j)  # ajoute le sommet j à l'extrémité
                             # droite de D
                VISITED.append(j) # le sommet j a été exploré
    if len(VISITED)==np.shape(M)[0]:
        return True
    else:
        return False
```

La liste `VISITED` de la fonction `testgrapheconnexe_largeur` donne la liste de tous les sommets explorés avec l'algorithme BFS.

`np.shape(M)[0]` retourne le nombre de lignes de la matrice `M`.

Il suffit de tester si la longueur de la liste `VISITED` est égale au nombre de lignes de la matrice `M` pour savoir si on a bien exploré tous les sommets du graphe.

Remarque : On peut également utiliser l'algorithme du parcours en profondeur pour explorer tous les sommets du graphe. Voir exercice 13.9 « Test de connexité d'un graphe – Parcours en profondeur – Arbre couvrant ».

3. L'arbre couvrant d'un graphe non orienté et connexe est un arbre inclus dans le graphe et qui relie tous les sommets du graphe *G*.

On considère deux listes : `PERE`, `VISITED` et une deque `D`. La liste `VISITED` contient la liste des sommets explorés. Les étapes de l'algorithme sont les suivantes :

Initialisation de l'algorithme :

- La liste `PERE` contient –1 : `PERE=[-1]`. On aurait pu écrire `PERE=[None]` au lieu de `PERE=[-1]` puisqu'on n'utilise pas `PERE[0]` par la suite.

- La deque `D` contient le sommet de départ.
- La liste `VISITED` contient le sommet de départ : `VISITED=[début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *i* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets non explorés *j* adjacents au sommet *i*. Pour chaque sommet non exploré adjacent au sommet *i*, on ajoute *i* dans la liste `PERE`. On connaît ainsi la liste des sommets parcourus et comment on atteint ce sommet.

Pour construire l'arbre couvrant, il suffit de parcourir les listes `VISITED` et `PERE` en commençant à `VISITED[1]`. Le père de `VISITED[k]` est `PERE[k]`. On peut ainsi remplir la matrice d'adjacence de l'arbre couvrant.

```
def arbrecouvrant_largeur(M, début):
    # la fonction retourne l'arbre couvrant de la matrice M
    # en partant du sommet début (entier)
    # le graphe est non orienté et connexe
    from collections import deque # module permettant d'utiliser
                                  # les deques
    PERE=[-1]        # on n'utilise pas le premier élément de PERE
    D=deque()        # deque vide
    D.append(début)  # ajoute le sommet de départ
    VISITED=[début]  # liste des sommets explorés
    while len(D)!=0:
        i=D.popleft() # supprime le sommet i à l'extrémité
                      # gauche de D
        for j in range(len(M[i, :])):
            if M[i,j]==1 and (j not in VISITED):
                D.append(j)    # ajoute le sommet j à l'extrémité
                               # droite de D
                VISITED.append(j) # le sommet j a été exploré
                PERE.append(i) # ajoute le père du sommet j

    # construction de l'arbre couvrant
    n=len(VISITED)
    mat=np.zeros((n,n))   # matrice à n lignes et n colonnes
    for k in range(1, n): # k varie entre 1 inclus et n exclu.
        # VISITED[0] n'a pas de père. C'est le premier élément
        # du parcours
        indice_pere=PERE[k]
        indice_fils=VISITED[k]
        mat[indice_pere, indice_fils]=1
        mat[indice_fils, indice_pere]=1  # la matrice est
                      # symétrique car le graphe n'est pas orienté
    return mat
```

`VISITED[0]` n'a pas de père puisque c'est le premier élément du parcours.

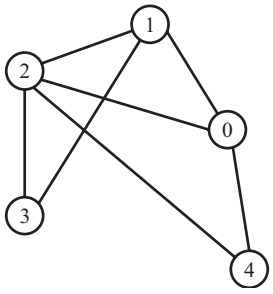
On n'utilise pas `PERE[0]`. On aurait pu écrire `PERE=[None]` au lieu de `PERE=[-1]`.

Remarque : On peut également utiliser l'algorithme du parcours en profondeur pour construire un arbre couvrant. Voir exercice13.9 « Test de connexité d'un graphe – Parcours en profondeur – Arbre couvrant ».

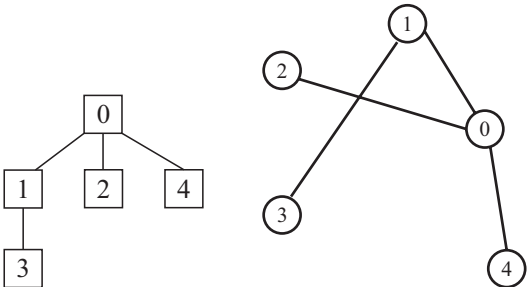
4. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
La liste `PERE` vaut : [None, 0, 0, 0, 1].
On obtient la matrice d'adjacence représentant l'arbre couvrant :

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Le graphe de départ est :



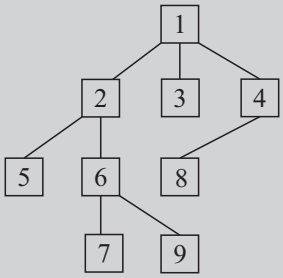
On obtient l'arbre couvrant :



On a réalisé un parcours en largeur en explorant d'abord le premier niveau : 1 – 2 – 4 puis le second niveau : 3.
On découvre les sommets niveau par niveau, d'où l'origine du nom : parcours en largeur.

Exercice 13.8 : Parcours en profondeur d'un graphe

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G=(S,A)$ représentant la structure du site web.



On considère une pile `P` et une liste `PARCOURS` qui contient la liste des sommets explorés. On pose `début=1` le sommet de départ.

Les étapes de l'algorithme de parcours en profondeur sont les suivantes :

Initialisation de l'algorithme :

La pile `P` contient le sommet de départ : `P=[début]`.

La liste `PARCOURS` contient le sommet de départ : `PARCOURS=[début]`.

Boucle tant que la pile `P` n'est pas vide :

- Si le sommet de la pile `P` (sommet *x*) possède un voisin non exploré *y* qui n'est pas dans la liste `PARCOURS`, alors on empile *y* dans `P` et on ajoute *y* dans la liste `PARCOURS`.
- Si le sommet de la pile `P` (sommet *x*) ne possède pas de voisin non exploré, alors on dépile *x* l'élément au sommet de la pile `P`.

On utilisera un flag `trouve` (de valeur `True` ou `False`) qui permet de savoir si le sommet *x* possède un voisin non exploré.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction itérative `parcoursprofondeur` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début`. La fonction retourne la liste `PARCOURS`.

3. Que retourne `parcoursprofondeur(dico, 1)` ?

Analyse du problème

L'algorithme de parcours en profondeur (ou DFS, Depth First Search, en anglais) explore une branche en profondeur depuis un sommet avant de passer à la suivante.

On va le plus profond possible pour chaque branche. On utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »).

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1], 4:[1, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4], 9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2. Comparaison entre les algorithmes de parcours en largeur et en profondeur :

- Parcours en largeur (BFS, Breadth First Search, en anglais) : voir exercice 13.5 « Parcours en largeur d'un graphe avec une deque ». On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »). **Tous les sommets non explorés adjacents** au sommet traité sont ajoutés dans la liste `VISITED`.
- Parcours en profondeur ou DFS : on utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »). On parcourt tous les sommets adjacents au sommet traité avec une boucle `while i<len(L) and trouve==False`.

  Dès qu'on trouve un sommet non exploré, le flag `trouve` passe à `True`, on ajoute ce sommet dans la liste `PARCOURS` et dans la pile `P`, on quitte la boucle `while i<len(L) and trouve==False`. On ne dépile pas ce sommet ajouté et on continue l'exploration en profondeur en repartant au début de la boucle `while len(P)!=0`.

Remarque : On empile dans `P` uniquement un sommet non exploré dans le parcours en profondeur alors qu'on ajoute dans `D` tous les sommets non explorés dans le parcours en largeur.

```
def parcoursprofondeur(dico, début):
    # fonction permettant un parcours en profondeur du
    # dictionnaire dico en partant de début et retournant
    # PARCOURS la liste des sommets visités
    P=[début]
    PARCOURS=[début]
    while len(P)!=0:
        x=P.pop()     # dépile pour récupérer l'élément au sommet
                      # de la pile
        P.append(x)   # empile x car il ne faut pas dépiler x
                      # à ce stade
        L=dico[x]     # L = liste des sommets adjacents à x
        trouve=False # on n'a pas trouvé un voisin non exploré
        i=0
        while i<len(L) and trouve==False:
            if L[i] not in PARCOURS:  # cherche dans L un voisin
                                      # non exploré
```

```
                trouve=True    # on a trouvé un voisin non exploré
                P.append(L[i]) # ajoute ce sommet en haut de
                               # la pile
                PARCOURS.append(L[i]) # marque ce voisin exploré
            i+=1
        if trouve==False:
            P.pop()  # dépile le haut de la pile
    return PARCOURS

print("Parcours en profondeur d'un graphe")
PARCOURS=parcoursprofondeur(dico, 1)
print(PARCOURS)
```

On utilise très souvent trois opérations de base avec les piles : « empiler », « dépiler » et « teste si la pile est vide ». Voir exercice 12.3 « Opérations de base sur les piles » dans le chapitre « Dictionnaire – Pile – File – Deque ». Les lignes suivantes permettent de récupérer le sommet *x* au sommet de la pile pour obtenir la liste des sommets adjacents.

```
x=P.pop()     # dépile pour récupérer l'élément en haut de la pile
P.append(x)   # empile x
L=dico[x]     # x est l'élément en haut de la pile
```

On aurait pu utiliser la ligne suivante mais on n'aurait plus été dans la logique d'utilisation des piles :

```
L=dico[P[-1]] #P [-1] est élément en haut de la pile
```

Remarque : Au début de la boucle `while len(P)!=0`, il faut récupérer le numéro du sommet en haut de la pile. Lorsqu'on utilise des piles, on ne peut utiliser que « empiler » et « dépiler » pour récupérer le numéro du sommet en haut de la pile. Il ne faut surtout pas dépiler ce sommet à cette étape du programme puisqu'il peut rester des sommets non explorés adjacents à ce sommet.

`PARCOURS.append(L[i])` permet de marquer le sommet que l'on a découvert lors de l'exploration.

3.

Initialisation de l'algorithme :

- La pile `P` contient le sommet de départ : `P` = [1].
- La liste `PARCOURS` contient le sommet de départ : `PARCOURS` = [1].

Étapes de la boucle `while` :

- Le haut de la pile `P` (sommet 1) possède un voisin non exploré : 2 car il n'est pas dans la liste `PARCOURS`. On ajoute alors 2 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2] ; `PARCOURS` = [1, 2].
- Le haut de la pile `P` (sommet 2) possède un voisin non exploré : 5 car il n'est pas dans la liste `PARCOURS`. On ajoute alors 5 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 5] ; `PARCOURS` = [1, 2, 5].

- Le haut de la pile `P` (sommet 5) n'a pas de voisin non exploré. On dépile 5 de la pile `P`. On obtient : `P` = [1, 2] ; `PARCOURS` = [1, 2, 5]. On remonte au sommet précédent et on continue l'exploration du sommet 2. On réalise un parcours en profondeur puisqu'on va le plus loin possible. On est bloqué au sommet 5. On remonte en arrière et on explore le sommet 2 en explorant en profondeur ce sommet.
- Le haut de la pile (sommet 2) possède un voisin non exploré : 6. On ajoute alors 6 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6].
- Le haut de la pile (sommet 6) possède un voisin non exploré : 7. On ajoute alors 7 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6, 7] ; `PARCOURS` = [1, 2, 5, 6, 7].
- Le haut de la pile `P` (sommet 7) n'a pas de voisin non exploré. On dépile 7 de la pile `P`. `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6, 7]. On remonte au sommet précédent et on continue l'exploration du sommet 6.
- Le haut de la pile (sommet 6) possède un voisin non exploré : 9. On ajoute alors 9 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6, 9] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- Le haut de la pile `P` (sommet 9) n'a pas de voisin non exploré. On dépile 9 de la pile `P`. On obtient : `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- On remonte au sommet précédent et on continue l'exploration du sommet 6.
- Le sommet 6 n'a plus de sommet non exploré. On remonte au sommet 2 qui n'a plus de sommet non exploré. On remonte au sommet 1. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- Le haut de la pile (sommet 1) possède un voisin non exploré : 3. On ajoute alors 3 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 3] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3].
- Le haut de la pile `P` (sommet 3) n'a pas de voisin non exploré. On dépile 3 de la pile `P`. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3]. On remonte au sommet précédent et on continue l'exploration du sommet 1.
- Le haut de la pile (sommet 1) possède un voisin non exploré : 4. On ajoute alors 4 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 4] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4].
- Le haut de la pile (sommet 4) possède un voisin non exploré : 8. On ajoute alors 8 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 4, 8] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8].
- Le haut de la pile `P` (sommet 8) n'a pas de voisin non exploré. On dépile 8 de la pile `P`. On obtient : `P` = [1, 4] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8]. On remonte au sommet précédent et on continue l'exploration du sommet 4.

- Le haut de la pile `P` (sommet 4) n'a pas de voisin non exploré. On dépile 4 de la pile `P`. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8]. On remonte au sommet précédent et on continue l'exploration du sommet 1.
- Le haut de la pile `P` (sommet 1) n'a pas de voisin non exploré. On dépile 1 de la pile `P`. On obtient : `P` = [ ] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8].

L'algorithme est terminé puisque la pile `P` est vide.

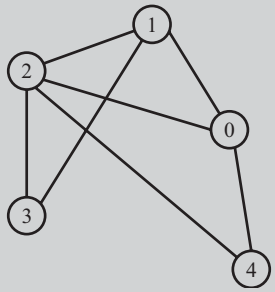
Remarques :

Dans l'algorithme du parcours en profondeur, on peut choisir n'importe quel parcours en profondeur :

- Le sommet 1 a trois sommets adjacents.
- On a commencé par explorer en profondeur 2 et 5 mais on aurait pu explorer n'importe quel autre parcours, par exemple 4 et 8.

Exercice 13.9 : Test de connexité d'un graphe – Parcours en profondeur – Arbre couvrant

On considère le graphe non orienté $G = (S, A)$:



1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe G. Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i, j] = 1$ sinon $\mathbf{M}[i, j] = 0$.

2. Écrire une fonction `testgrapheconnexe_profondeur` qui admet comme argument une matrice d'adjacence `M` et retourne `True` si le graphe est connexe et `False` sinon en utilisant le **parcours en profondeur**.

3. Écrire une fonction `arbrecouvrant_profondeur` qui admet comme arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de récupérer une matrice d'adjacence représentant l'arbre couvrant correspondant au parcours en profondeur pour un **graphe non orienté** et connexe.

On considère deux listes (`PERE`, `PARCOURS`) et une pile `P`. La liste `PARCOURS` contient la liste des sommets explorés. `PERE[k]` représente le père de `PARCOURS[k]`.

Les étapes de l'algorithme sont les suivantes :

Initialisation de l'algorithme :

- La pile `P` contient le sommet de départ.
- La liste `PARCOURS` contient le sommet de départ.
- La liste `PERE` contient −1.

Boucle tant que la pile `P` n'est pas vide :

- Si le sommet de la pile `P` (sommet *x*) possède un voisin non exploré *i* qui n'est pas dans la liste `PARCOURS`, alors on ajoute *x* dans la liste `PERE`, on empile *i* dans `P` et on ajoute *i* dans la liste `PARCOURS`.
- Si le sommet de la pile `P` (sommet *x*) ne possède pas de voisin non exploré, alors on dépile *x* l'élément au sommet de la pile `P`.

4. Représenter l'arbre couvrant du graphe *G*.

Analyse du problème

Un graphe est connexe (ou simplement connexe) si on peut relier, directement ou non, n'importe quel sommet à n'importe quel autre sommet du graphe par une chaîne. Il n'y a pas de sommet isolé.

On utilise le parcours en profondeur pour construire un arbre inclus dans le graphe *G* et qui relie tous les sommets de ce graphe (voir exercice précédent « Parcours en profondeur d'un graphe »).

Pour construire l'arbre couvrant, on définit deux listes : `PARCOURS` et `PERE`. La liste `PERE` contient la liste des pères pour chaque sommet de `PARCOURS`. Il faut en effet connaître la liste des sommets parcourus et savoir comment on a atteint ce sommet.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
import numpy as np    # bibliothèque numpy renommée np
M=np.array([[0, 1, 1, 0, 1], [1, 0, 1, 1, 0], [1, 1, 0, 1, 1],\
  [0, 1, 1, 0, 0], [1, 0, 1, 0, 0]])
```

Remarque :

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $\mathbf{M}[i, j] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

2. Pour savoir si le graphe *G* est connexe, on récupère la liste des sommets explorés avec l'algorithme de parcours en profondeur. Si tous les sommets du graphe *G* sont dans cette liste, alors *G* est connexe.

On appelle `début` le sommet de départ. On pose par exemple `début = 0`. On aurait pu prendre un autre sommet de *G*.

```
def testgrapheconnexe_profondeur(M):
    # la fonction retourne True si le graphe est connexe
    # et False sinon pour la matrice d'adjacence M
    n=np.shape(M)[0]
    début=0  # on prend un sommet quelconque, par exemple 0
    P=[début]
    PARCOURS=[début]
    while len(P)!=0:
        x=P.pop()    # dépile pour récupérer l'élément au sommet
                     # de la pile
        P.append(x)  # empile x car il ne faut pas dépiler x
                     # à ce stade
        trouve=False # on n'a pas trouvé un sommet non exploré
        i=0
        while i<n and trouve==False:
            if M[x,i]>0 and i not in PARCOURS: # cherche un sommet
                                              # non exploré
                trouve=True    # on a trouvé un sommet non exploré
                P.append(i)
                PARCOURS.append(i) # marque ce sommet exploré
            i+=1
        if trouve==False:
            P.pop()  # on dépile le haut de la pile
    if len(PARCOURS)==n:
        return True  # le graphe est connexe
    else:
        return False # le graphe n'est pas connexe
```

Dans la fonction `testgrapheconnexe_profondeur`, la liste `PARCOURS` donne la liste de tous les sommets explorés avec l'algorithme en profondeur.

`np.shape(M)[0]` retourne le nombre de lignes de la matrice `M`.

Il suffit de tester si la longueur de la liste `PARCOURS` est égale au nombre de lignes de la matrice `M` pour savoir si on a bien exploré tous les sommets du graphe.

3. L'arbre couvrant d'un graphe non orienté et connexe est un arbre inclus dans le graphe et qui relie tous les sommets du graphe *G*.

Pour construire l'arbre couvrant, il suffit de parcourir les listes `PARCOURS` et `PERE` en commençant à `PARCOURS[1]`. Le père de `PARCOURS[k]`

est `PERE[k]`. On peut ainsi remplir la matrice d'adjacence de l'arbre couvrant.

```
def arbrecouvrant_profondeur(M, début):
    # la fonction retourne l'arbre couvrant de la matrice M
    # en partant de la chaîne de caractères début
    # le graphe est non orienté et connexe
    n=np.shape(M)[0]
    P=[début]
    PARCOURS=[début]
    PERE=[-1]   # on n'utilise pas le premier élément de PERE
    while len(P)!=0:
        x=P.pop()    # dépile pour récupérer l'élément au sommet
                     # de la pile
        P.append(x)  # empile x car il ne faut pas dépiler x
                     # à ce stade
        trouve=False # on n'a pas trouvé un sommet non exploré
        i=0
        while i<n and trouve==False:
            if M[x,i]>0 and i not in PARCOURS: # cherche un sommet
                                              # non exploré
                trouve=True    # on a trouvé un sommet non exploré
                PERE.append(x)     # ajoute le père du sommet i
                P.append(i)
                PARCOURS.append(i) # marque ce sommet i exploré
            i+=1
        if trouve==False:
            P.pop()     # dépile le haut de la pile

    # construction de l'arbre couvrant
    mat=np.zeros((n,n)) # matrice à n lignes et n colonnes
    for k in range(1, len(PARCOURS)):   # k varie entre 1 inclus
                         # et len(PARCOURS) exclu.
                         # PARCOURS[0] n'a pas de père.
        indice_pere=PERE[k]
        indice_fils=PARCOURS[k]
        mat[indice_pere, indice_fils]=1
        mat[indice_fils, indice_pere]=1  # la matrice est
                    # symétrique car le graphe n'est pas orienté
    return mat
```

On n'utilise pas `PERE[0]`. On aurait pu écrire `PERE=[None]` au lieu de `PERE=[-1]`.

Remarque :

On peut également utiliser l'algorithme du parcours en largeur pour explorer tous les sommets du graphe. Voir exercice 13.7 « Test de connexité d'un graphe – Parcours en largeur – Arbre couvrant ».

On utilise très souvent les opérations de base avec les piles : empiler, dépiler et test pour savoir si la pile est vide. Voir exercice 12.3 « Opérations de base sur les

piles » dans le chapitre « Dictionnaire – Pile – File – Deque ». Les lignes suivantes permettent de récupérer le sommet x du haut de la pile.

```
x=P.pop()   # dépile pour récupérer l'élément en haut de la pile
P.append(x) # empile x
```

On aurait pu utiliser la ligne suivante mais on n'aurait plus été dans la logique d'utilisation des piles :

```
x=P[-1]     # P[-1] élément en haut de la liste P
```

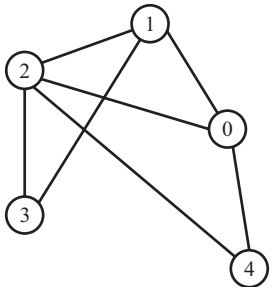
4. La liste `PARCOURS` vaut : [0, 1, 2, 3, 4].

La liste `PERE` vaut : [None, 0, 1, 2, 2].

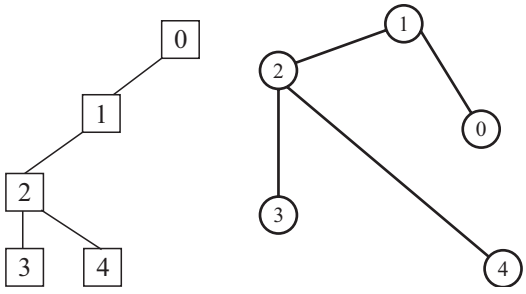
On obtient la matrice d'adjacence représentant l'arbre couvrant :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Le graphe de départ est :



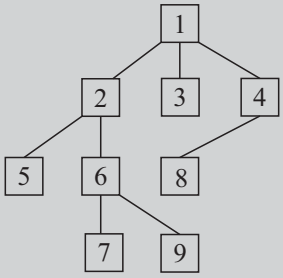
On obtient l'arbre couvrant :



On a réalisé un parcours en profondeur en explorant le plus loin possible : 0 – 1 – 2 – 3. On arrive à 3. On est bloqué. On remonte à 2 et on redescend vers 4.

Exercice 13.10 : Algorithme récursif du parcours en largeur

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



La liste `PARCOURS` contient la liste des sommets parcourus par l'algorithme. On ajoute le sommet de départ `début` dans la liste `PARCOURS` initialement vide et dans la deque `D` initialement vide.

Principe de l'algorithme récursif :

- On supprime le sommet *x* à l'extrémité gauche de `D`. On ajoute tous les sommets non explorés adjacents à *x*, dans la liste `PARCOURS` et à l'extrémité droite de `D`.
- On appelle la fonction récursive avec la deque `D` et la liste `PARCOURS`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction récursive `BFS_rec` qui admet comme arguments un dictionnaire `dico`, une deque `D` et une liste `PARCOURS`.

3. Le sommet de départ vaut 1. Que vaut `PARCOURS` après l'appel de la fonction `BFS_rec(dico, D, PARCOURS)` ?

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. L'implémentation repose sur une deque `D` dans laquelle on place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2.

- La condition d'arrêt de la fonction récursive est que la deque `D` est vide.
- On supprime le sommet *x* à l'extrémité gauche de `D` et `L=dico[x]` contient tous les sommets adjacents à *x*. La boucle `for` permet de parcourir tous les sommets adjacents à *x*. Tous les sommets adjacents non explorés sont ajoutés dans `PARCOURS` et à l'extrémité droite de `D`. Si *x* = 1, on ajoute les sommets 2, 3 et 4. On a bien un parcours en largeur.
- On appelle à nouveau la fonction récursive avec la deque `D` et la liste `PARCOURS` modifiées précédemment.

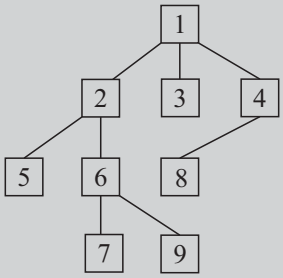
Les listes et les deques sont passées par référence et non par valeur dans les fonctions. On considère donc la même liste `PARCOURS` et la même deque `D` lors des différents appels récursifs.

```
def BFS_rec(dico, D, PARCOURS):
    # la fonction permet un parcours en largeur du dictionnaire
    # dico en partant de début et permettant d'obtenir
    # PARCOURS la liste des sommets visités
    if len(D)==0:
        return ()  # condition d'arrêt
    else:
        x=D.popleft()  # supprime le sommet à l'extrémité
                       # gauche de D
        L=dico[x]      # liste contenant les sommets adjacents à x
        for elt in L:  # parcourt les éléments de L
            if elt not in PARCOURS:
                D.append(elt)      # ajoute elt à l'extrémité
                                   # droite de D
                PARCOURS.append(elt)
        BFS_rec(dico, D, PARCOURS) # appel récursif

from collections import deque # module permettant d'utiliser
                              # les deques
D=deque()    # deque vide
début=1
D.append(début)
PARCOURS=[début]
BFS_rec(dico, D, PARCOURS)
print('Algorithme parcours en largeur récursif - Parcours :', \
      PARCOURS)
```

3. On obtient : `PARCOURS = [1, 2, 3, 4, 5, 6, 8, 7, 9]`.

Exercice 13.11 : Algorithme récursif du parcours en profondeur

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



On définit la liste `PARCOURS` qui contient la liste des sommets parcourus par l'algorithme. La liste `PARCOURS` est initialement vide.

Principe de l'algorithme récursif :

- On ajoute le sommet `s` dans la liste `PARCOURS`.
- On appelle la fonction récursive pour le premier sommet non exploré adjacent à `s`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction récursive `profondeur_rec` qui admet comme arguments un dictionnaire `dico`, un sommet de départ `s` et une liste `PARCOURS`.

3. Qu'affiche le programme suivant ?

```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

Représenter l'arbre des appels de la fonction récursive `profondeur_rec (dico, 1)`.

Analyse du problème

L'algorithme de parcours en profondeur (ou DFS, Depth First Search, en anglais) explore une branche en profondeur depuis un sommet avant de passer à la suivante. On va le plus profond possible pour chaque branche.

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2.

```
def profondeur_rec(dico, s, PARCOURS):
    # la fonction permet un parcours en profondeur du dictionnaire
    # dico en partant de s et permettant d'obtenir
    # PARCOURS la liste des sommets visités
    PARCOURS.append(s) # ajoute le sommet s dans liste PARCOURS
    L=dico[s]          # liste des sommets adjacents
    for elt in L:
        if elt not in PARCOURS:
            profondeur_rec(dico, elt, PARCOURS) # appel récursif
    # condition d'arrêt si tous les éléments de L ont été explorés
    # On n'a pas besoin de return car on ajoute
    # les sommets explorés au fur et à mesure dans PARCOURS
```

3. On considère le programme principal suivant :

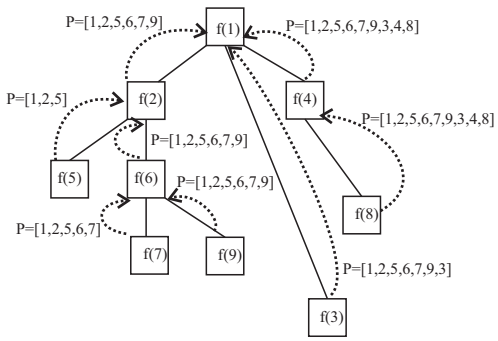
```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

Le programme Python affiche : `[1, 2, 5, 6, 7, 9, 3, 4, 8]`.

La fonction récursive est notée `f`. Pour une meilleure lisibilité, on écrit `f(1)` au lieu de `profondeur_rec(dico, 1, PARCOURS)`. La liste `PARCOURS` est notée `P`.

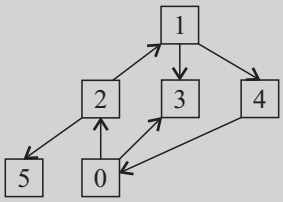
- On appelle `f(1)`. Le sommet 1 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste `L` contient les sommets adjacents à 1 : `L` = [2, 3, 4]. On considère le premier sommet de la liste. Le sommet 2 n'a pas encore été exploré. On a un appel récursif `f(2)`.
- Le sommet 2 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste L contient les sommets adjacents à 2 : `L` = [1, 5, 6]. On considère le premier sommet de la liste non exploré. Le sommet 5 n'a pas encore été exploré. On a un appel récursif `f(5)`.
- Le sommet 5 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste L contient les sommets adjacents à 5 : `L` = [2]. Aucun élément de `L` est non exploré. On est dans la condition d'arrêt de la fonction récursive. Phase de remontée avec `P` = [1, 2, 5].
- On revient au sommet 2. On explore le sommet 6 puis appel récursif pour le sommet 7. Phase de remontée avec `P` = [1, 2, 5, 6, 7].

L'arbre ci-contre représente les différents appels de la fonction `profondeur_rec` que l'on note `f`.



Exercice 13.12 : Recherche d'un circuit – Parcours en largeur

On considère le graphe orienté $G = (S, A)$:



On utilise la liste `couleur` pour mémoriser la couleur des sommets. Un sommet est blanc lorsqu'il n'a pas été traité. Lorsqu'on commence à traiter un sommet *i*, il est de couleur grise. Après avoir traité en largeur tous les successeurs (qui deviennent de couleur grise) de ce sommet *i*, le sommet *i* est de couleur noire.

On utilise une deque `D` pour gérer la file d'attente (FIFO : First In First Out). On supprime le sommet grisé à l'extrémité gauche de `D` qui devient noir. Tous les successeurs blancs de ce sommet sont ajoutés à l'extrémité droite de `D` et deviennent grisés.

1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe *G*. Si deux sommets d'origine *i* et d'extrémité $j \neq i$ sont reliés par une arête, alors $\mathbf{M}[i, j] = 1$ sinon $\mathbf{M}[i, j] = 0$.

2. Écrire une fonction `circuit_sommet` qui admet comme arguments une matrice d'adjacence `M` et un sommet de départ `début`. Cette fonction parcourt en largeur le graphe *G*. La fonction retourne `True` lorsqu'un des successeurs du sommet *x* n'est pas blanc et que ce successeur est le sommet de départ `début`. La fonction retourne `False` s'il n'existe aucun circuit passant par le sommet *s*.

3. Écrire le programme principal permettant d'afficher si le graphe orienté possède un circuit passant par le sommet `début`.

4. Écrire le programme principal permettant d'afficher si le graphe orienté possède au moins un circuit. On pourra appliquer la fonction `circuit_sommet` à chaque sommet du graphe.

Analyse du problème

On utilise le parcours en largeur du graphe.

Un chemin est simple si tous les arcs du chemin sont différents. Un circuit est un chemin simple tel que le sommet d'arrivée est le même que le sommet de départ.

Pour savoir s'il existe un circuit dans un graphe, il suffit de vérifier pour chaque sommet du graphe s'il existe un circuit passant par ce sommet.

On peut utiliser le parcours en largeur ou en profondeur pour la recherche d'un circuit.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$.

```
import numpy as np   # bibliothèque numpy renommée np
M=np.array([[0,0,1,1,0,0], [0,0,0,1,1,0], [0,1,0,0,0,1],\
            [0,0,0,0,0,0], [1,0,0,0,0,0], [0,0,0,0,0,0]])
```

Remarque :

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $\mathbf{M}[i, i] = 0$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

2.

```
def circuit_sommet(M, début):
    # la fonction permet un parcours en largeur de la matrice M
    # en partant de l'entier début
    # couleur[i]="blanc" pour un sommet i non traité
    from collections import deque # module permettant d'utiliser
                                 # les deques
    n=np.shape(M)[0]             # nombre de sommets du graphe
    couleur=["blanc" for i in range(n)]
            # les sommets non traités sont de couleur blanche
    D=deque()                    # création d'une deque vide
    D.append(début)              # ajoute début à l'extrémité
                                 # droite de D
    couleur[début]="gris"    # sommet début en cours de traitement
    while len(D)!=0:
        x=D.popleft()        # supprime le sommet x à l'extrémité
                             # gauche de D
        couleur[x]="noir"    # le sommet x a été traité
        for i in range(n):
            if M[x,i]>0 and couleur[i]=="blanc":
                D.append(i)  # ajoute le sommet i à l'extrémité
                             # droite de D
                couleur[i]="gris" # sommet à traiter
            elif M[x,i]>0 and i==début:
                             # on a trouvé un chemin entre x et i
                             # i est le sommet début
                return True  # on a trouvé un circuit
    return False             # on n'a pas trouvé de circuit
```

Remarque : On pourrait supprimer la ligne `couleur[x]="noir"` puisqu'on ne fait pas la différence entre les couleurs noire et grise.

3.

```
début=0
if circuit_sommet(M, début)==True:
    print('Le graphe orienté possède un circuit passant par'\
          ' le sommet', début)
else:
    print('Le graphe orienté ne possède pas de circuit passant par'\
          ' le sommet', début)
```

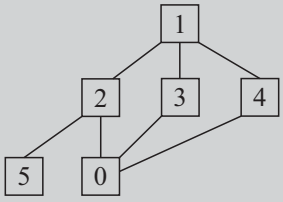
Le programme Python affiche : « Le graphe orienté possède un circuit passant par le sommet 0 ».

4. On applique la fonction `circuit_sommet` à chaque sommet du graphe. Si on trouve un circuit passant par un sommet, alors le graphe possède au moins un circuit.

```
def rec_circuit_largeur(M):
    # la fonction retourne False si le graphe non orienté
    # (pour la matrice M) ne possède pas de circuit
    n=np.shape(M)[0]      # nombre de sommets du graphe
    for i in range(n):    # i varie entre 0 inclus et n exclu
        if circuit_sommet(M, i)==True:
            return True   # quitte la fonction si un circuit
                          # est trouvé
    return False

if rec_circuit_largeur(M)==True:
    print('Le graphe orienté possède au moins un circuit.')
else:
    print('Le graphe non orienté ne possède pas de circuit.')
```

Exercice 13.13 : Recherche d'un cycle – Parcours en largeur

On considère le graphe connexe et non orienté $G = (S, A)$:



On utilise la liste `couleur` pour mémoriser la couleur des sommets. Un sommet est blanc lorsqu'il n'a pas été traité. Lorsqu'on commence à traiter un sommet *i*, il est de couleur grise. Après avoir traité en largeur tous les sommets adjacents au sommet *i*, le sommet *i* est de couleur noire.

On utilise la liste `PERE` : `PERE[i]` désigne le père du sommet *i* lors du parcours du graphe en largeur.

On utilise une deque `D` pour gérer la file d'attente (FIFO : First In First Out). On supprime le sommet grisé à l'extrémité gauche de `D` qui devient noir. Tous les sommets adjacents à ce sommet sont ajoutés à l'extrémité droite de `D` et deviennent grisés.

1. Construire la matrice d'adjacence $\mathbf{M}_{i,\,j}$ du graphe *G*. Si deux sommets d'origine *i* et d'extrémité $j \neq i$ sont reliés par une arête, alors $\mathbf{M}[i, j] = 1$ sinon $\mathbf{M}[i, j] = 0$.

2. Écrire une fonction `cycle_som` qui admet comme arguments une matrice d'adjacence `M` et un sommet `début`. Cette fonction parcourt en largeur le graphe *G*. La fonction retourne `True` lorsqu'un sommet *i* adjacent à un sommet *x* n'est pas blanc et que le père de *x* n'est pas *i*. La fonction retourne `False` sinon.

3. Écrire le programme principal permettant d'afficher si un graphe connexe et non orienté possède au moins un cycle. On pourra appliquer la fonction `cycle_som` à un sommet quelconque du graphe.

4. Écrire le programme principal permettant d'afficher si un graphe non orienté possède au moins un cycle. On pourra appliquer la fonction `cycle_som` à chaque sommet du graphe.

Analyse du problème

On utilise le parcours en largeur du graphe.

Une chaîne est simple si toutes les arêtes de la chaîne sont différentes. Un cycle est une chaîne simple telle que le sommet d'arrivée est le même que le sommet de départ.

L'énoncé n'impose pas de trouver un cycle passant par le sommet de départ.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$.

```
import numpy as np   # bibliothèque numpy renommée np
M=np.array([[0,0,1,1,0,0], [0,0,0,1,1,0], [0,1,0,0,0,1],\
            [0,0,0,0,0,0], [1,0,0,0,0,0], [0,0,0,0,0,0]])
```

Remarque :

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $\mathbf{M}[i, j] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i, j] = \mathbf{M}[j, i]$.

2. Il y a des différences dans la recherche de circuit (pour un graphe orienté) et de cycle (pour un graphe non orienté) :

- Pour un graphe orienté (voir exercice précédent « Recherche d'un circuit – Parcours en largeur »), il suffit de tester que le successeur de *x* est le sommet de départ. Par exemple : 1⟶2⟶1 peut représenter un circuit. La condition pour trouver un circuit est : `if M[x,i]>0 and i==début`.
  Un chemin doit exister entre *x* et *i*. Il faut également que *i* soit le sommet de départ `début`. Dans cet exercice, on impose de trouver un circuit passant par `début`.
- Pour un graphe non orienté, 1–2–1 ne représente pas un cycle.
  La condition pour trouver un cycle est : `if M[x,i]>0 and couleur [i]!="blanc" and PERE[x]!=i`.
  Une chaîne doit exister entre *x* et *i*. Le sommet *i* doit déjà être visité et le père de *x* ne doit pas être i (on évite ainsi de considérer *i* – *x* – *i* comme un cycle). Dans cet exercice, on n'impose pas de trouver un cycle passant par le sommet de départ `début`.

```
def cycle_som(M, début):
    # la fonction retourne False si la matrice M ne possède pas
    # de cycle en partant de l'entier début
    from collections import deque  # module permettant d'utiliser
                                   # les deques
    n=np.shape(M)[0]               # nombre de sommets du graphe
    PERE=[-1 for i in range(n)]
    couleur=["blanc" for i in range(n)]
            # les sommets non traités sont de couleur blanche
    D=deque()              # création d'une deque vide
    D.append(début)        # ajoute début à l'extrémité droite de D
    couleur[début]="gris" # sommet début en cours de traitement
    while len(D)!=0:
        x=D.popleft()      # supprime le sommet x à l'extrémité
                           # gauche de D
        couleur[x]="noir" # le sommet x a été traité
        for i in range(n):
            if M[x,i]>0 and couleur[i]!="blanc" and PERE[x]!=i:
                    # on a trouvé une chaîne de x vers i
                    # i a déjà été visité
                    # on teste que le père de x n'est pas i
                return True
            elif M[x,i]>0 and couleur[i]=="blanc":
                D.append(i)      # ajoute le sommet i à l'extrémité
                                 # droite de D
                PERE[i]=x        # le père de i est x
                couleur[i]="gris" # sommet à traiter
    return False    # pas de cycle en partant de début
```

Remarque : Pour les graphes orientés, on parle de circuit au lieu de cycle.

3. On considère un sommet quelconque pour le graphe connexe et non orienté.

```
début=0
if cycle_som(M, début)==True:
    print('Le graphe connexe et non orienté possède au moins'\
          ' un cycle.')
else:
    print('Le graphe connexe et non orienté ne possède pas de'\
          ' cycle en partant du sommet ',début)
```

4. On applique la fonction `cycle_som` à chaque sommet du graphe.

```
def rec_cycle_larg(M):
    # la fonction retourne False si la matrice M ne possède
    # pas de cycle
    n=np.shape(M)[0]     # nombre de sommets du graphe
    for i in range(n):  # i varie entre 0 inclus et n exclu
        if cycle_som(M, i)==True:
            return True # quitte la fonction si un cycle trouvé
    return False

if rec_cycle_larg(M)==True:
    print('Le graphe possède au moins un cycle.')
else:
    print('Le graphe ne possède pas de cycle.')
```

Partie 10

Recherche du plus court chemin

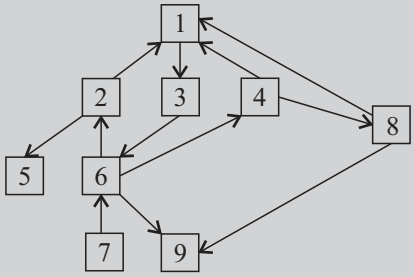
Plan

Recherche du plus court chemin

14

Exercice 14.1 : Recherche du plus court chemin – Graphe orienté

On considère le graphe orienté $G = (S, A)$:



`L.reverse()` permet d'inverser les éléments de la liste `L`.

On considère dans l'algorithme une liste `PARCOURS` et un dictionnaire `PERE` :

- La liste `PARCOURS` contient la liste des sommets du plus court chemin du sommet de départ `début` jusqu'au sommet d'arrivée `fin` différent de `début`.
- `PERE[i]` représente le père du sommet *i* lors du parcours en largeur du graphe depuis le sommet de départ `début`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des successeurs.

2. Écrire une fonction itérative `BFS` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début`. La fonction retourne un dictionnaire `PERE` en utilisant l'algorithme de parcours en largeur.

3. La liste `PARCOURS` est initialement vide.

Principe de l'algorithme du plus court chemin :

Pour obtenir les sommets du plus court chemin de `début` jusqu'à `fin`, il faut remonter dans l'arborescence du dictionnaire `PERE` depuis le sommet `fin` jusqu'à la racine `début`.

Écrire une fonction récursive `pluscourtchemin` qui admet comme arguments un sommet de départ `début`, un sommet d'arrivée `fin`, un dictionnaire `PERE` et une liste `PARCOURS` permettant d'obtenir le plus court chemin de `début` jusqu'à `fin`.

4. Écrire le programme principal permettant d'afficher le plus court chemin entre le sommet de départ `début` et le sommet d'arrivée `fin`. Qu'obtient-on pour `début = 1` et `fin = 8` ?

Analyse du problème

Le plus court chemin de `début` jusqu'à `fin` est le chemin comportant le moins d'arcs. On utilise l'algorithme de parcours en largeur (BFS, Breadth First Search, en anglais) permettant de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. Voir exercice 13.5 « Parcours en largeur d'un graphe avec une deque » dans le chapitre « Graphes ».

1.

```
dico={1:[3], 2:[1, 5], 3:[6],\
      4:[1, 8], 5:[], 6:[2, 4, 9],\
      7:[6], 8:[1, 9], 9:[]}
```

Le sommet 2 a deux successeurs : 1 et 5. La valeur de la clé 2 est la liste des successeurs [1, 5]. Le sommet 6 n'est pas le successeur du sommet 4.

2. Les sommets déjà visités sont marqués pour éviter d'explorer plusieurs fois un même sommet. La liste `VISITED` contient les sommets visités.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque `D` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED = [début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *x* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets non explorés adjacents au sommet *x*.

```
def BFS(dico, début):
    # la fonction renvoie le dictionnaire PERE avec un parcours
    # en largeur pour le dictionnaire dico
    from collections import deque # module permettant d'utiliser
                                  # les deques
    D=deque()          # deque vide
    D.append(début)    # ajoute le sommet de départ
    PERE={ }           # dictionnaire vide
    VISITED=[début]
    while len(D)!=0:
        x=D.popleft() # supprime le sommet x à l'extrémité
                      # gauche de D
        L=dico[x]     # liste contenant les sommets adjacents à x
        for elt in L:     # parcourt les éléments de L
            if elt not in VISITED: # teste si le sommet n'a pas
                                   # déjà été exploré
                D.append(elt) # ajoute le sommet elt à l'extrémité
                              # droite de D
                VISITED.append(elt) # le sommet elt a été exploré
                PERE[elt]=x # ajoute clé, valeur dans le dico PERE
    return PERE
```

La ligne `PERE[elt]=x` permet d'ajouter `elt` (clé) et `x` (valeur de la clé) dans le dictionnaire `dico`.

3.

```
def pluscourtchemin(début, fin, PERE, PARCOURS):
    # la fonction permet d'avoir le plus court chemin
    # de début (int) à fin (int) dans la liste PARCOURS
    # à partir du dictionnaire PERE
    if début==fin:
        PARCOURS.append(fin)
        return()   # condition d'arrêt
    elif PERE[fin]=='':
        PARCOURS=[]
        return ()  # condition d'arrêt
    else:
        PARCOURS.append(fin)  # on ajoute le sommet fin dans
                              # la liste PARCOURS
        pluscourtchemin(début, PERE[fin], PERE, PARCOURS)
                              # appel récursif
```

4.

```
début, fin=1, 8
PERE=BFS(dico, début)
if fin not in PERE:     # teste si le sommet fin est dans
                        # le dictionnaire PERE
    print("Il n'y a pas de chemin entre", début, "et", fin,".")
else:
    PARCOURS=[]
    pluscourtchemin(début, fin, PERE, PARCOURS)
    PARCOURS.reverse()  # inverse les éléments de la liste
    print('Plus court chemin :', PARCOURS)
```

Si le sommet `fin` n'est pas dans le dictionnaire `PERE`, alors il n'y a pas de plus court chemin entre `début` et `fin`.

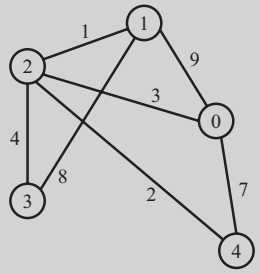
Le programme Python affiche :

```
Parcours : [1, 3, 6, 4, 8]
```

Remarque : Dans le cas d'un graphe non orienté, on parle de chaîne au lieu de chemin.

Exercice 14.2 : Algorithme de Dijkstra

On considère le graphe non orienté $G=(S,A)$, où le nombre situé sur l'arête joignant deux villes (ou sommets) est leur distance :



`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Construire la matrice d'adjacence $(\mathbf{M}_{i,j})_{0\leq i,j\leq n-1}$ (appelée également **matrice de distances)** du graphe *G*, définie par :

Pour tous les indices *i, j*, $\mathbf{M}_{i,j}$ représente la distance entre les villes d'origine *i* et d'extrémité *j*.

Lorsque les villes ne sont pas reliées, cette distance vaut l'infini. On définit la variable `inf=float("inf")` qui représente une distance infinie.

2. On cherche à déterminer le plus court chemin pour aller d'une ville de départ notée `départ` à une ville d'arrivée notée `arrivée` en utilisant l'algorithme de Dijkstra :

On définit la liste `VILLES` contenant les informations suivantes pour chaque ville : [ville précédente sur le chemin, distance parcourue depuis la ville de départ, ville sélectionnée ou non (booléen vrai ou faux)].

a) Initialisation de l'algorithme :

Toutes les villes sont non sélectionnées sauf la ville de départ. Pour cette ville, la distance parcourue vaut 0.

Pour les villes non sélectionnées, les distances parcourues depuis la ville de départ sont initialisées à l'infini.

On définit une variable `position` qui correspond à la ville pour laquelle l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée`, répéter les opérations suivantes pour toutes les villes *i* non sélectionnées :

- Calculer la variable `somme` = distance entre la ville `départ` et la ville `position` + distance entre la ville `position` et la ville *i*.
- Si la variable `somme` est inférieure à la distance entre la ville `départ` et la ville *i*, alors remplacer cette distance par `somme`. La ville précédente sur le chemin est alors `position`.

Chercher la ville (parmi les villes non sélectionnées) pour laquelle la distance entre celle-ci et la ville `départ` est la plus petite. Cette ville définit alors la nouvelle valeur de la variable `position` et cette ville devient sélectionnée.

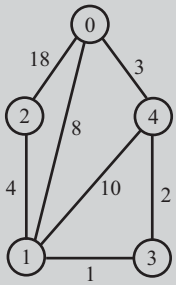
Écrire une fonction `dijkstra` qui admet comme arguments d'entrée la matrice d'adjacence `M`, la ville `départ` et la ville `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme de Dijkstra.

Écrire le programme principal permettant de déterminer le plus court chemin entre la ville de départ 3 et la ville d'arrivée 0. Le programme affichera le chemin suivi ainsi que la distance parcourue.

3. Pourquoi cet algorithme est appelé algorithme glouton ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

5. On considère le graphe suivant :



Écrire le programme permettant de déterminer le plus court chemin entre la ville de départ 0 et la ville d'arrivée 2. Le programme affichera le chemin suivi ainsi que la distance parcourue.

Analyse du problème

On étudie l'algorithme de Dijkstra, qui un algorithme de plus court chemin. On définit la matrice d'adjacence qui contient l'ensemble des distances entres les villes. La longueur d'une chaîne est la somme des poids des arêtes qui le constituent. Dans le cas d'un graphe non orienté, on parle de chaîne au lieu de chemin. On utilise ici le terme chemin par abus de langage.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 9 & 3 & \infty & 7 \\ 9 & 0 & 1 & 8 & \infty \\ 3 & 1 & 0 & 4 & 2 \\ \infty & 8 & 4 & 0 & \infty \\ 7 & \infty & 2 & \infty & 0 \end{pmatrix}$.

Remarque : On peut se reporter à l'exercice 13.1 « Matrice d'adjacence » du chapitre « Graphes » pour avoir plus d'explications sur la construction de cette matrice.

2.

- Dans l'algorithme de Dijkstra, on calcule la distance de `départ` à toutes les villes *i* non sélectionnés en passant par la ville `position`. On cherche ensuite le minimum des distances pour les villes non sélectionnées. Ce minimum permet de déclarer une ville sélectionnée. Cette distance est définitive.

  On définit un sous-graphe *G'*. Dans l'état initial, *G'* contient uniquement la ville `départ`. À chaque étape de la boucle `while(position!=arrivée)`, on ajoute la ville `position` dans *G'*. Toutes les distances des villes de *G'* sont définitives.

- Dans l'algorithme de Bellman-Ford (qui sera traité dans l'exercice « Algorithme de Bellman-Ford » du livre de deuxième année), on peut trouver ultérieurement une distance encore plus petite en passant par d'autres sommets avec des arêtes de poids négatif. Il faut recalculer toutes les distances à l'étape suivante.

On définit une liste `VILLES` qui contient pour chaque ville *i* :

- `VILLES[i][0]` : ville précédente.
- `VILLES[i][1]` : $g(i)$ = distance entre la ville `départ` et la ville *i*.
- `VILLES[i][2]` : `True` si la ville *i* est sélectionnée, sinon `False`.

a) Initialisation de l'algorithme :

Toutes les villes sont non sélectionnées (`VILLES[i][2] = False`) sauf la ville de départ. Pour cette ville, la distance parcourue vaut 0. Pour les villes non sélectionnées, les distances parcourues depuis la ville de départ sont initialisées à l'infini : `VILLES[i][1] = inf`. La variable `position` prend initialement la valeur `départ`.

b) On définit une boucle `while (position!=arrivée)` tant que la variable `position` (sommet appartenant à la frontière entre les villes sélectionnées et les villes non sélectionnées) n'est pas égale à la variable `arrivée`.

- La boucle `for i in range(n)` avec le test `if VILLES [i][2]==False` permet de parcourir toutes les villes non sélectionnées. On calcule `somme` = `VILLES[position][1]` + `M[position, i]` (= distance entre `départ` et `position` + distance entre `position` et *i*). Si la nouvelle distance `somme` est inférieure à `VILLES[i][1]`, alors `VILLES[i][1] = somme` et la ville `position` est la ville précédente de *i* : `VILLES[i][0]` = `position`.
- On cherche la ville `indice` (parmi les villes non sélectionnées) pour laquelle la distance `VILLES[indice][1]` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et cette ville devient sélectionnée.
- Si `indice` = `position`, on ne peut pas atteindre la ville d'arrivée.

```
import numpy as np  # bibliothèque numpy renommée np
inf=float("inf")

def init(départ, nb_villes):
    # la fonction initialise la liste VILLES
    # à partir de départ (int)
    # nb_villes est le nombre de villes dans le graphe
    VILLES=[]   # initialisation de la liste VILLES
    for i in range(nb_villes):
        if i==départ:
            VILLES.append([-1, 0, True])
           # la valeur -1 n'est pas utilisée car ville de départ
           # True : uniquement la ville de départ est sélectionnée
```

```
        else:
            VILLES.append([-1, inf, False])
           # la valeur -1 n'est pas utilisée car distance infinie
           # False : ville non sélectionnée
return VILLES

def dijkstra(M, départ, arrivée):
    # la fonction permet d'avoir le plus court chemin
    # de départ (int) à arrivée (int) dans la liste L à partir
    # de la matrice d'adjacence M. On récupère la liste VILLES
    nb_villes=np.shape(M)[0] # nb de villes = nb de lignes de M
    VILLES=init(départ, nb_villes) # initialisation de la
                                   # liste VILLES
    # l'algorithme est appliqué pour la ville position
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(nb_villes):   # i décrit toutes les villes
            if VILLES[i][2]==False : # ville non sélectionnée
                somme=VILLES[position][1]+M[position, i]
                if somme<VILLES[i][1]:
                    VILLES[i][1]=somme
                       # nouvelle valeur de la distance à
                       # la ville de départ
                    VILLES[i][0]=position # nouvelle ville
                                          # précédente sur le chemin
        # recherche du minimum des distances pour les villes
        # non sélectionnées
        val_min=inf
        for i in range(nb_villes):
            if VILLES[i][2]==False and VILLES[i][1]<val_min:
                indice=i
                val_min=VILLES[i][1]
        if indice==position:
            return [],inf           # on n'atteint pas arrivée
        else:
            VILLES[indice][2]=True  # cette ville est sélectionnée
            position=indice  # nouvelle valeur de la variable
                             # position

    # liste des villes parcourues
    i=arrivée
    L=[arrivée]  # L = liste des villes parcourues en sens inverse
    while (i!=départ):
        i=VILLES[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des villes parcourues dans le sens direct
    return L, VILLES[arrivée][1]

# initialisation du programme
M=np.array([[0,9,3,inf,7],[9,0,1,8,inf],[3,1,0,4,2],\
            [inf,8,4,0,inf],[7,inf,2,inf,0]])
```

```
départ, arrivée=3, 0  # ville de départ et ville d'arrivée
L, dist=dijkstra(M, départ, arrivée)
print("Algorithme de Dijkstra - Chemin suivi : ", L) # liste des
                                                     # villes
print ("Distance parcourue = ", dist)
```

Le programme Python affiche :

```
Chemin suivi : [3, 2, 0]
Distance parcourue = 7.0
```

Remarque :

On considère l'exemple suivant pour expliquer l'algorithme de Dijkstra.

Étape d'initialisation

- Ville de départ = `depart` = 3 et ville d'arrivée = `arrivee` = 0.
- On définit une liste `VILLES` contenant les informations suivantes pour chaque ville : ville précédente sur le chemin, distance parcourue depuis la ville de départ, ville sélectionnée ou non (booléen `True` ou `False`).
- On a alors : `VILLES`=[[−1,inf,False],[−1,inf,False],[−1,inf,False],[−1,0,True],[−1,inf,False]]. Les valeurs −1 ne sont pas significatives puisque la distance est infinie ou la ville est `départ`.

Algorithme

- On définit la variable `position`, qui est la ville atteinte avec le plus court chemin depuis la ville de départ. Pour toutes les villes *i* non sélectionnées et différentes de `position`, on compare `somme` (distance entre ville de départ et position + distance entre `position` et `i = VILLES[position][1] + M[position, i]`) et la distance depuis la ville de départ (`VILLES[i][1]`).
- Chercher pour toutes les villes *X* précédentes celle où la variable « distance depuis la ville de départ » est minimale. Cette ville définit alors la nouvelle valeur de la variable `position` et la variable « ville sélectionnée » passe à `True`.

1re itération : position = 3

- On parcourt les villes *X* non sélectionnées : 0, 1, 2 et 4.

  On obtient alors : `VILLES`=[[−1,inf,False],[3,8,False],[3,4,False],[−1,0,True],[−1,inf,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 2.

  On obtient alors : `position = 2` et `VILLES`=[[−1,inf,False],[3,8,False],[3,4,True],[−1,0,True],[−1,inf,False]].

2e itération : position = 2

- On parcourt les villes *X* non sélectionnées : 0, 1, 4.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,False],[3,4,True],[−1,0,True],[2,6,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 1.

  On obtient alors : `position` = 1 et `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,False]].

3e itération : position = 1

- On parcourt les villes *X* non sélectionnées : 0, 4.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 4.

  On obtient alors : `position` = 4 et `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,True]].

4e itération : position = 4

- On parcourt les villes *X* non sélectionnées : 0.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,True]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimal. C'est la ville 0.

  On obtient alors : `position` = 0 et `VILLES` = [[2,7,True], [2,5,True], [3,4,True], [−1,0,True], [2,6,True]].

Pour obtenir le trajet, on part de la ville d'arrivée, la distance minimale entre la ville de départ et la ville d'arrivée est obtenue par `VILLES[arrivée][1]` = 7. Pour obtenir le trajet, on obtient la ville précédente avec `VILLES[arrivée][0]` = 2 et, de proche en proche, on remonte à la ville de départ.

3. L'algorithme glouton (greedy en anglais) repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu à la ville `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des villes non sélectionnées, la ville suivante `indice` dont la distance entre `départ` et `indice` est la plus petite. Ce choix n'est pas remis en cause ultérieurement.

4. Soit *n* le nombre de villes.

Initialisation de la liste `VILLES` : 1 opération et, pour chaque valeur de *i* : 1 comparaison et 1 affectation. La boucle `for` est parcourue *n* fois. On a donc 2*n*+1 opérations élémentaires.

Dans le pire des cas, on a (n–1) itérations dans la boucle `while`. Pour chaque valeur de `position`, on a :

- 1 comparaison, 1 calcul de somme, 1 test et deux affectations pour chaque valeur de i : 5n opérations élémentaires.
- 1 affectation : 1 opération élémentaire.
- Recherche du minimum : trois comparaisons et deux affectations pour chaque valeur de i : 5n opérations élémentaires.
- 2 affectations : 2 opérations élémentaires.

On a donc $(2n+1)+(n-1)\times(5n+1+5n+2)=2n+1+(n-1)(10n+3)$ opérations élémentaires.

La complexité dans le pire des cas est quadratique en $O(n^2)$.

Remarque : On peut accepter des petites différences dans l'évaluation du nombre total d'opérations élémentaires. La complexité de l'algorithme ne sera pas modifiée.

5. La matrice d'adjacence est : $\mathbf{M}_2 = \begin{pmatrix} 0 & 8 & 18 & \infty & 3 \\ 8 & 0 & 4 & 1 & 10 \\ 18 & 4 & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & 2 \\ 3 & 10 & \infty & 2 & 0 \end{pmatrix}$.

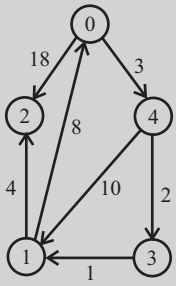
```
M2=np.array([[0,8,18,inf,3],[8,0,4,1,10],[18,4,0,inf,inf],\
             [inf,1,inf,0,2],[3,10,inf,2,0]])
départ2, arrivée2=0, 2       # ville de départ et ville d'arrivée
L2, dist2=dijkstra(M2, départ2, arrivée2)
print("Trajet suivi : ", L2) # affichage de la liste des villes
print ("Distance parcourue = ", dist2)
```

Le programme Python affiche :

```
Trajet suivi : [0, 4, 3, 1, 2]
Distance parcourue = 10.0
```

Exercice 14.3 : Algorithme de Dijkstra avec des arcs

On considère le graphe orienté $G=(S,A)$ qui représente le réseau routier d'un département en prenant en compte le sens de la circulation. Une route à sens unique est représentée par un arc dont le poids est la distance en kilomètres entre deux villes (ou sommets).



`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Construire la matrice d'adjacence $(\mathbf{M}_{i,j})_{0\leq i,j\leq n-1}$ (appelée également **matrice de distances)** du graphe *G*, définie par :

Pour tous les indices *i, j*, $\mathbf{M}_{i,j}$ représente la distance entre les sommets d'origine *i* et d'extrémité *j*.

Lorsque les sommets ne sont pas reliés, cette distance vaut l'infini. On définit la variable `inf=float("inf")` qui représente une distance infinie.

2. Écrire le programme Python permettant de déterminer le plus court chemin entre la ville de départ 0 et la ville d'arrivée 2 en utilisant l'algorithme de Dijkstra. Le programme affichera le chemin suivi ainsi que la distance parcourue.

Analyse du problème

On utilise l'algorithme de Dijkstra, qui un algorithme de plus court chemin. On définit la matrice d'adjacence qui contient l'ensemble des distances entres les villes. On considère un graphe orienté dont le poids des arcs est un réel positif.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 10 & \infty & 2 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

2.

```
import numpy as np   # bibliothèque numpy renommée np
inf=float("inf")

def init2(départ, nb_villes):
    # la fonction initialise la liste VILLES à partir
    # de départ (int)
    # nb_villes est le nombre de villes dans le graphe
    VILLES=[]   # initialisation de la liste VILLES
```

```
    for i in range(nb_villes):
        if i==départ:
            VILLES.append([-1, 0, True])
           # la valeur -1 n'est pas utilisée car ville de départ
           # True : uniquement la ville de départ est sélectionnée
        else:
            VILLES.append([-1, inf, False])
           # la valeur -1 n'est pas utilisée car distance infinie
           # False : ville non sélectionnée
    return VILLES

def dijkstra2(M, départ, arrivée):
    # la fonction permet d'avoir le plus court chemin
    # de départ (int) à arrivée (int) dans la liste L à partir
    # de la matrice d'adjacence M. On récupère la liste VILLES
    nb_villes=np.shape(M)[0] # nb de villes = nb de lignes de M
    VILLES=init2(départ, nb_villes) # initialisation de la liste
                                    # VILLES
    # l'algorithme est appliqué pour la ville position
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(nb_villes):   # i décrit toutes les villes
            if VILLES[i][2]==False : # ville non sélectionnée
                somme=VILLES[position][1]+M[position, i]
                if somme<VILLES[i][1]:
                    VILLES[i][1]=somme
                       # nouvelle valeur de la distance à
                       # la ville de départ
                    VILLES[i][0]=position  # nouvelle ville
                                           # précédente sur le chemin
        # recherche du minimum des distances pour les villes
        # non sélectionnées
        val_min=inf
        for i in range(nb_villes):
            if VILLES[i][2]==False and VILLES[i][1]<val_min:
                indice=i
                val_min=VILLES[i][1]
        if indice==position:
            return [],inf           # on n'atteint pas arrivée
        else:
            VILLES[indice][2]=True  # cette ville est sélectionnée
            position=indice  # nouvelle valeur de la variable
                             # position

    # liste des villes parcourues
    i=arrivée
    L=[arrivée]  # L = liste des villes parcourues en sens inverse
    while (i!=départ):
        i=VILLES[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des villes parcourues dans le sens direct
    return L, VILLES[arrivée][1]
```

```
# initialisation du programme
M=np.array([[0,9,3,inf,7],[9,0,1,8,inf],[3,1,0,4,2],\
            [inf,8,4,0,inf],[7,inf,2,inf,0]])
départ, arrivée=3, 0  # ville de départ et ville d'arrivée

L, dist=dijkstra2(M, départ, arrivée)
print("TRAJET NUMERO 1")
print("Chemin suivi : ", L) # affichage de la liste des villes
print ("Distance parcourue = ", dist)
```

Le programme Python affiche :

```
Chemin suivi : [0, 4, 3, 1, 2]
Distance parcourue = 10.0
```

Remarque :

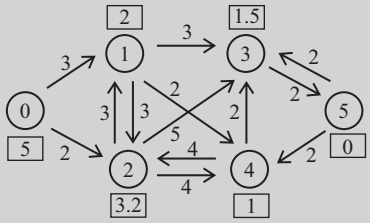
Il faut bien écrire `M[position, i]`, en respectant cet ordre des arguments.

L'algorithme de Dijkstra ne fonctionne pas correctement pour les graphes orientés contenant des arêtes de poids négatif : il déclare trop tôt la distance vers un sommet comme définitive.

On étudiera en deuxième année des algorithmes (Bellman-Ford et Floyd-Warshall) permettant de traiter des arcs dont le poids est négatif.

Exercice 14.4 : Algorithme A*

On considère le graphe orienté $G = (S, A)$ qui représente le réseau routier d'un département en prenant en compte le sens de la circulation. Une route à sens unique est représentée par un arc dont le poids est la distance en kilomètres entre deux sommets (ou deux villes). Les distances indiquées dans les rectangles sur le graphe ci-dessous représentent les distances à vol d'oiseau entre les sommets et le sommet d'arrivée 5.



Une heuristique est un algorithme qui calcule rapidement une solution pouvant être approximative. On utilise la distance euclidienne (ou distance à vol d'oiseau) pour estimer le coût restant $h(i)$ permettant d'atteindre le sommet `arrivée` à partir du sommet *i*.

`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Construire la matrice d'adjacence $(\mathbf{M}_{i,j})_{0\leq i,j\leq n-1}$ (appelée également **matrice de distances**) du graphe *G*, définie par :

Pour tous les indices *i, j*, $\mathbf{M}_{i,j}$ représente la distance entre les sommets d'origine *i* et d'extrémité *j*.

Lorsque les sommets ne sont pas reliés, cette distance vaut l'infini. On définit la variable `inf=float("inf")` qui représente une distance infinie.

2. L'algorithme A* est une variante de l'algorithme de Dijkstra. On dispose pour chaque sommet *i* d'une estimation du coût restant pour atteindre le sommet `arrivée` à partir du sommet *i* : $h(i)$ = distance euclidienne (ou distance à vol d'oiseau) entre le sommet *i* et le sommet `arrivée`. On définit la fonction d'évaluation *f* telle que : $f(i) = g(i) + h(i)$.

- $g(i)$ est le coût réel du chemin optimal entre le sommet `départ` et le sommet *i* dans la partie déjà explorée ;
- $h(i)$ est le coût estimé du chemin qui reste à parcourir entre *i* et `arrivée`.

On définit la liste `SOMMETS` contenant les informations suivantes pour chaque sommet :

[sommet précédent sur le chemin, distance parcourue depuis le sommet de départ, sommet sélectionné ou non (booléen vrai ou faux), distance évaluée entre `départ` et `arrivée`]

a) Initialisation de l'algorithme A* :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance *f* (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

On définit la liste `H=[5, 2, 3.2, 1.5, 1, 0]` telle que `H[i]` = distance euclidienne entre le sommet *i* et le sommet `arrivée` = 5.

Écrire une fonction `algoA` qui admet comme arguments d'entrée la matrice d'adjacence `M`, la liste `H`, le sommet `départ` et le sommet `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme A*.

3. Écrire le programme principal permettant de déterminer le plus court chemin entre le sommet de départ 0 et le sommet d'arrivée 5. Le programme affichera le chemin suivi ainsi que la distance parcourue.

Analyse du problème

La fonction *h* est une fonction heuristique telle que $h(i)$ est le coût estimé du chemin qui reste à parcourir entre le sommet *i* et le sommet `arrivée`. On utilise la distance à vol d'oiseau dans ce problème alors que, dans l'exercice suivant « Variantes de l'algorithme A* – Distance de Manhattan », on utilise la distance de Manhattan. L'idée est de choisir un sommet qui semble être le plus prêt du sommet d'arrivée.

La fonction d'évaluation permet de déterminer quel sommet est sélectionné en premier, c'est-à-dire retiré de la frontière entre les sommets sélectionnés et les sommets non sélectionnés : $f(i) = g(i) + h(i)$ renvoie une estimation de la distance entre le sommet de départ et le sommet d'arrivée en passant par le sommet *i* du graphe. On considère un graphe orienté dont le poids des arcs est un réel positif.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & 3 & 3 & 2 & \infty \\ \infty & 3 & 0 & 5 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & 4 & 2 & 0 & \infty \\ \infty & \infty & \infty & 2 & 2 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

2. On définit une liste `SOMMETS` qui contient pour chaque sommet *i* :

- `SOMMETS[i][0]` : sommet précédent ;
- `SOMMETS[i][1]` : $g(i)$ = distance entre le sommet `départ` et le sommet *i* ;
- `SOMMETS[i][2]` : `True` si le sommet *i* est sélectionné, sinon `False` ;
- `SOMMETS[i][3]` : $f(i)$ = distance entre le sommet `départ` et le sommet `arrivée`.

La méthode gloutonne repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu au sommet `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des sommets non sélectionnés, le sommet suivant `indice` dont la distance *f*(`indice`) entre `départ` et `arrivée` est la plus petite. Ce choix n'est pas remis en cause ultérieurement. On privilégie les sommets « qui semblent » nous rapprocher de la destination.

a) Initialisation de l'algorithme :

Tous les sommets sont non sélectionnés (`SOMMETS[i][2]=False`) sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini : `SOMMETS[i][1]=inf` et `SOMMETS[i][3]=inf`. La variable `position` prend initialement la valeur `départ`.

b) On définit une boucle `while(position!=arrivée)` : tant que la variable `position` (sommet appartenant à la frontière entre les sommets sélectionnés et les sommets non sélectionnés) n'est pas égale à la variable `arrivée`.

- La boucle `for i in range(n)` avec le test `if SOMMETS[i][2]==False` permet de parcourir tous les sommets non sélectionnés. On calcule `g_i=SOMMETS[position][1]+M[position, i]` (= distance entre `départ` et `position` + distance entre `position` et *i*) et `f_i=g_i+H[i]`.
- Si la nouvelle distance `f_i` entre `départ` et `arrivée` est inférieure à `SOMMETS[i][3]`, alors `SOMMETS [i][3]=f_i`. On met à jour également la distance entre le sommet `départ` et le sommet `i` : `SOMMETS[i][1]=g_i`. Le sommet `position` est le sommet précédent de `i` : `SOMMETS[i][0]=position`.
- On cherche le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance *f* (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.
- Si `indice` = `position`, on ne peut pas atteindre le sommet d'arrivée.

```
import numpy as np      # bibliothèque numpy renommée np
inf=float("inf")

def init3(départ, n):  # n = nombre de sommets dans le graphe
    # la fonction initialise la liste SOMMETS à partir de départ
    # (int)
    SOMMETS=[]         # initialisation de la liste SOMMETS
    for i in range(n): # i varie entre 0 inclus et n exclu
        if i==départ:
            SOMMETS.append([-1, 0, True, 0])
                # la valeur -1 n'est pas utilisée
                # True : uniquement le sommet de départ est
                # sélectionné
        else:
            SOMMETS.append([-1, inf, False, inf])
                # la valeur -1 n'est pas utilisée car distance
                # infinie
                # False : sommet non sélectionné
    return SOMMETS

def algoA(M, H, départ, arrivée):
    # la fonction permet d'avoir le plus court chemin
    # de départ (int) à arrivée (int) dans la liste L à partir
    # de la matrice d'adjacence M. On récupère la liste SOMMETS
    n=np.shape(M)[0]   # nombre de sommets = nombre de lignes de M
    inf=float("inf")         # float infini
    SOMMETS=init3(départ, n) # initialisation de la liste SOMMETS
    position=départ
```

```
    while (position!=arrivée):
        indice=position
        for i in range(n):   # i décrit tous les sommets
            if SOMMETS[i][2]==False: # sommet non sélectionné
                g_i=SOMMETS[position][1]+M[position, i]
                            # g(i) = coût réel entre départ et i
                h_i=H[i]     # coût estimé entre i et arrivée
                f_i=g_i+h_i
                if f_i<SOMMETS[i][3]:
                    SOMMETS[i][1]=g_i # distance départ->i = g(i)
                    SOMMETS[i][3]=f_i
                                # distance départ->arrivée = f(i)
                    SOMMETS[i][0]=position
                                # sommet précédent sur le chemin
        # recherche du minimum des distances départ->arrivée
        # pour les sommets non sélectionnés
        val_min=inf         # initialisation de val_min à +infini
        for i in range(n):  # i varie entre 0 inclus et n exclu
            if SOMMETS[i][2]==False and SOMMETS[i][3]<val_min:
                indice=i
                val_min=SOMMETS[i][3]  # f(i)
        if indice==position:
            return [],inf   # on n'atteint pas le sommet arrivée
        else:
            SOMMETS[indice][2]=True  # ce sommet est sélectionné
            position=indice          # nouvelle valeur de position

    # liste des sommets parcourus
    i=arrivée
    L=[arrivée]  # L = liste des sommets parcourus en sens inverse
    while (i!=départ):
        i=SOMMETS[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir
                 # la liste des sommets parcourus dans le sens
                 # direct
    return L, SOMMETS[arrivée][1]
```

3.

```
inf=float("inf")
M=np.array([[0,3,2,inf,inf,inf],[inf,0,3,3,2,inf], \
  [inf,3,0,5,4,inf],[inf,inf,inf,0,inf,2], \
  [inf,inf,4,2,0,inf],[inf,inf,inf,2,2,0]])
départ, arrivée=0, 5
H=[5, 2, 3.2, 1.5, 1, 0]
L2, dist2=algoA(M, H, départ, arrivée)
print('Algo A* chemin :',L2,'; distance',dist2)
```

Le programme Python affiche :

```
Algo A* chemin : [0, 1, 3, 5]
distance = 8.0
```

Remarque :

L'algorithme A* utilise une heuristique dont le coût évalué (distance à vol d'oiseau) est toujours inférieur au coût réel. On dit que cette heuristique est admissible. On peut montrer que cet algorithme retourne toujours une solution optimale si elle existe.

On étudiera (dans l'ouvrage de deuxième année) des algorithmes (Bellman-Ford et Floyd-Warshall) permettant de traiter des arcs dont le poids est négatif.

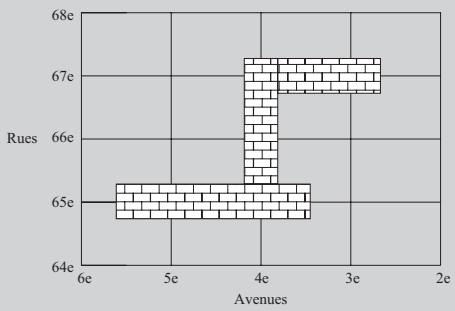
Exercice 14.5 : Variantes de l'algorithme A* – Distance de Manhattan

On considère le graphe orienté $G = (S, A)$ qui représente le quartier de Manhattan. Les routes sont à double sens et certaines sont bloquées à cause de travaux (rectangles avec des briques). Les n sommets du graphe sont les intersections des routes : le sommet 11 représente l'intersection de la 5e avenue avec la 66e rue. On considère que la distance entre deux sommets adjacents vaut 1. On suppose que le nombre d'avenues est égal au nombre de rues.

Position des 25 sommets :

0	1	2	3	4
5	6	~~7~~	~~8~~	9
10	11	~~12~~	13	14
15	~~16~~	~~17~~	18	19
20	21	22	23	24

Les sommets barrés ne sont pas accessibles à cause de travaux.



Une heuristique est un algorithme qui calcule rapidement une solution pouvant être approximative. On utilise la distance de Manhattan pour estimer le coût restant ($h(i)$ = nombre d'avenues et de rues entre le sommet i et le sommet `arrivée`) permettant d'atteindre le sommet `arrivée` à partir du sommet i. On définit la variable `inf=float("inf")` qui représente une distance infinie.

On définit la liste `SOMMETS` contenant les informations suivantes pour chaque sommet :

[sommet précédent sur le chemin, distance parcourue depuis le sommet de départ, sommet sélectionné ou non (booléen vrai ou faux), distance évaluée entre `départ` et `arrivée`]

`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des sommets adjacents.

2. Écrire une fonction `listeH` qui admet comme arguments d'entrée le sommet `arrivée` et le nombre de sommets *n*. Cette fonction retourne la liste $H = \left[h(0), h(1), ..., h(i), ..., h(n-1)\right]$ telle que $h(i)$ = distance de Manhattan entre le sommet *i* et le sommet `arrivée`.

3. L'algorithme glouton BFS (Best First Search : meilleure première recherche) permet de déterminer le plus court chemin entre le sommet `départ` et le sommet `arrivée`.

a) Initialisation de l'algorithme BFS :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance $h(\text{indice})$ entre `indice` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

Écrire une fonction `BFS` qui admet comme arguments d'entrée un dictionnaire `dico`, le sommet `départ` et le sommet `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme BFS.

4. On dispose pour chaque sommet *i* d'une estimation du coût restant pour atteindre le sommet `arrivée` à partir du sommet *i* : $h(i)$ = distance de Manhattan entre le sommet *i* et le sommet `arrivée`. On pose *w* un réel compris entre 0 et 1. On définit la fonction d'évaluation *f* telle que : $f(i) = (1-w) \cdot g(i) + w \cdot h(i)$.

- $g(i)$ est le coût réel du chemin optimal entre `départ` et *i* dans la partie déjà explorée ;
- $h(i)$ est le coût estimé du chemin qui reste à parcourir entre *i* et `arrivée`.

a) Initialisation de l'algorithme :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance *f* (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

Écrire une fonction `MANHATTAN` qui admet comme arguments d'entrée un dictionnaire `dico`, le sommet `départ`, le sommet `arrivée` et le réel `w`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme décrit précédemment.

Écrire le programme principal permettant de déterminer le plus court chemin entre le sommet de départ 10 et le sommet d'arrivée 13. Le programme affichera le chemin suivi ainsi que la distance parcourue pour `w` = 0, `w` = 0.5 et `w` = 1.

5. Comment appelle-t-on les algorithmes lorsque `w` = 0, `w` = 0.5 et `w` = 1 ?

Analyse du problème

La fonction *h* est une fonction heuristique telle que $h(i)$ est le coût estimé du chemin qui reste à parcourir entre le sommet *i* et le sommet `arrivée`. On utilise la distance de Manhattan alors que dans l'exercice précédent « Algorithme A* » on utilise la distance euclidienne (ou distance à vol d'oiseau).

On considère différentes fonctions d'évaluation dans ce problème pour sélectionner un sommet :

- Algorithme glouton BFS (question 3) : la fonction d'estimation $f(i)=h(i)$ renvoie la distance de Manhattan entre le sommet *i* et le sommet `arrivée`.
- Algorithme de la question 4 : la fonction d'évaluation $f(i)=(1-w)\cdot g(i)+w\cdot h(i)$ renvoie une estimation de la distance entre le sommet de départ et le sommet d'arrivée en passant par un sommet *i*.

1.

```
dico={0:[1,5], 1 :[0,2,6], 2 :[1,3], 3 :[2,4], 4 :[3,9],\
  5 :[0,6,10], 6 :[1,5,11], 7 :[], 8 :[], 9 :[4,14],\
  10 :[5,11,15], 11 :[6,10], 12 :[], 13 :[14, 18],\
  14 :[9,13,19], 15 :[10,20], 16 :[], 17 :[],\
  18 :[13, 19, 23], 19 :[14,18, 24], 20 :[15,21,],\
  21 :[20,22], 22 :[21,23], 23 :[18, 22,24], 24 :[19,23]}
```

Voir l'exercice 12.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire – Pile – File – Deque » pour l'utilisation des dictionnaires.

Le sommet 6 est relié aux sommets 1, 5 et 11. La valeur de la clé 6 est la liste des sommets adjacents [1, 5, 11].

Le nombre d'éléments du dictionnaire correspond au nombre *n* de sommets du graphe. On définit `nb_rues` le nombre de rues. Comme le nombre d'avenues est égal au nombre de rues, alors :

```
nb_rues=int(np.sqrt(n))  # n = nombre de sommets
                         # = nombre de rues * nombre d'avenues
```

Remarque : On a accès très facilement à la liste des successeurs d'un sommet.

2. La distance de Manhattan entre un sommet *A* (de coordonnées x_A, y_B) et un sommet *B* (de coordonnées *xB*, *yB*) vaut $|x_B - x_A| + |y_B - y_A|$. Elle correspond au nombre d'avenues et de rues entre le sommet *A* et le sommet *B*. Les abscisses correspondent aux lignes et les ordonnées correspondent aux colonnes. Le sommet 0 a pour coordonnées 0, 0. Le sommet 13 a pour coordonnées 2, 3.

```
import numpy as np   # bibliothèque numpy renommée np
inf=float("inf")

def listeH(arrivée, n):
    # la fonction retourne la liste H telle que H[i] = distance
    # de Manhattan entre le sommet i et le sommet arrivée (int)
    nb_rues=int(np.sqrt(n)) # n = nombre de sommets
                            # = nombre de rues * nombre d'avenues
            # on suppose que nombre d'avenues = nombre de rues
    xB, yB= arrivée//nb_rues, arrivée%nb_rues
            # abscisse et ordonnée du sommet d'arrivée
            # quotient et reste de la division euclidienne
    H=[]     # initialisation de la liste H
    for i in range(0, n) :  # i varie entre 0 inclus et n exclu
        xA, yA=i//nb_rues, i%nb_rues  # abscisse et ordonnée
                                      # du sommet i
            # quotient et reste de la division euclidienne
        y=np.abs(yB-yA)+np.abs(xB-xA) # distance de Manhattan pour
                                      # le sommet i
        H.append(y)
    return H
```

3. On définit une liste `SOMMETS` qui contient pour chaque sommet *i* :

- `SOMMETS[i][0]` : sommet précédent ;
- `SOMMETS[i][1]` : $g(i)$ = distance entre le sommet `départ` et le sommet *i* ;
- `SOMMETS[i][2]` : `True` si le sommet *i* est sélectionné, sinon `False` ;
- `SOMMETS[i][3]` : $f(i)$ = distance entre le sommet `départ` et le sommet `arrivée`.

La méthode gloutonne repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu au sommet `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des sommets non sélectionnés, le sommet suivant `indice` dont la distance de Manhattan entre `indice` et

arrivée est la plus petite. Ce choix n'est pas remis en cause ultérieurement.

On privilégie les sommets « qui semblent » nous rapprocher de la destination.

a) Initialisation de l'algorithme :

On définit la liste H en utilisant la fonction listeH définie dans la question 2.

Tous les sommets sont non sélectionnés (SOMMETS[i][2]=False) sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini : SOMMETS[i][1]=inf et SOMMETS[i][3]=inf. La variable position prend initialement la valeur départ.

b) On définit une boucle while(position!=arrivée) tant que la variable position (sommet appartenant à la frontière entre les sommets sélectionnés et les sommets non sélectionnés) n'est pas égale à la variable arrivée.

- La boucle for i in range(n) avec le test if SOMMETS[i][2]==False permet de parcourir tous les sommets non sélectionnés. On calcule g_i=SOMMETS[position][1]+1 (= distance entre départ et position + distance entre position et *i*) et f_i=H[i].
- Si la nouvelle distance f_i entre départ et arrivée est inférieure à SOMMETS[i][3], alors SOMMETS[i][3] = f_i. On met à jour également la distance entre le sommet départ et le sommet i : SOMMETS[i][1] = g_i. Le sommet position est le sommet précédent de i : SOMMETS[i][0] = position.
- On cherche le sommet indice (parmi les sommets non sélectionnés) pour lequel la distance *f* (indice) entre départ et arrivée est la plus petite.
- Ce sommet indice définit alors la nouvelle valeur de la variable position et ce sommet devient sélectionné.
- Si indice = position, on ne peut pas atteindre le sommet d'arrivée.

```
def init4(départ, n):  # n = nombre de sommets dans le graphe
    # la fonction initialise la liste SOMMETS à partir
    # de départ (int)
    SOMMETS=[]         # initialisation de la liste SOMMETS
    for i in range(n): # i varie entre 0 inclus et n exclu
        if i==départ:
            SOMMETS.append([-1, 0, True, 0])
           # la valeur -1 n'est pas utilisée
           # True : uniquement le sommet de départ est sélectionné
        else:
            SOMMETS.append([-1, inf, False, inf])
           # la valeur -1 n'est pas utilisée car distance infinie
           # False : sommet non sélectionné
    return SOMMETS
```

```
def BFS(dico, départ, arrivée):
    # la fonction permet d'avoir le plus court chemin
    # de départ (int) à arrivée (int) dans la liste L
    # à partir du dictionnaire dico.
    # On récupère la liste SOMMETS
    n=len(dico)               # nombre de sommets
    H=listeH(arrivée, n)
    inf=float("inf")          # float infini
    SOMMETS=init4(départ, n)  # initialisation de la liste SOMMETS
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(n):    # i décrit tous les sommets
            if SOMMETS[i][2]==False: # sommet non sélectionné
                L=dico[position]
                if i in L:    # position et i sont adjacents
                    g_i=SOMMETS[position][1]+1
                              # g(i) = coût réel entre départ et i
                    f_i=H[i]  # coût estimé entre i et arrivée
                    if f_i<SOMMETS[i][3]:
                        SOMMETS[i][1]=g_i
                              # distance départ->i = g(i)
                        SOMMETS[i][3]=f_i
                              # fonction d'évaluation f(i) = H[i]
                        SOMMETS[i][0]=position
                              # sommet précédent sur le chemin
        # recherche du minimum des valeurs de f(i) = H[i]
        # pour les sommets non sélectionnés
        val_min=inf         # initialisation de val_min à +infini
        for i in range(n):  # i varie entre 0 inclus et n exclu
            if SOMMETS[i][2]==False and SOMMETS[i][3]<val_min:
                indice=i
                val_min=SOMMETS[i][3]  # f(i)
        if indice==position:
            return [],inf   # on n'atteint pas le sommet arrivée
        else:
            SOMMETS[indice][2]=True  # ce sommet est sélectionné
            position=indice          # nouvelle valeur de position

    # liste des sommets parcourus
    i=arrivée
    L=[arrivée]  # L = liste des sommets parcourus en sens inverse
    while (i!=départ):
        i=SOMMETS[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des sommets parcourus dans le sens direct
    return L, SOMMETS[arrivée][1]
```

4. C'est le même algorithme que précédemment. On change la fonction d'évaluation : `f_i = (1-w)*g_i+w*h_i` au lieu de `f_i = h_i` pour sélectionner le sommet suivant lorsqu'on est rendu au sommet `position`. On retrouve l'algorithme de la question 2 avec `w = 1`.

```
def MANHATTAN (dico, départ, arrivée, w):
    # la fonction permet d'avoir le plus court chemin
    # de départ (int) à arrivée (int) dans la liste L
    # à partir du dictionnaire dico.
    # w est compris entre 0 et 1.
    # On récupère la liste SOMMETS
    n=len(dico)              # nombre de sommets
    H=listeH(arrivée, n)
    inf=float("inf")         # float infini
    SOMMETS=init4(départ, n) # initialisation de la liste SOMMETS
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(n):   # i décrit tous les sommets
            if SOMMETS[i][2]==False: # sommet non sélectionné
                L=dico[position]
                if i in L:   # position et i sont adjacents
                    g_i=SOMMETS[position][1]+1
                             # g(i) = coût réel entre départ et i
                    h_i=H[i]  # coût estimé entre i et arrivée
                    # w=0:Dijkstra, w=0.5:algorithme A*, w=1:BFS
                    f_i=(1-w)*g_i+w*h_i
                    if f_i<SOMMETS[i][3]:
                        SOMMETS[i][1]=g_i
                            # distance départ->sommet i = g(i)
                        SOMMETS[i][3]=f_i
                            # distance départ->arrivée = f(i)
                        SOMMETS[i][0]=position
                            # sommet précédent sur le chemin
        # recherche du minimum des valeurs de f(i)
        # = distance départ->arrivée
        # pour les sommets non sélectionnés
        val_min=inf
        for i in range(n):
            if SOMMETS[i][2]==False and SOMMETS[i][3]<val_min:
                indice=i
                val_min=SOMMETS[i][3]  # f(i)
        if indice==position:
            return [],inf   # on n'atteint pas le sommet arrivée
        else:
            SOMMETS[indice][2]=True  # ce sommet est sélectionné
            position=indice          # nouvelle valeur de position

    # liste des sommets parcourus
    i=arrivée
    L=[arrivée]  # L = liste des sommets parcourus en sens inverse
    while (i!=départ):
        i=SOMMETS[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des sommets parcourus dans le sens direct
    return L, SOMMETS[arrivée][1]
```

```
# initialisation du programme
départ, arrivée=10, 13  # sommet de départ et sommet d'arrivée
L2, dist2=BFS(dico, départ, arrivée)
print('Algo BFS : chemin :', L2, '; distance', dist2)
w=0
for i in range(3):
    L2, dist2=MANHATTAN(dico, départ, arrivée, w)
    print('w =',w, '; chemin :', L2, '; distance', dist2)
    w=w+0.5
```

5. On retrouve plusieurs cas particuliers :

- `w` = 0 : algorithme de Dijkstra. L'algorithme devient une recherche en largeur sans stratégie d'exploration des sommets ;
- `w` = 0.5 : algorithme A* ;
- `w` = 1 : algorithme glouton BFS.

Si on souhaite aller de Bordeaux à Lyon, l'algorithme A* va plutôt explorer les sommets vers l'est qui ont une distance plus faible que les autres sommets alors que l'algorithme de Dijkstra fait une recherche en largeur sans stratégie d'exploration des sommets.

Le programme Python affiche les résultats suivants pour `départ` = 10 et `arrivée` = 13 :

- `w` = 0 ; chemin : [10, 15, 20, 21, 22, 23, 18, 13] ; distance = 7 ;
- `w` = 0.5 ; chemin : [10, 15, 20, 21, 22, 23, 18, 13] ; distance = 7 ;
- `w` = 1 ; chemin : [10, 11, 6, 1, 2, 3, 4, 9, 14, 13] ; distance = 9.

L'algorithme BFS (`w` = 1) ne donne pas la solution optimale : il privilégie le chemin passant par 11 qui semble plus près de 13 mais qui nécessite un long contournement pour atteindre effectivement 13.

Les algorithmes de Dijkstra et A* utilisent une heuristique dont le coût évalué (distance de Manhattan) est toujours inférieur ou égal au coût réel. On dit que ces heuristiques sont admissibles. On peut montrer que ces deux algorithmes retournent toujours une solution optimale si elle existe. Il peut y avoir plusieurs solutions optimales de même coût.

Dans la méthode gloutonne, on ne revient pas sur un choix local optimal alors que, avec la programmation dynamique étudiée en deuxième année, on peut revenir sur les choix précédents. L'algorithme est glouton pour toutes les valeurs de `w`.

On étudiera en deuxième année des algorithmes (Bellman-Ford et Floyd-Warshall) permettant de traiter des arcs dont le poids est négatif.

Index

L

M, N

O, P

R

S, T

V